Secure, Accurate and Privacy-Aware Fully Decentralized Learning via Co-Utility

Jesús Manjón[1], Josep Domingo-Ferrer[1], David Sánchez, Alberto Blanco-Justicia

UNESCO Chair in Data Privacy,
CYBERCAT-Center for Cybersecurity Research of Catalonia,
Department of Computer Engineering and Mathematics,
Universitat Rovira i Virgili
Av. Països Catalans 26, E-43007, Tarragona, Catalonia

Abstract

Fully decentralized learning is a setting in which each peer in a P2P network trains a machine learning model with the help of the other peers. Each peer acts as a model manager by periodically sending her current model to other peers, who answer by returning model updates they compute on their private data. This creates a tension among privacy, accuracy and security. The privacy risk is that model updates returned by a peer can leak some of the peer's private data. Unfortunately, distorting model updates to protect privacy works against the accuracy of the trained model. On the other hand, aggregating the updates of several peers and then sending the aggregate to the model manager may preserve privacy but it goes against security, because the model manager cannot filter out individual bad updates. Also, peers are autonomous and hence it cannot be taken for granted that they will honestly supply model updates to help the model manager train her model. To reconcile accuracy, privacy and security, we present a fully decentralized learning protocol such that: i) it allows perfectly accurate individual updates to be returned by peers to the model manager in a privacy-preserving manner; ii) it is co-utile by design, that is, it incentivizes rational peers to follow the protocol without deviating. The latter feature discourages rational attacks that might compromise security and it also deters free riding, thereby ensuring the sustainability of the protocol.

Keywords: Fully decentralized learning, privacy, security, accuracy, co-utility, ethics by design.

Email addresses: jesus.manjon@urv.cat (Jesús Manjón), josep.domingo@urv.cat (Josep Domingo-Ferrer), david.sanchez@urv.cat (David Sánchez), alberto.blanco@urv.cat (Alberto Blanco-Justicia)

[1]Equal contribution.

1. Introduction

Fully decentralized learning (FDML) is the extreme form of decentralized machine learning [15, 6, 22, 14, 19, 16, 20, 24]. In FDML, each peer in a peer-to-peer (P2P) network trains a deep learning model with the help of the other peers. In her role as a model manager, each peer periodically sends the current model to other peers and the latter return model updates based on the private data they hold.

Unlike in the related federated learning (FL, [17]), where there is a fixed split between a model manager and the workers computing model updates based on their private data, in FDML all nodes may be managers of their own models and workers for others' models; this is why nodes are called peers and FDML can also be termed P2P learning. In FDML peers exchange messages without any central coordination. The P2P network topology can be a complete graph or an arbitrarily connected graph G. FDML can be deterministic (peers exchange messages with neighbors chosen in a deterministic manner) or randomized (peers exchange messages with a random set of neighbors).

For example, if learning a model via P2P stochastic gradient descent (SGD), each node in G maintains its own model parameters and alternates between a local SGD step and a weighted averaging among neighbors. The strong point of FDML is that it is more robust and scalable as the number of nodes increases. Note that in centralized machine learning, respectively in FL, an attacker who wants to disrupt the learning process only needs to "intoxicate" the model manager with bad data, respectively bad updates. In contrast, in P2P learning it is harder for the attacker to disrupt a significant proportion of learning processes.

In FDML a conflict between accuracy, security and privacy arises. On the one hand, model updates returned by a peer can leak some of the peer's private data [18]. A countermeasure that can be implemented by the peers themselves to protect their privacy is to distort their updates via, *e.g.*, differential privacy [21]. Alas, this distortion works against accuracy of the trained model. Another countermeasure consists in securely aggregating the updates of several peers (*i.e.*, via secure multiparty computation) and then sending only the update aggregation to the model manager [4]. However, this goes against security, because the model manager cannot filter out individual bad updates. Additionally, peers are autonomous and it cannot be taken for granted that they will help the model manager by supplying honest model updates. This may compromise the sustainability of the network.

Contribution and plan of this paper

The notion of *co-utility* was introduced in [9] to ensure mutually beneficial cooperation among peers in decentralized protocols. In this paper we present a co-utile approach to fully decentralized machine learning that has the following properties:

- Perfectly accurate individual updates can be returned by peers to the model manager in a privacy-preserving manner.

- Rational (that is, self-interested) peers can be expected to follow the FDML protocol we propose without deviating, thanks to a tit-for-tat mechanism that renders the protocol naturally co-utile. This makes our proposal sustainable and robust against rational security attacks.

Our experiments show that peers receive good updates for their respective models from other peers if and only if they compute good updates for the other peers' models. Furthermore, our approach based on tit-for-tat incurs less overhead to achieve co-utility than protocols based on decentralized global reputation.

Section 2 gives background on co-utility and on related work using decentralized global reputations to enforce co-utility. Section 3 introduces a co-utile protocol for FDML. Section 4 justifies that the proposed protocol suite achieves co-utility (and thus is rationally sustainable). Section 5 shows that it satisfies privacy for the inputs and the outputs, and that correct computation is the best option for rational peers. Section 6 presents experimental results. Finally, Section 7 summarizes conclusions and future research avenues.

2. Background

A self-enforcing protocol is *co-utile* [9] if it results in mutually beneficial collaboration between the participating agents. More specifically, a protocol Π is co-utile if and only if *the three* following conditions hold: (i) Π is self-enforcing; (ii) the utility derived by each agent participating in Π is strictly greater than the utility the agent would derive from not participating; and (iii) there is no alternative protocol Π' giving greater utilities to all agents and strictly greater utility to at least one agent.

The first condition ensures that if participants engage in the protocol, they will not deviate. The second condition is needed to ensure that engaging in the protocol is attractive for everyone. The third condition can be rephrased in game-theoretic terms by saying that the protocol is a Pareto-optimal solution of the underlying game.

We can find situations in which co-utility arises naturally. For example, this happens when agents querying a database or a web search engine (WSE) want to protect against profiling [9]: it is rational for an agent P_1 wishing to submit a query to the WSE to ask another agent, say P_2, to do the submission on P_1's behalf, and it is rational for P_2 to accept to do so if P_1's query is useful as "noise" to blur P_2's real profile of interests. Another example is multiparty secure computation in cases in which the output sought by every participant P_i critically depends on P_i's input and computation being correct (*e.g.* as in the millionaires' problem [25] or in private set intersection [10]). In such cases, cooperation between participants is a game-theoretic equilibrium, which results in co-utility. Unfortunately, in most cases, "negative" incentives (like costs or privacy loss) may override positive incentives for the agents to follow the rules. Hence, artificial positive incentives may need to be added to the agents' utility functions to compensate negative incentives and thereby spark co-utility.

Reputation is a convenient artificial incentive for co-utility: an agent that adheres to the rules is rewarded with a reputation increase, and an agent that deviates is punished by a reputation decrease. A high reputation entails some benefits that are defined in an application-dependent manner, whereas a bad reputation may imply being banned from the offered service. In [8] a co-utile (and, hence, sustainable) version of the EigenTrust decentralized reputation protocol [12] was presented. By means of this protocol, a global (i.e., system-wide) reputation value is computed for each of the participating peers. Using global reputations, [7] presented a co-utile protocol for federated learning. In order to preserve the peer-to-peer (P2P) computing paradigm, global reputations were also computed in a P2P decentralized way and their correctness was ensured by co-utility as well. In a nutshell, the reputation of each peer was redundantly computed by a set of randomly chosen peers named *accountability managers* (AM) as a result of the peer's actions (like sending good or bad updates to the model manager). The random choice of AMs thwarted collusion (because the AMs in the set assigned to a certain peer were unlikely to know the peer or each other), whereas the redundancy of AMs allowed neutralizing malicious behaviors (because the false reputation computed by a malicious AM would not match the reputations computed by the other AMs).

However, decentralized management of global reputations complicates protocols and entails a significant cost: in [7] reputation management requires any reputation updates for a peer to be sent to the peer's set of AMs. Moreover, to obtain consistent global reputations, values are normalized into the range $[0, 1]$ by dividing them by the largest reputation in the network; to that end, at each epoch AMs need to broadcast all reputation values greater than 1.

The fact that we tackle here fully decentralized machine learning (unlike [7], which was about federated learning), makes the complexity of the above reputation management even more unaffordable. Indeed, in FDML many models are being learned in parallel, rather than a single model being learned under the control of a designated manager as in FL. In the next sections, we craft a more efficient way to incentivize co-utility in FDML.

3. Co-utile FDML based on tit-for-tat

Global reputation-based incentives for workers are needed in federated learning, due to the divide between the model manager and workers: since all workers contribute to a single model, there should be a common –global– way to account for their behavior. However, in fully decentralized learning peers can be managers of their own models and at the same time be workers for other peers' models. In this symmetric setting, co-utility can be attained in a more efficient way, specifically using tit-for-tat and local reputations. This is the idea underlying Protocol 1. Before starting the protocol description, we summarize in Table 1 the notation used in the rest of this paper.

In Protocol 1, every time a peer P_i receives a model m_k from another peer P_k, P_i computes her update u_{ki} on m_k. Such an update can be viewed as a "currency" that can be redeemed at P_i against an update on P_i's model m_i. It

Table 1: Notation in this paper

Notation	Concept	Notation	Concept
P_i, P_k, P_l, P_ℓ	Peers	i, k, l, ℓ	Pseudonyms of P_i, P_k, P_l, P_ℓ, respectively
m_i	Model managed and learned by P_i	u, u_{**}	Model updates
m^u	Model m after updating it with update u	u_{ki}	Update on P_k's model computed by P_i
u_{i*}	Update on P_i's model computed by some other peer	t_i	Number of updates computed by P_i on her private data
κ_i	Privacy parameter such that there is a probability $1/\kappa_i$ that an update exchanged by P_i has been computed by P_i on her private data	θ	Centroid of a batch of updates
$Dest(\cdot)$	Function that computes the pseudonym of the peer to whom the message in the argument is to be sent	$Sign_i(\cdot)$	Function computing P_i's signature on the argument
$Ver_i(\cdot,\cdot)$	Verification function for P_i's signature	$H(\cdot)$	Cryptographic hash function
$Enc_i(\cdot,\cdots)$	Function that encrypts the message and the nonce in the arguments under P_i's public key	R_i	Request for updates message sent by P_i
τ	Epoch identifier	$r, r_2, r^*, \cdots$	Random nonces
$U^i_{j\to k}$	Update message that contains $Enc_i(u_{ij})$ signed and sent by P_j to P_k	$U^i_{j\to *}$	Same as above but sent by P_j to any peer
$U_{u_{i*}}$	Set of update messages containing u_{i*}	$S_{u_{i*}}$	Set of peers having sent update messages in $U_{u_{i*}}$
$C_i = \{c_{i1}, ..., c_{in}\}$	Local reputations managed by P_i on the other peers she has interacted with	δ	Reputation quantum for punishment and reward
T_i	Interaction threshold, which is used by P_i to decide whether to interact with other peers		

is important to note that, when P_i sends her model m_i, m_i carries P_i's identity so that every peer receiving m_i knows it is managed by P_i. In contrast, when P_i returns an update u_{ki} on m_k, u_{ki} carries P_k's identity but *not* the identity of the update generator P_i. In fact, the protocol prevents P_k from directly receiving u_{ki} from P_i, in order to protect the update generator's privacy. Indeed, u_{ki} might leak information on P_i's private data, and hence dissociating u_{ki} from P_i's identity protects the latter peer's privacy.

To account for the actions of the peers involved in the interactions, Protocol 1 relies on local reputations: each peer P_i maintains local reputation scores $C_i = \{c_{i1}, ..., c_{in}\}$, which represent P_i's opinion on the other peers she has interacted with. These local reputations are never shared. Hence, the interactions between two peers depend only on their corresponding local reputations.

Local reputations take values within the range $[0, 1]$. Depending on the actions of P_k during the execution of Protocol 1, P_i's opinion on peer P_k (c_{ik}) can either improve or worsen. Generating a bad update for P_i or duplicating updates (to save computation) decreases the generator's reputation c_{ik} by a certain quantum δ (*hard punishment*). Furthermore, unintentionally forwarding bad or duplicated updates to P_i slightly decreases the forwarder's reputation c_{ik} by a fraction of δ (*soft punishment*). On the other hand, helping a good update reach the model manager or helping disseminate the model brings a reward (fraction of δ) to one of the helping peers.

After each tit-for-tat phase, every peer P_i computes an interaction threshold T_i. This threshold defines the minimum reputation value P_i requires to another peer in order to *trust* him and, therefore, interact with him. The threshold is computed as the average of the local reputations managed by a peer minus their standard deviation:

$$T_i = \overline{C_i} - \sigma(C_i) \tag{1}$$

In what follows, we will use the term *trusted* (by a certain peer P_i) to denote a peer P_k with a local reputation equal or above T_i; that is, "P_i trusts P_k" means $c_{ik} \geq T_i$. Inversely, we will use the term *untrusted* for a peer P_l such that $c_{il} < T_i$.

Protocol 1 works as follows. At each epoch, each P_i disseminates her model m_i to some of the others peers, trusted or not (subprotocol DISSEMINATE(P_i,m_i)). Since P_i would anyway be unable to prevent her model from reaching peers untrusted by her (because dissemination is multi-hop), it makes no sense for P_i to limit dissemination to her trusted peers.

After disseminating her model, each peer P_i computes updates based on her local private data for as many models as the peer has received from trusted peers.

Next come two tit-for-tat blocks:

- Up to the end of the first half of the epoch, a "privacy tit-for-tat" is run. In parallel, every peer P_i peer calls subprotocol EXCHANGE_UPDATE(P_i,P_l) to exchange the updates she holds with other trusted peers P_l. The updates that a peer P_i holds may have been computed by P_i or may have been

Protocol 1: CO-UTILE FDML BASED ON TIT-FOR-TAT

Peer P_i calls DISSEMINATE(P_i,m_i);
for *each model* m_k *received* **do**
    if P_i *trusts* P_k **then**
        P_i computes an update u_{ki}
Let t_i be the number of updates computed by P_i;
`/* Privacy tit-for-tat */`
for $j = 1$ **to** $\kappa_i \times t_i$ **do**
    P_i randomly selects a P_l such that $c_{il} \geq T_i$;
    P_i calls EXCHANGE_UPDATE(P_i,P_l);
P_i waits until the first half of the epoch has elapsed;
P_i hard-punishes any peer to whom P_i has sent an update in the privacy tit-for-tat without being reciprocated with another update from that peer;
`/* Learning tit-for-tat */`
for *each* u_{k*} *received by* P_i *from a trusted peer and not yet forwarded* **do**
    if $c_{ik} \geq T_i$ **then**
        P_i tells P_k she has an update on P_k's model and requests an update from P_k, if possible an u_{i*} on P_i's model;
        if P_i *receives an update from* P_k **then**
            P_i sends u_{k*} to P_k;
for *each good update* u_{i*} *received by* P_i *from a trusted peer* P_j **do**
    P_i updates her model m_i with u_{i*}, where in case of duplication u_{i*} is considered only once;
    P_i rewards P_j;
    P_i also rewards the first destination peer that participated in the model dissemination;
P_i hard-punishes any peer to whom P_i has sent an update in the learning tit-for-tat without being reciprocated with another update from that peer;
for *each* P_k *from whom* P_i *has received a bad* u_{i*} **do**
    P_i calls PUNISH_BAD(P_i, u_{i*}, P_k);
for *each duplicate update* u_{i*} **do**
    P_i calls PUNISH_DUPLICATE(P_i, u_{i*}).

just received from other trusted peers (P_i systematically discards updates received from untrusted peers due to their demonstrated bad behavior in previous epochs).

Hence, when a peer P_l receives an update from P_i, P_l cannot be sure that the update was really computed by P_i, which protects the privacy of P_i's private data. See the details of EXCHANGE_UPDATE in Section 3.2 below.

If a peer P_i has computed t_i updates, she tries to exchange $\kappa_i t_i$ updates, where κ_i is a privacy parameter selected by P_i.

As a result, the probability that an update exchanged by P_i has actually been computed by P_i on her private data is at most $1/\kappa_i$; thus, the greater κ_i, the more work for P_i, but also the more privacy.

- The second tit-for-tat is called "learning tit-for-tat" and starts in the second half of the epoch. In parallel, peers send to the corresponding trusted model managers all the updates that: (i) they got from trusted peers; and (ii) they did not forward in the privacy tit-for-tat.

  Due to the first tit-for-tat, the peer P_i that sends an update u_{k*} to P_k is not in general the peer having computed the update, which is good for privacy. On the other hand, if P_i sends an update u_{k*} to P_k (tit), P_i expects to receive from P_k an update that, ideally, should be u_{i*} (tat). However, if P_k does not have any u_{i*}, P_i will also accept any other update from P_k, say u_{l*}. This will enable P_i to contact another model manager (P_l). By repeating this exchange of updates, P_i increases the chance of receiving updates u_{i*} on her own model. At a network level, this will make it possible for most updates to reach their corresponding model managers.

  Before sending u_{k*} to P_k, P_i asks P_k whether she has got any update for her (ideally u_{i*}).

  If P_k sends an update to P_i, P_k expects to receive u_{k*} from P_i; if this does not happen, then P_k hard-punishes P_i by decreasing her local reputation drastically.

  If P_k is not able to send any update to P_i (because she has run out of them), P_i selects another update to be exchanged with its corresponding model manager, and keeps u_{k*} to be exchanged later with other peers holding u_{i*}.

  At the end of the epoch, P_i checks whether the updates u_{i*} she has received as a model manager are good (see below). Then P_i uses each good update to improve her model m_i. *If a good update has been received several times (duplicate), it is only used once.* Also, P_i rewards the last peer that forwarded the update, and the first destination peer that helped disseminate the model.

  Finally, P_i launches punishment protocols involving those peers P_k from whom P_i has received bad or duplicate updates u_{i*}. The peers who computed these updates will eventually be hard-punished. The reason is that

bad or duplicate updates correspond to malicious behaviors that should be discouraged: bad updates can bias the model or prevent it from converging [11], whereas duplicate updates signal that some peer is trying to avoid the cost of calculating updates on her own private data[2].

After the two tit-for-tat blocks, each peer will have received a number of updates for her own model that is highly correlated with the number of updates computed by her for the models of other peers (see Figure 9 in the experimental section). In this way, the protocol fosters the computation of new updates.

Figure 1 shows an example of the execution of Protocol 1. P_0 and P_3 disseminate their respective models, which reach P_1 and P_3 in the case of P_0's model, and P_0, P_1 and P_2 in the case of P_3' model. Each peer computes an update and, afterwards, the "privacy tit-for-tat" phase begins. In this phase, the updates are exchanged among the peers and, at the end, P_0 owns u_{31}, P_1 owns u_{32} and u_{03}, P_2 owns u_{30} and P_3 owns u_{01}. For the sake of clarity, in the "learning tit-for-tat" phase we have just shown the interaction between P_0 and P_3. Since P_0 trusts P_3, P_0 tells P_3 that she has an update for her, P_3 sends u_{01} to P_0 and finally, P_0 sends u_{31} to P_3. The remaining updates would be exchanged among the other peers in a similar way. After receiving an update, each peer should check it. In our example, P_0 checks u_{01}. If u_{01} is a good update, P_0 updates her model and rewards both the first disseminating peer (not shown in the diagram, say P_x) and the last forwardee (P_3). If u_{01} is a bad update or it has been duplicated, P_0 launches the corresponding punishment protocol.

In the next subsections, we describe the auxiliary (sub)protocols that appear in Protocol 1, namely DISSEMINATE, UPDATE_EXCHANGE, update forwarding, PUNISH_BAD, and PUNISH_DUPLICATE.

3.1. Model dissemination

To describe the DISSEMINATE subprotocol, we first define some notation.

Addressing in our P2P network is designed to function as a distributed hash table (DHT) using consistent hashing [13], rendezvous hashing [23], or similar approaches. In other words, all peers in the network are able to determine, using an agreed-upon cryptographic hash function denoted by $Dest(\cdot)$, which peer is responsible for processing a specific message. Additionally, peers are identified by a pseudonym consisting in an integer in the range $(0, 2^\lambda - 1)$ (*e.g.* for $\lambda = 160$ we have a 160-bit integer). We use $Sign_i(\cdot)$ to denote a function computing P_i's signature on the argument using her private key. Likewise, we denote by $Ver_i(\cdot, \cdot)$ the signature verification function for P_i's signature using P_i's public key. For correctness, $Ver_i(m, Sign_i(m)) = 1$ for any message m.

[2]One might also think of "smart duplicators", who slightly change another peer's update to forgo calculation costs while escaping the duplicator punishment. However, this behavior is not possible because, as it will be seen below, updates are encrypted under the model manager's public key, and hence if they are altered in transit, no proper update will be recovered by the manager, who will count the resulting gibberish as a bad update and will punish the "smart duplicator" for having generated it.

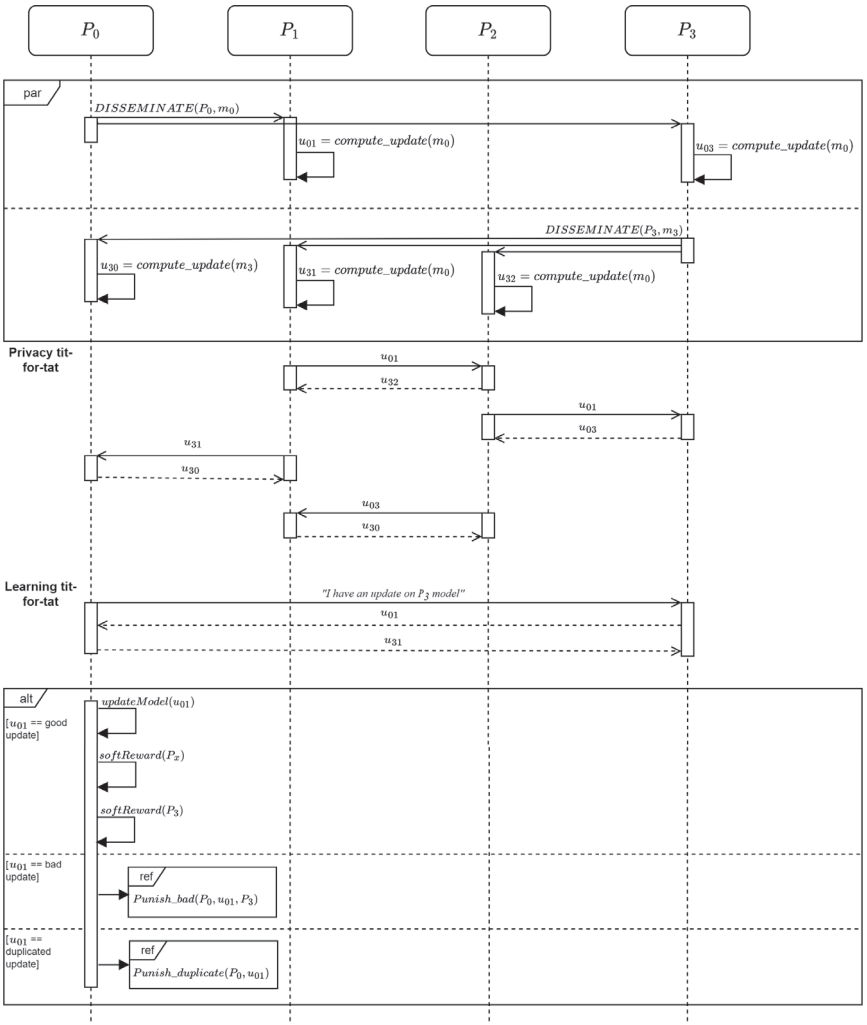


Figure 1: Sequence diagram showing the interactions among peers in an example execution of Protocol 1

Note that the identifier i of peer P_i is deterministically linked to P_i's public key, and so all peers are able to obtain P_i's public key from her identifier. Finally, $H(\cdot)$ is a cryptographic hash function.

DISSEMINATE(P_i, m_i) works as follows:

1. Peer P_i constructs a request for updates message
$$R_i = \langle i, \tau, m_i, Sign_i(H(i, \tau, m_i)), r \rangle, \quad (2)$$
where i is P_i's pseudonym, τ is an epoch identifier, m_i is P_i's model, and r is a random nonce.
2. P_i computes $j = Dest(R_i)$.
3. P_i sends R_i to $P_{Dest(R_i)}$ who, upon receiving the message, checks that (i) she is the intended receiver by ensuring her identifier matches $Dest(R_i)$, (ii) the epoch identifier τ is correct, (iii) she has not received another request for updates for the same model m_i in the current epoch, and (iv) the signature on i, τ and m_i is valid.
4. If any of the previous checks fails, $P_{Dest(R_i)}$ drops R_i.
Otherwise, $P_{Dest(R_i)}$ appends a new sequence number r_2 to obtain $R_i^2 = \langle i, \tau, m_i, Sign_i(H(i, \tau, m_i)), r, r_2 \rangle$, and then forwards R_i^2 to $P_{Dest(R_i^2)}$. Nothing prevents $P_{Dest(R_i)}$ from disseminating m_i to additional destination peers, by replacing r_2 with other sequence numbers.

The rationale of the DISSEMINATE subprotocol is as follows.

- The identifiers and the public keys of peers are linked to each other. Hence, peers cannot maliciously alter their identifiers.
- Due to the above and to the use of a cryptographic hash function (resistant to preimage attacks), a malicious peer P_i is prevented from choosing a designated target peer P_j as a destination for P_i's requests for updates (in view of attacking P_j's privacy by obtaining updates computed on P_j's private data). Indeed, R_i contains i, τ, m_i, a signature on i, τ and m_i, and a nonce r. Since i and τ cannot be altered and the signature has to be valid for the message to be accepted by P_j, P_i needs to find either a format-conforming m_i^*, a nonce r^*, or a combination of both such that $j = Dest(i, \tau, m_i^*, Sign_i(H(i, \tau, m_i^*)), r^*)$, which is computationally infeasible for cryptographically secure hash functions. The same holds for the second hop.

3.2. Update exchange

In the EXCHANGE_UPDATE(P_i, P_l) subprotocol, a peer P_i seeks to exchange an update with another peer P_l that is trusted by P_i.

When exchanging updates during the privacy tit-for-tat, *peers want to avoid receiving repeated updates*, that is, updates the peers had themselves generated or received from other peers in previous exchanges. Note that avoiding repeated updates also avoids loops in the path from the update generator to the model manager for whom the update is intended.

The reason to avoid repeated updates is that it is dangerous for a peer to forward an update more than once. Indeed, forwarding more than once due to a loop can hardly be distinguished from forwarding more than once due to malicious update duplication. In fact, subprotocol PUNISH_DUPLICATE (Section 3.5) punishes any peer that forwards the same update more than once.

Therefore, UPDATE_EXCHANGE(P_i,P_l) operates in the following way:

1. P_i shows to P_l the hash images $H(u)$ of all updates u computed by P_i or received by P_i, such that u has not been forwarded yet.
2. P_l shows to P_i the hash images $H(u')$ of all updates u' computed by P_i or received by P_i, such that u' has not been forwarded yet.
3. If P_i finds a hash $H(u_{**})$ among those shown by P_l that does not correspond to any update previously held by P_i, then P_i requests to P_l the update u_{**}.
4. If P_l finds a hash $H(u'_{**})$ among those shown by P_i that does not correspond to any update previously held by P_l, then P_l requests to P_i the update u'_{**}.

As specified in Line 10 of Protocol 1, if a peer does not reciprocate in the above exchange subprotocol, she will be hard-punished by the other peer at the end of the privacy tit-for-tat.

3.3. Update forwarding

Both the privacy tit-for-tat and the learning tit-for-tat require the exchange of model updates among peers. In order to protect the privacy of the model update generators against the peers in the update's path to the model manager, a peer P_j that computes an update u_{ij} of model m_i owned by P_i encrypts u_{ij} under P_i's public key.

Preserving the update generator's privacy as just discussed must be compatible with being able to trace bad or duplicate updates during the punishment subprotocols. To do so, every update sent by a peer P_j to a peer P_k includes the identities of both peers and is signed by P_j. In this manner, if the update contained in the message is flagged as suspicious, P_k can prove that she did not compute the update, but just forwarded it. The only peer that cannot show a signature from a previous peer is the update generator. Thus, an update message containing an update for the model m_i managed by P_i, sent by peer P_j to P_k at epoch τ and timestamp T takes the form

$$\begin{aligned} U^i_{j\to k} = \langle & j, k, i, \tau, T, \\ & Enc_i(u_{ij}, r), \\ & Sign_j(H(j, k, i, \tau, T, r, Enc_i(u_{ij,r})))\rangle \end{aligned} \tag{3}$$

In Expression (3), r is the nonce inserted during the dissemination phase in the request-for-updates message R_i to which the update corresponds (see Expression (2)). Upon receiving a good update $U^i_{j\to k}$, the model manager P_i

can reward the first destination peer $Dest(R_i)$ that took care of forwarding R_i rather than dropping it. The reward consists in a small increase (by a fraction of δ) of $Dest(R_i)$'s local reputation.

3.4. Punishing bad updates

In this section, we describe the PUNISH_BAD subprotocol for punishing bad updates. We first need to discuss how a bad update can be detected.

There are many approaches in the literature that a model manager can use to decide whether a received update is good or bad (see the surveys [11, 3]). They fall into three main classes:

- *Detection via model metrics [5].* The model manager P_i updates her current model m_i with each single received update u and compares the updated model m_i^u with m_i in terms of accuracy and possibly other performance metrics. If m_i^u offers poorer performance than m_i, then u is probably bad and m_i should be kept rather than m_i^u. This detection approach requires a validation data set on which the two models can be compared and it takes significant computation for each received update.

- *Detection via update statistics.* In this approach, the model manager observes the statistics of the magnitude of the received updates [26]. As the model converges, the update magnitudes tend to zero. Even before convergence, any update that lies far from the other updates is suspicious. Hence, a way to detect bad updates is to use distances [1, 2]: given a batch of updates, an update u is classified as bad if it is much more distant than the other updates from the centroid θ of the batch. One possible way to quantify what "much more distant" means is as follows: if the distance between u and θ is greater than the third quartile (or greater than a small multiple of the third quartile, say 1.5 times) of the set of distances between updates in the batch and θ, then u is classified as a bad update. Note that this distance-based procedure can only be run after a peer has received a batch of updates. This can be accommodated if P_i waits until the end of the epoch before evaluating her updates.

- *Neutralization of bad updates via special aggregation.* This approach consists in ignoring bad updates rather than seeking to explicitly detect them. It combines the above intuitions of distance-based detection and majority voting. The authors of [2] propose to aggregate the received updates using an aggregation function called Krum that is resilient against up to f bad updates. Basically, the Krum aggregation of a set of updates is the most central update, where the centrality of each update u is computed by taking into account all the other updates in the set *except* the f most distant updates from u. Alternative aggregation functions that are also resistant against outlying updates are the coordinate-wise median and the coordinate-wise trimmed mean, proposed in [26]. The former returns the median for every dimension of the received updates, whereas the latter

returns the mean for every dimension after removing a fraction of the smallest values and a fraction of the largest values.

In our protocol, we want to actually detect bad updates in order to avoid interaction with the malicious peers generating them. Hence, we discard the third approach based on ignoring bad updates. On the other hand, the approach based on model metrics is quite costly and requires validation data. Thus, resorting to update statistics seems the most suitable option in our case.

The PUNISH_BAD(P_i, u_{i*}, P_k) subprotocol is called by P_i upon receiving a bad update u_{i*} from another peer P_k and it works as follows:

1. P_i asks P_k to prove that P_k did not generate u_{i*}.
2. If P_k did not generate u_{i*}, according to the update forwarding subprotocol (Section 3.3), P_k should have received a message $U^i_{l\to k}$ with the structure of Expression (3) and containing u_{i*} from another peer, say P_l. Now:

   P_k calls PUNISH_BAD(P_k, u_{i*}, P_l);

   If P_k decides to help P_i (*e.g.*, because P_k trusts P_i), then P_k sends $U^i_{l\to k}$ to P_i.
3. If P_i receives a proof that P_k did not generate u_{i*}, then P_i soft-punishes P_k by slightly decreasing (by a fraction of δ) her local reputation c_{ik}.
   Else P_i hard-punishes P_k by decreasing P_k's local reputation by δ.

If P_k proves she did not generate the bad update by showing $U^i_{l\to k}$ to P_i, P_k avoids a severe decrease of her local reputation c_{ik}. Yet, c_{ik} is decreased a little. The rationale is to have P_k share some of the damage caused by the bad update, so that P_k has an incentive to launch punishment against P_l. In this way, punishment hops up to the generator of the bad update, who cannot prove she got it from anyone else, and is therefore hard-punished by the peer who directly got the bad update from the generator.

Resuming the example in Figure 1, Figure 2 shows the case where the update received by P_0 (u_{01}) was bad. P_0 asks P_3 to prove that she did not generate u_{01} and P_3 returns the message $U^0_{2\to 3}$ received from P_2 to avoid being hard-punished. P_0 soft-punishes P_3 and P_3 executes the same protocol with P_2. P_2 can also prove she was not the generator of u_{01} and try to execute the same protocol with P_1. Since P_1 was the generator of the bad update, she cannot send a message $U^0_{*\to 1}$ to P_2 and, therefore, P_1 is hard-punished.

An alternative to the above hop-wise punishment would be for the model manager P_i to trace and punish the generator upstream with the help of the intermediate forwarders. While this procedure would allow the model manager P_i to directly punish the generator, it might also be misused: P_i might pretend that a good update is bad in order to find which peer generated it and thereby break the generator's privacy (to the extent that an update can leak information on the generator's private data). A peer P_l can defend against this potentially misusable alternative by never answering bad update claims for another peer P_i to whom P_l did not directly send the update.

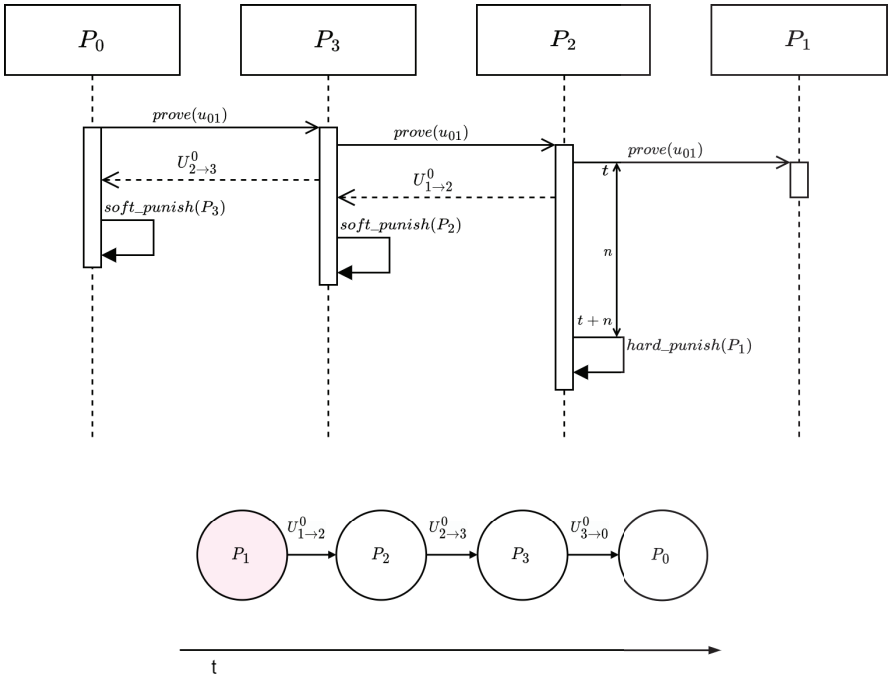


Figure 2: Sequence diagram showing the interactions among peers in an example execution of the Punish_bad subprotocol. The diagram at the bottom shows the path followed by the update.

3.5. Punishing duplicate updates

PUNISH_DUPLICATE(P_i, u_{i*}) aims at punishing the most downstream duplicator(s), that is, the peer(s) closest to P_i in the update forwarding path who duplicated an update to try to save her (their) own computing resources.

PUNISH_DUPLICATE traces the duplicator(s) upstream. It works as follows:

1. If u_{i*} is a duplicate, this means that P_i has received a set of messages $\mathbf{U}_{u_{i*}} = \{U^i_{\ell\to i} | U^i_{\ell\to i}$ contains $u_{i*}\}$ such that $|\mathbf{U}_{u_{i*}}| \geq 2$. Let $S_{u_{i*}} = \{P_\ell | U^i_{\ell\to i} \in \mathbf{U}_{u_{i*}}\}$.
2. Non-questioned_peers = $S_{u_{i*}}$.
3. While Duplicator= not_found and Non_questioned_peers $\neq \emptyset$ do

   If $\exists P_\ell \in$ Non_questioned_peers such that $\mathbf{U}_{u_{i*}}$ contains more than one message $U^i_{\ell\to *}$ then
   (a) P_i hard-punishes all P_ℓ that satisfy the condition;
   (b) P_i sets Duplicator = found;

   Else
   (a) P_i picks $P_\ell \in$ Non_questioned_peers such that P_ℓ sent the most recently timestamped message in $\mathbf{U}_{u_{i*}}$;
   (b) P_i questions P_ℓ by sending $\mathbf{U}_{u_{i*}}$ to P_ℓ;
   (c) If P_i receives from P_ℓ a message $U^i_{e\to\ell}$ that contains u_{i*} then
      i. P_i appends P_e to Non_questioned_peers;
      ii. P_i removes P_ℓ from Non_questioned_peers;
      iii. P_i lets $\mathbf{U}_{u_{i*}} = \mathbf{U}_{u_{i*}} \cup \{U^i_{e\to\ell}\}$;

      Else
      i. P_i hard-punishes P_ℓ;
      ii. P_i sets Duplicator = found.

Note that, unlike in PUNISH_BAD, in PUNISH_DUPLICATE a malicious P_i cannot make false claims: P_i must actually show a set $\mathbf{U}_{u_{i*}}$ of signed messages containing the same update in order to get help in upstream tracing.

Also, in the last step of PUNISH_DUPLICATE, a peer P_ℓ that sent only one message but does not help tracing the upstream duplicator is hard-punished Thus, failing to collaborate to track a duplicator receives the same punishment as being a duplicator. Note that there is no danger that an honest update generator is punished in this way, even if the generator cannot by definition help tracing upstream (because she did not receive the update from anyone else). The reason is that PUNISH_DUPLICATE prioritizes the most recently timestamped updates, so that it will always find the real duplicator before reaching an honest generator.

Hence, if an honest generator comes to be questioned in PUNISH_DUPLICATE, she will hard-punish the questioning peer because the latter should not have questioned her.

Figure 3 shows the case where u_{01} was duplicated in Figure 1. If we assume the updates followed the paths shown in the bottom diagram, P_0 sends all the messages received with the duplicated update ($\mathbf{U}_{u_{01}}$) to P_3 (because $U^0_{3\to 0}$ was

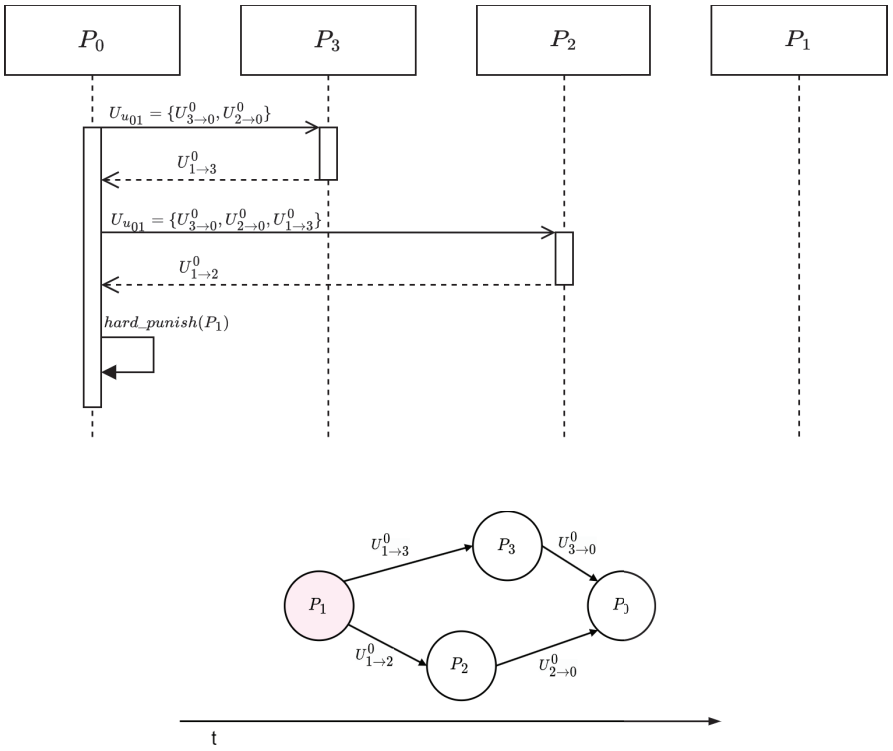


Figure 3: Sequence diagram showing the interactions among peers in an example execution of the PUNISH_DUPLICATE subprotocol. The diagram at the bottom shows the paths followed by the updates.

the last message received, right before $U^0_{2\to 0}$), and P_3 replies with the message she received from P_1 ($U^0_{1\to 3}$). Then, P_0 questions P_2 by sending her all the collected messages. When P_2 sends back the message she received from P_1 ($U^0_{1\to 2}$), P_0 knows that P_1 is the duplicator and hard-punishes her.

4. Co-utility analysis

In this section we will show that Protocol 1 will be adhered to by rational peers. Hence, the rational behavior of peers is to follow the protocol rather than attack it.

4.1. Model dissemination

Step 1 of Protocol 1 is rationally sustainable: without disseminating her model, a peer cannot expect to receive updates from other peers. A different issue is whether the DISSEMINATE subprotocol as described in Section 3.1 will be rationally followed by P_i and the other peers. We justify this.

On P_i's side, it is rational to avoid swamping the other peers with requests for updates: sending each request for updates has a computation cost (the signature) and a communication cost, and P_i knows that destination peers will reject duplicate requests for updates.

On the other hand, the destination peer, say P_j, is incentivized to avoid dropping the request for updates R_i, because if R_i brings a good update to P_i, P_i may be rewarded by P_j with a reputation increase.

However, P_j is interested in making sure she is not deliberately chosen by P_i and hence the target of a privacy attack by P_i. Hence, P_j is motivated to check the format of message R_i, to make sure she was chosen using the hash function and to check that R_i carries P_i's signature, in order to be sure about who is the model manager (this is necessary for the privacy tit-for-tat).

Last, P_j is rationally interested in launching a second dissemination hop to $P_{Dest(R^2_i)}$ in order to forestall an attack by a malicious P_i who would send m_i only to P_j (if the model owner sends her model to a single peer P_j, then the owner knows that any updates on her model have been computed by P_j, which compromises P_j's privacy). The more peers P_j sends m_i to, the more privacy P_j secures for herself.

4.2. Update computation

Step 4 of Protocol 1 is also rationally sustainable: by computing updates for the received models, a peer "mints" a currency that she will be able to use in the tit-for-tats. If P_i purposefully computes a bad update u_{k*} for P_k's model m_k, this is detected in the learning tit-for-tat and P_i is eventually hard-punished by the peer that firstly forwarded the bad update in the privacy tit-for-tat (Step 22). Hence, a peer who computes bad updates will receive fewer and fewer updates for her model from other peers as a consequence of a general worsening of her local reputations c_{*i}. Unless attacking another peer's model has a higher utility than training one's own model, the best option for a peer is to compute good updates in Step 4.

4.3. The privacy tit-for-tat

The privacy tit-for-tat is co-utile by design:

- At Step 8, peer P_i calls the UPDATE_EXCHANGE subprotocol to exchange non-repeated updates with another peer P_l. As described in Section 3.2, in this subprotocol P_i makes a tit move consisting of showing to P_l the hash images of all updates P_i has available for exchange. If no corresponding tat move by P_l follows (in which P_l shows the hash images of her available updates), P_i can exit the subprotocol. Otherwise, if P_i (resp. P_l) finds that P_l (resp. P_i) has a non-repeated update, P_i (resp. P_l) requests that update from P_l (resp. P_i).

  Both peers are rationally interested in getting non-repeated updates: indeed, PUNISH_DUPLICATE punishes a peer if she has sent more than once the same update, no matter whether this comes as a result of message duplication or as a result of a loop.

- In case P_l (resp. P_i) fails to provide the requested update, she will be hard-punished by P_i (resp. P_l) at Step 10. This can be viewed as a "currency" exchange. The exchange can involve updates that P_i and P_l have computed and updates they have received from other peers. The more updates a peer receives from other peers, the more updates she can send other than those she computes herself; the received updates are used as noise to hide the computed updates, which results in higher privacy. Hence, forwarding updates computed by other peers is co-utile.

4.4. The learning tit-for-tat

The learning tit-for-tat is also co-utile by design. Peer P_i makes a tit move by sending an update u_{k*} to P_k (Step 15) after having secured a tat move consisting of an update from P_k (Step 14). When requesting the tat move from P_k (Step 13), P_i expresses her preference for an update u_{i*} on her model.

In case P_k possesses an update u_{i*}, there is no reason why P_k should not send it to P_i; yet if P_k does not have any update u_{i*}, she can send another update to P_i as a valid tat move.

4.5. Model updating

For every good update received by P_i, it is in P_i's interest to use it to update her own model m_i (Step 17). After that, if the good update was sent by P_k, it is in P_i's interest to increase the local reputation c_{ik}, for the sake of keeping accurate records on the other peers' behavior and to increase the number of trusted peers with whom P_i can interact. From the point of view of a newcomer peer P_k, the possibility of being rewarded with a reputation increase is an incentive to cooperate as an honest update generator or forwarder: as soon as the local reputation c_{ik} reaches P_i's interaction threshold T_i, P_k can expect P_i to compute or send to P_k updates for P_k's model.

4.6. Punishment

Finally, it is also in P_i's interest to punish the misbehavior by other peers, in order to discourage it. Punishment involves a decrease of the offender P_k's local reputation, which is a dissuasive punishment. If her local reputation c_{ik} is less than the interaction threshold T_i, P_k knows P_i will not send or compute updates for P_k' model.

Specifically,

- In Step 20, P_i hard-punishes those peers to whom P_i sent an update (in a requested tat-move) without receiving the corresponding tit.
- Although P_i can detect bad or duplicate updates and ignore them, it is in P_i's interest to discourage other peers from generating such updates. In the case of bad updates, P_i's incentive is to save detection work. In the case of duplicate updates, P_i's incentive is to deter bandwidth and processing wastage as a result of duplication.

  Hence:

  - P_i calls PUNISH_BAD to punish every peer P_k from whom P_i has received a bad update. If P_k did not generate that update, P_k's rational decision is to escape the most severe punishment (hard-punishing by P_i) by revealing the name P_l of the peer that sent the bad update to P_k. Even if P_k collaborates, P_k still shares some of the burden of the bad update, because P_k will be soft-punished by P_i, by slightly decreasing her local reputation c_{ik}. This motivates P_k to carry on the punishment upstream, in order to make sure the generator of the bad update gets hard-punished by the forwarder that first got the bad update.

    On the other hand, as pointed out in Section 3.4, a peer's rational behavior is, for privacy reasons, *not* to answer bad update claims from another peer to whom the former peer did *not* directly sent that update. See more detailed explanations at the end of Section 3.4.
  - P_i calls PUNISH_DUPLICATE to punish duplicate updates and thus discourage them. In PUNISH_DUPLICATE, if a peer P_k did not generate a duplicate, it is in P_k's interest to escape punishment by revealing who sent that duplicate to her.

    On the other hand, although PUNISH_DUPLICATE only punishes the most downstream duplicator(s), this is enough. If an upstream peer decides to duplicate an update, there might not be any other duplicator further downstream, in which case she will be punished if at least two peers that have received her duplicates co-operate with the model manager to trace her. Hence duplicating always poses a risk to the duplicator.

5. Privacy and security

In this section we examine how the confidentiality of the peers' private data is protected and how bad and duplicate updates can be detected by model managers.

5.1. Privacy

As mentioned in Section 3 when describing the privacy tit-for-tat, breaking the link between an update and the peer that computed it goes a long way towards guaranteeing that the private data sets of peers stay confidential. We can state the following proposition.

Proposition 1. *In Protocol 1, the probability that a passive attacker (in particular, a peer) receiving updates from P_i can make inferences on P_i's private data is upper-bounded by $1/\kappa_i$.*

Proof. Any peer P_i sends the t_i updates she has computed on her private data set only in the privacy tit-for-tat. Thus, this is the only point in Protocol 1 where the confidentiality of P_i's private data set is at risk. This confidentiality can be violated if two circumstances concur: i) an update computed by P_i can be attributed by a peer P_l receiving it or by an attacker to P_i; ii) that update allows P_l or the attacker to make inferences on P_i's private data set.

Let us look at the attribution probability. In the privacy tit-for-tat, P_i sends a random mix of the t_i updates she has computed with $\kappa_i(t_i - 1)$ updates she has received. Hence, a peer P_l receiving an update from P_i or an attacker intercepting it cannot decide whether the update was computed or merely received by P_i. From P_l or the attacker's point of view, the probability it was computed by P_i is at most $1/\kappa_i$, which is the proportion of updates computed by P_i among the updates sent by P_i. □

While Proposition 1 deals with passive attacks, a model manager might also attempt active attacks against privacy by trying to identify the generator of an update through abuse of the PUNISH_BAD or PUNISH_DUPLICATE subprotocols. More specifically:

- As explained in Section 3.4, the hop-wise operation of subprotocol PUNISH_BAD only allows the model manager to trace one hop upstream. Hence, the model manager cannot reach as far as the update generator.
- In PUNISH_DUPLICATE, the model manager certainly needs to show real duplicates to question peers. Yet, the model manager could decide to proceed further upstream even after finding the real duplicator, in order to reach the update generator. When the model manager hits a peer who does not help tracing further upstream, this peer may be just an uncollaborative peer or the update generator (the manager cannot know which is the case). Even if the manager guesses right that it is the generator, the latter can hard-punish the manager, who runs the risk of the attacked honest generator refusing to compute further updates for the manager: this will

certainly occur if, as a result of the hard punishment, the manager's local reputation falls below the generator's interaction threshold. Hence, such an active attack offers an uncertain outcome for the manager and it is likely to entail hard punishing. Thus, this attack seems hardly rational for the model manager to carry out.

5.2. Security

In a decentralized learning process such as FDML, the main security goal is to ensure that no incorrect updates will be used to update the model being learned.

First of all, note that good updates in our approach are fully accurate, unlike noise-added updates obtained if using differential privacy.

On the other hand, as pointed out in Section 4, the co-utility of Protocol 1 ensures that rational peers are more interested in computing good updates than bad ones. Thus, co-utility helps towards correctness by reducing the proportion of bad updates: having fewer bad updates means that they are more outlying with respect to good updates and hence easier to detect by model managers.

In the event that a bad update reaches a model manager P_i, the latter can detect it using the techniques mentioned in Section 3.4. As pointed out in that section, a distance-based approach [1, 2] is a convenient option provided that the model-managing peer waits to receive a batch of updates before deciding on the quality of each single update.

Regarding duplicate updates, it is straightforward for P_i to detect them if updates are continuous numerical values: it is very unlikely that two different peers generate exactly the same update u_{i*} for m_i. Hence, in case there is more than one update with the same value u_{i*}, this is a duplicate. If updates have low precision or are not continuous, equal updates for m_i may be genuinely computed by different peers. A fix is for each peer computing an update to make it unique by appending a random nonce; in this case, if P_i receives two equal updates with the same nonce, P_i knows they are duplicates.

6. Experimental results

In this section we report experimental work on several aspects. On the one hand, we evaluated how the rate of good updates received by the peers evolves over time depending on whether they compute good or bad updates and on whether they follow the protocol. On the other hand, we assessed the extent to which our approach reduces the communication overhead compared to systems based on decentralized global reputation management.

Note that the contribution of our paper is largely independent of the actual models being learned, and of their particular parameters and performance. What we present is a protocol to incentivize the peers to generate and forward good updates (which improves accuracy), in such a way that updates cannot be linked to the peers generating them (privacy). Further, we show that our protocols are effective at discouraging misbehavior (security). Our mechanism

can be combined with any decentralized learning models and it will help those models to get rid of bad updates and thus improve their performance (note that our protocol does not alter good updates).

Therefore, in this experimental section we use one specific machine learning model on a particular data set just for the sake of illustration. As justified below, the only parameters of the learning process that are relevant to our protocol are the false positive rates and the false negative rates when model managers classify the received updates into good or bad.

6.1. Experimental setting

In our experiments we used the MNIST data set of handwritten digits, which we evenly split among $n = 100$ peers as their local data.

Following the experimental setting from [3], the baseline global model was a neural network where all convolutional layers and the first dense layer used the ReLU activation function. The second dense layer, which was the output layer, used softmax. The loss function was categorical crossentropy. The model substitution rate η was set to 0.25, meaning that in each epoch the model kept information about past training epochs, which made the training process more stable because there were no abrupt changes in the model performance from one epoch to the next. To compensate for the low substitution rate, the training process needed to be longer. Training was carried out for 100 epochs, in which all peers ran Protocol 1. Local updates were computed after 2 epochs of local training, with a batch size of 32. We used distance-based outlier detection to detect bad updates. The accuracy of the main task (classification of MNIST digits) was 97.87% due to the presence of attacks (bad updates), down from 99.6% in the absence of attacks. The outlier detection mechanism was highly effective, with a false negative rate (good updates detected as bad) as low as FNR=3.8%, and a false positive rate (bad updates detected as good) FPR=2.1% on average over all epochs. In other words, only a small proportion of updates generated by honest peers were classified as bad updates and were wrongly punished, and only a small proportion of bad updates were rewarded as good.

While the development and discussion of attack detection mechanisms on FL such as those recalled in Section 3.4 are out of the scope of this paper, high FPR and FNR rates might degrade the performance of our protocol, as they lead to more frequent unjust punishment of good peers and incorrect reward of evil peers.

The model dissemination subprotocol (Section 3.1) requires the requester P_i to find a set of different destination peers by generating sequence numbers and computing $j = Dest(R_i)$ for each of the requests for updates of her model m_i. In case P_i finds a collision, she can choose new sequence numbers and recompute $j = Dest(R_i)$. Note that the more requests P_i wants to send, the more difficult it is for P_i to find peers to directly send her request and, eventually, after the forwards of the other peers, randomly hit different final destination peers for all requests about m_i.

Therefore, the rational behavior of a requester P_i is to limit her number of requests so that the probability of sending all requests to different destination

peers stays above an acceptable threshold, say 0.5. This problem is similar to the birthday problem[3], that consists in computing the probability that in a group of people all individuals have different birthday dates. In our setting, the number of possible birthdays was the number of peers ($n = 100$) and we found that a peer could send up to 12 requests while keeping the probability of sending all requests to different peers above 0.5. Hence, we set the number of requests sent by peers in every epoch to 12.

We should recall that a peer is soft-rewarded when she is the first peer to receive a request directly from a manager in the dissemination phase, and when she is the last forwardee of a good update. Even peers who are not trusted by others (with a reputation below the interaction threshold) participate in the dissemination phase, so they can be rewarded. Similarly, they can be the last forwardee of a good update if they start the interaction in the "learning tit-for-tat" protocol. These good behaviors should be rewarded (because they help good updates reach their respective managers), but the reputation increase should be much smaller than the hard punishment (δ) they receive for bad behavior (computing bad updates or duplicating them). Also, the soft punishment is used in the PUNISH_BAD protocol against peers that unintentionally forward bad or duplicated updates to incentivize them to launch punishment against the malicious peers. However, this punishment should also be much lower than the hard punishment received when a peer behaves badly.

We tested different values of δ with soft punishment and rewarding fractions $\delta/10$ and $\delta/4$, respectively, and 10% of malicious peers performing the attacks described in the following sections. We let the network of $n = 100$ peers evolve during 100 epochs and we computed the ratio between the number of useful updates received by a malicious peer and the number received by an honest peer. The main goal of our protocol is for honest peers to receive more useful updates than malicious peers. In this way, honest behavior will be the rational option. The results are shown in Table 2.

Table 2: Ratio of useful updates received by a malicious peer vs an honest peer for different values of δ

delta	Evil	Duplicator	Whitewasher	Selfish
0,10	0,1849	0,2279	**0,1114**	0,2872
0,20	0,1774	0,2491	0,1324	0,2765
0,30	0,1722	0,2287	0,1520	0,2766
0,40	0,1810	0,2213	0,1773	0,2762
0,50	0,1849	0,2277	0,1816	0,2734
0,60	0,1799	0,2294	0,2049	0,2731
0,70	0,1748	0,2284	0,1996	0,2744
0,80	0,1684	0,2308	0,2047	0,2748
0,90	0,1896	0,2410	0,2095	0,2750
1,00	0,1669	0,2495	0,2047	0,2726

[3] `https://en.wikipedia.org/wiki/Birthday_problem`

Table 3: Ratio of useful updates received by malicious peers vs honest peers for different values of soft punishment and reward fractions of δ

Soft reward	Soft punishment	Evil	Selfish	Duplicator	Whitewasher	Avg
1.00	1,00	0,657	0,726	0,233	0,268	0,471
	0.50	0,736	0,765	0,206	0,267	0,493
	0.25	0,760	0,756	0,198	0,265	0,495
	0.10	0,813	0,779	0,191	0,267	0,513
0.5	1,00	0,466	0,384	0,467	0,288	0,401
	0.50	0,380	0,345	0,169	0,277	0,293
	0.25	0,392	0,345	0,133	0,275	0,286
	0.10	0,414	0,374	0,120	0,271	0,295
0.25	1,00	0,997	0,286	0,971	0,339	0,648
	0.50	0,371	0,262	0,536	0,301	0,368
	0.25	0,224	0,253	0,172	0,290	0,235
	0.10	**0,180**	**0,239**	**0,116**	**0,287**	**0,205**
0.10	1,00	1,001	0,606	1,002	0,433	0,760
	0.50	0,997	0,316	0,989	0,400	0,675
	0.25	0,885	0,268	0,890	0,351	0,599
	0.10	0,231	0,224	0,226	0,320	0,250

We can see different values of δ had no effect on the ratio of good updates received by malicious peers but they did in the case of whitewashers. The latter peers always compute bad updates and whitewash as newcomers every 10 epochs (see details in Section 6.3.4). Indeed, we can see that the lower the value of δ, the lower was the ratio of good updates whitewashers received. With $\delta = 0.1$ a whitewasher got only 11.14% of the good updates received by an honest peer, while with $\delta = 1.0$ the ratio increased to 20.47%. The reason is that a larger δ causes the local reputations c_{ik} to change swiftly. Newcomers enter the system with the lowest possible local reputation (0) and they should earn reputation by forwarding requests or sending useful updates to another peers. When the value of δ is larger, whitewashers surpass the interaction threshold of other peers earlier, and they can start flooding the system with bad updates. Given these results, we set the value of δ to 0.1.

We next show that our previous choice of fractions $\delta/10$ and $\delta/4$ for soft punishment and rewarding, respectively, is a good one. To that end, we tested different fraction choices with $\delta = 0.1$ and 10% of malicious peers performing the attacks described in the next sections, and we computed again the ratio between the number of useful updates received by a malicious peer and the number received by an honest peer. The results are shown in Table 3. We can see that the best combination is indeed to set the soft reward to 25% of δ, and the soft punishment to 10% of δ. With this combination, all malicious peers obtain the lowest ratio of good updates.

Hence, in what follows we set the parameter δ used for hard punishment to 0.1 and the soft punishment and rewarding fractions to $\delta/10$ and $\delta/4$, respectively.

6.2. Baseline scenario

Ideally, in our protocol a peer should receive good updates if and only if that peer computes good updates for other peers. Thus, we first tested how the system evolved when all the peers in the network were honest: they adhered to

the protocol without deviating and they always computed good updates for the other peers.

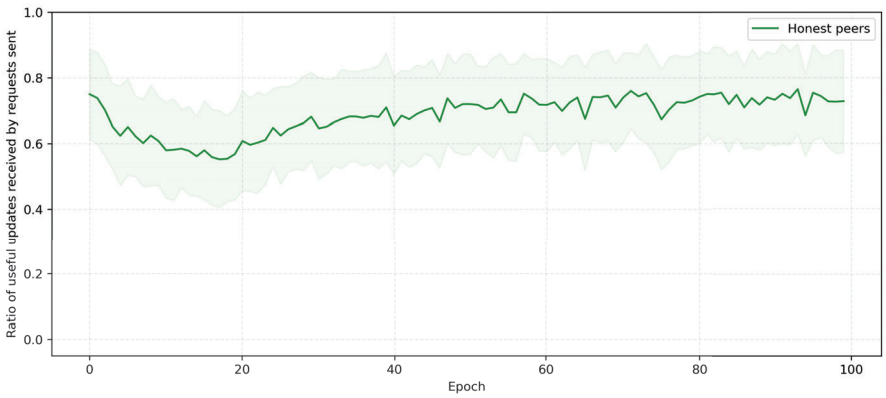


Figure 4: Baseline scenario. Ratio of useful (good and non-duplicate) updates received vs requests sent at each epoch.

Figure 4 shows the ratio between the number of useful updates (where an update is useful if it is good and non-duplicate) received and the number of requests sent by all the peers. The green line represents the average ratio over all peers for the epoch in the abscissae, while the shaded region represents the standard deviation of the ratio. In the first epochs, the behavior of the peers had not yet been fully captured and, for this reason, the misclassification of some updates caused a decrease in the ratio of useful updates received. When the system stabilized, the peers received around 72% of their requested updates, because some requests or updates got lost in different phases of the protocol execution.

These losses can be explained as follows. In spite of limiting to 12 per epoch the number of requests for updates sent by each model managing peer, around 6% of requests were dropped due to collisions at the dissemination stage. Then, some updates were also dropped because some model managers could not offer other updates in exchange for them during the learning tit-for-tat. Last, 3.8% of updates were wrongly classified as bad, which triggered the PUNISH_BAD subprotocol. As a result of this misclassification, some honest peers were punished and therefore their local reputations temporarily fell below the interaction threshold of other peers, which prevented these honest peers from receiving some updates.

6.3. Protection against attacks

We next considered attacks by malicious peers. We wanted to check to what extent honest peers did better than attackers in terms of received useful updates: if the former did better, then honest behavior is the rational option, as intended by the design of our protocol.

Specifically, we considered the following attacks by malicious peers: i) evil peers who follow the protocol but always compute bad updates; ii) selfish peers who try to free-ride on the other peers' computational effort; iii) duplicator peers who duplicate updates computed by other peers in order to save computational effort; and iv) whitewashers, who are peers with poor local reputations that choose to leave the network and re-enter it with a new identity as newcomers.

6.3.1. Evil peers

This kind of peers follow the protocol but always compute bad updates. Our protocol tries to thwart evil peers from finding other peers who compute good updates for them. Hence, if our protocol works well, evil peers should receive a lower ratio of good updates than honest peers.

To test this attack, we set up a scenario with the same parameters as the baseline scenario but with 10% of evil peers. The remaining 90% were honest peers.

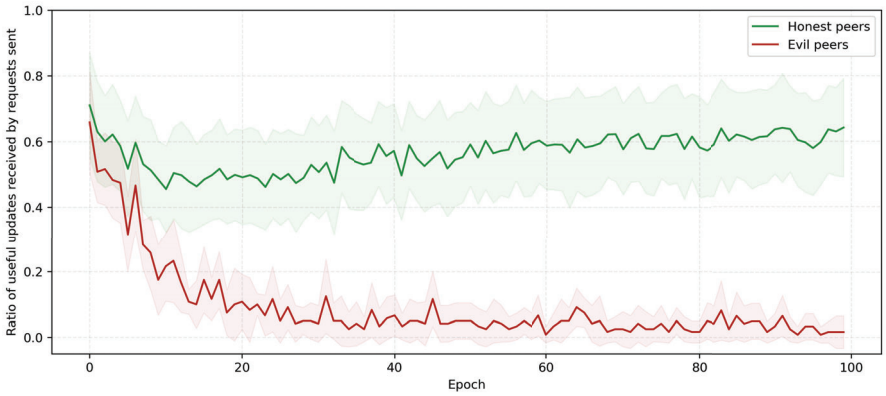


Figure 5: Scenario with evil peers. Ratio of useful updates received vs requests sent at each epoch.

Figure 5 displays the ratios between the number of useful updates received and the number of requests sent by honest and evil peers. As in the baseline scenario, during the first epochs when the behavior of the peers was being modeled, the ratio of useful updates received by the honest peers decreased. However, after this initial stage, this ratio increased constantly. After 100 epochs, honest peers were receiving around 64% of their requested updates, whereas evil peers received fewer than 3%. In the first epochs, evil peers were detected and hard-punished as soon as their updates reached the model managers. In this way, their local reputations eventually fell below the interaction threshold of the other peers and, at each subsequent epoch, evil peers found fewer and fewer other peers who were willing to interact with them.

In Figure 6 we show the behavior of our protocol when simulating a degraded outlier detection mechanism with a higher FNR of 10% (*i.e.*, 10% of good up-

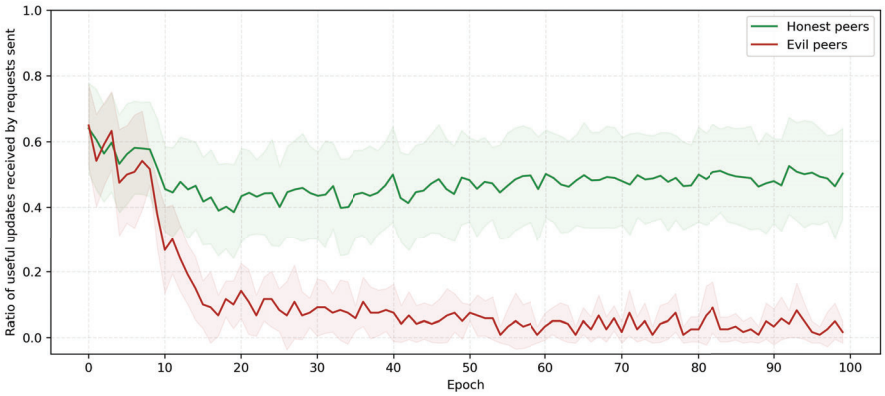


Figure 6: Scenario with evil peers. Ratio of useful updates received vs requests sent at each epoch when the effectiveness of the model manager's detection mechanism for bad updates is degraded to FNR=10% and FPR=8%.

dates are classified as bad) and a higher FPR of 8% (*i.e.*, 8% of bad updates are classified as good). While these figures mean that the outlier detection procedure now misclassifies updates nearly three times more often than the procedure used in previous experiments (with FNR=3.8% and FPR=2.1%), our protocol is not significantly affected, and it is still able to differentiate between good and evil peers. Despite this poorer detection, evil peers receive only around 5% of useful updates out of their total sent requests. These results demonstrate that, even if the model manager uses an ineffective detection mechanism, malicious peers are still substantially penalized, and thus they have a rational motivation to stop their behavior.

Figure 7 shows the ratio between the number of useful updates received and the number of requests made as a function of the proportion of evil peers, after 100 iterations. Honest peers obtained a higher ratio of useful updates than evil peers when the proportion of evil peers remained below 35%. However, this ratio decreased as the proportion of evil peers increased, due to both the lower number of trusted peers and the higher total number of bad updates computed. Fewer trusted peers imply fewer peers to exchange updates with, and fewer good updates that can be sent to their corresponding model managers. Evil peers kept a low ratio of useful updates when their proportion was below 30%. Nevertheless, at a proportion of around 40%, the computation of the interaction thresholds T_i was not able to capture the peers' behavior correctly, and all peers began to interact with each other and received the same ratio of useful updates. As the proportion of evil peers increased, fewer good updates were generated and the ratio of useful updates dropped to almost zero. Nonetheless, proportions of evil peers higher than 30% can be regarded as unrealistic.

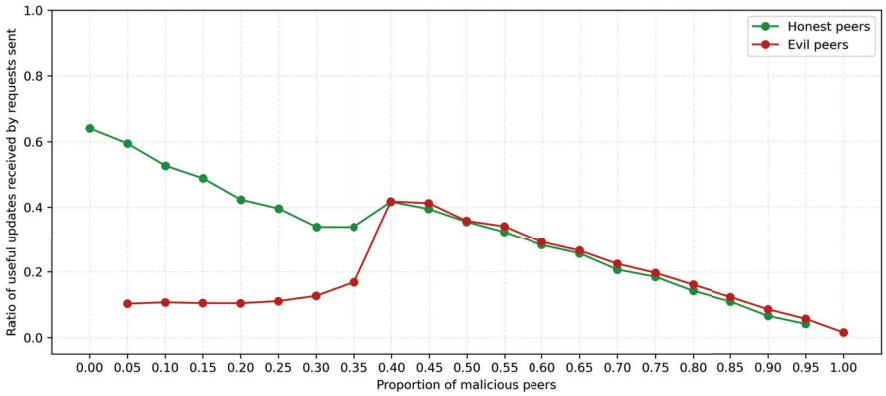


Figure 7: Scenario with evil peers. Ratio of useful updates received vs requests during 100 epochs, as a function of the proportion of evil peers.

6.3.2. Selfish peers

This type of peers disseminate their model and request new updates, but they ignore all the requests from other peers except one. In this way, they try to obtain new good updates while saving most of their own computing resources.

Note that if selfish peers did not compute any update, they would not be able to participate in the tit-for-tats.

We tested this attack in a scenario with the same parameters as the baseline scenario but with 10% of selfish peers. The remaining 90% were honest peers.

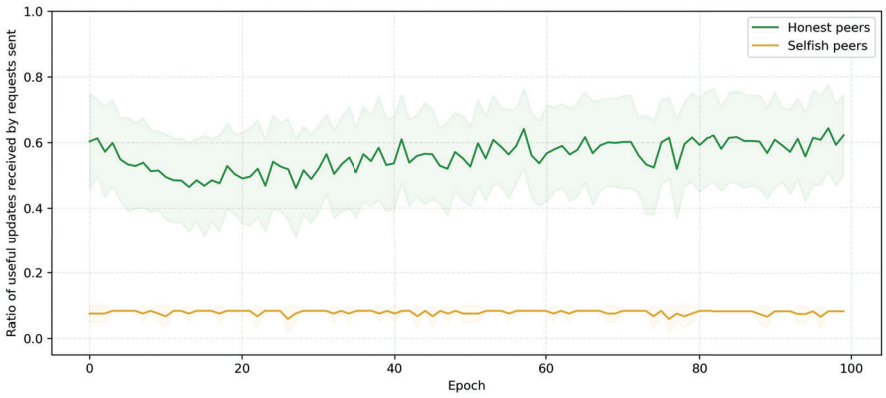


Figure 8: Scenario with selfish peers. Ratio of useful updates received vs requests sent at each epoch.

Figure 8 shows the ratios between the number of useful updates received and

the number of requests sent by honest and selfish peers. Honest peers ended up receiving around 60% of their requested updates, whereas selfish peers received at most 8%. Indeed, since selfish peers computed only one update, they only had one update to trade with the other peers in the learning tit-for-tat; hence, they received at most one new good update per epoch.

Additionally, Figure 9 plots the number of updates computed by selfish and honest peers against the number of updates they receive from other peers. The bottom-left points correspond to the 10 selfish peers who compute and receive 100 updates each, one per epoch, which explains why they appear as a single (overprinted) point. On the other hand, the top-right points correspond to the 90 honest peers. The correlation between the number of computed and received updates is really high (0.983), as anticipated in Section 3 above.

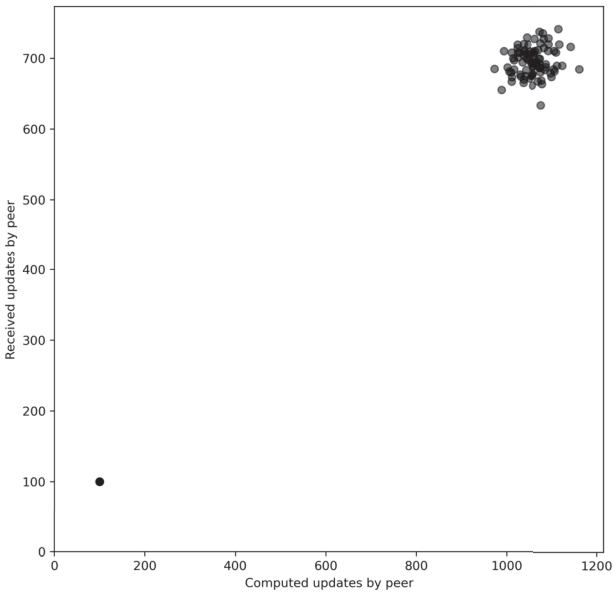


Figure 9: Scenario with selfish peers. Computed updates against received updates for selfish and honest peers.

Figure 10 shows the ratio between the number of useful updates received and the number of requests made as a function of the proportion of selfish peers, after 100 iterations. Regardless of their proportion, since selfish peers computed only one update per epoch, they received at most one new good update (8% of the 12 update requests they sent per epoch). Meanwhile, honest peers received more useful updates than selfish peers for almost any proportion of selfish peers, but this ratio decreased as the proportion of selfish peers increased, due to the lower overall number of updates computed.

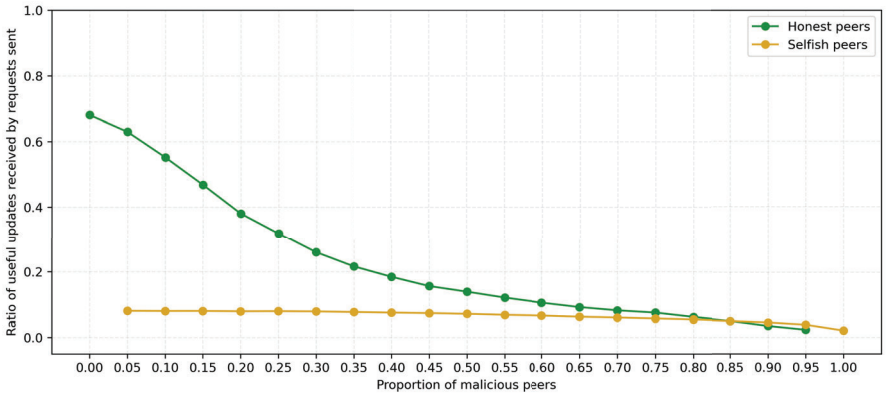


Figure 10: Scenario with selfish peers. Ratio of useful updates received vs requests during 100 epochs, as a function of the proportion of selfish peers.

6.3.3. Duplicator peers

An alternative option for a peer to save her own computing resources is to become a duplicator rather than a selfish peer.

Instead of computing updates themselves, duplicator peers just duplicate some of the updates they receive from other peers. By doing so, the duplicator effortlessly obtains several updates to trade with the other peers in the learning tit-for-tat.

To test this scenario, we used again the same parameters as in the baseline scenario but with 10% of duplicators. These peers duplicated every update they sent in the privacy tit-for-tat, thereby amassing updates to be traded later.

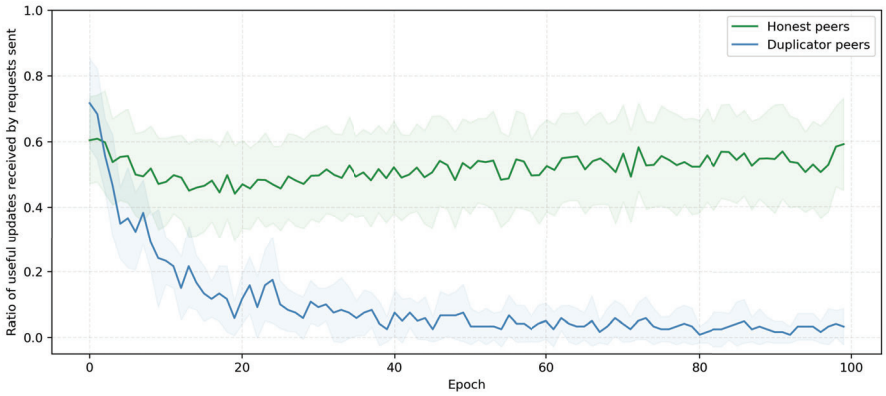


Figure 11: Scenario with duplicators. Ratio of useful updates received vs requests sent at each epoch.

Figure 11 shows the ratios between the number of useful updates received and the number of requests sent by honest and duplicator peers. While honest peers received almost 57% of their requested updates, duplicators only received around 5%. As soon as the model managing peers received a duplicated update, they launched the PUNISH_DUPLICATE subprotocol, identified the nearest duplicator and hard-punished her by decreasing her local reputation, thereby avoiding new interactions with that peer as soon as her reputation became lower than the interaction threshold. As duplicators kept duplicating updates, their local reputations never recovered.

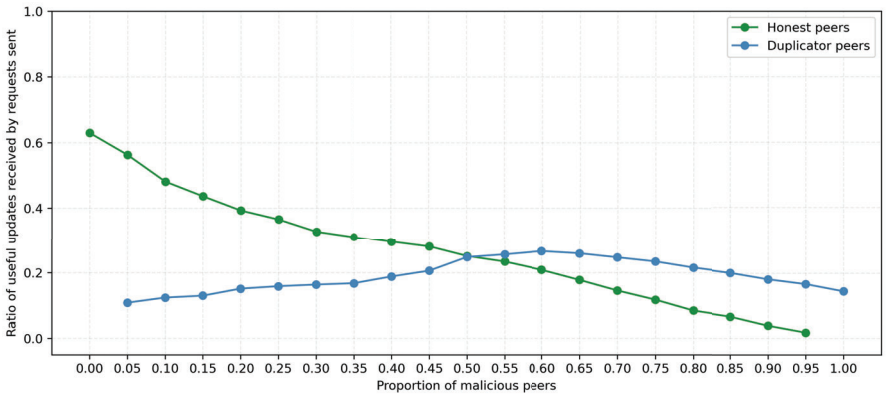


Figure 12: Scenario with duplicators peers. Ratio of useful updates received vs requests during 100 epochs, as a function of the proportion of duplicators peers.

Figure 12 shows the ratio between the number of useful updates received and the number of requests made as a function of the proportion of duplicator peers, after 100 iterations. Honest peers obtained a higher ratio of useful updates than duplicators when the proportion of duplicators remained below 45%. However, this ratio decreased as the proportion of duplicator peers increased, due to the lower number of trusted peers on the one hand, and to the higher total number of duplicated updates on the other hand. At 50% of duplicators, all peers received the same ratio of useful updates, and above this value, duplicators obtained a higher ratio than honest peers. At this point, the interaction threshold failed to capture the peer's behavior and all peers began to interact with each other. Since duplicators produced more updates (by duplicating some of them) than honest peers, they were able to exchange them and received more updates for their own model. Nevertheless, duplicators never received more than 30% of the expected updates, a value far from the 64% received by the honest peers in the absence of duplicators.

6.3.4. Whitewashers

Evil peers and duplicators are hard-punished by several peers at each epoch. After a few epochs, as their local reputations have decreased to the point of

being smaller than the interaction thresholds of most of the other peers, they receive a low percentage of the updates they request, as shown in the previous experiments. In these circumstances, evil peers and duplicators may be tempted to whitewash themselves by leaving the network and returning as newcomers. However, by design a newcomer enters the system with the lowest possible local reputation (zero), which is below the interaction threshold of almost every other peer.

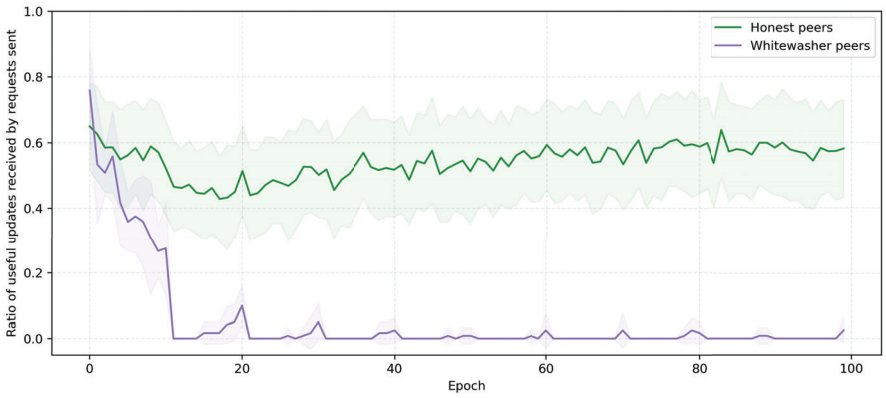


Figure 13: Scenario with whitewashers. Ratio of useful updates received vs requests sent at each epoch.

We tested this scenario by using the same parameters as in the baseline scenario but with 10% of whitewashers. These peers always computed bad updates and whitewashed as newcomers every 10 epochs.

Figure 13 depicts the ratios between the number of good updates received and the number of requests sent by honest peers and whitewashers. Honest peers eventually received around 60% of their requested updates, whereas whitewashers ended up receiving no useful updates. Although the whitewashers had the option of improving their local reputation by forwarding requests or updates to other peers, they kept computing bad updates and thus they were hard-punished by the model managing peers who received these bad updates. Therefore, in general they never managed to be trusted by the other peers; they only could occasionally get good updates from peers they had not yet interacted with (which explains the peaks).

Figure 14 shows the ratio between the number of useful updates received and the number of requests made as a function of the proportion of whitewashers, after 100 iterations. The results are very similar to those we obtained with evil peers (see Figure 7). Again, when the proportion of whitewashers was below 35%, honest peers obtained a higher ratio of good updates than malicious peers. However, from around 40% upwards, the computation of the interaction thresholds T_i was not able to capture the peers' behavior correctly (most of the

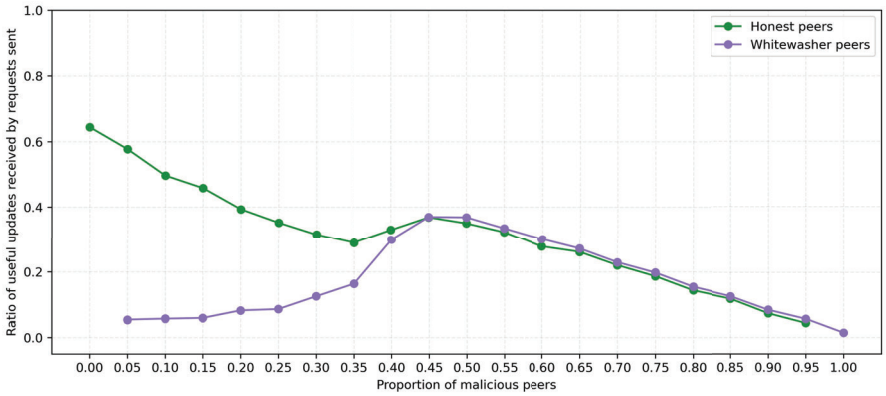


Figure 14: Scenario with evil peers. Ratio of useful updates received vs requests during 100 epochs, as a function of the proportion of whitewasher peers

interaction thresholds dropped close to zero, the initial reputation of newcomers), and all peers started to interact with each other and received the same ratio of good updates.

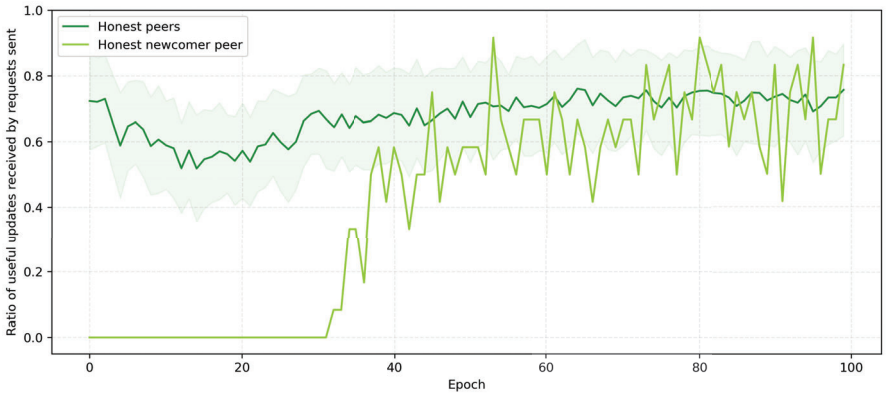


Figure 15: Scenario with an honest newcomer. Ratio of useful updates received vs requests sent at each epoch.

We finally considered the case of an honest newcomer, who computes good updates, to illustrate that she does much better than whitewashers. We had this newcomer join the network right after epoch 20. Figure 15 shows that the honest newcomer increased her ratio of received useful updates as a consequence of her participation in the protocol, by computing and forwarding new updates and requests for the rest of the peers. As soon as the newcomer forwarded a request

from another peer in the dissemination protocol or sent a useful update to another peer in the learning tit-for-tat, the beneficiary peers rewarded her with an increase of her local reputation. Although in the first epochs after entering the system these small increases were not enough to surpass the interaction threshold of other peers, eventually they were and the newcomer became trusted to more and more peers, who sent updates to her.

6.4. Overhead incurred by the protocol

Decentralized global reputation can be used as an alternative to tit-for-tat and local reputations to build artificial incentives that yield co-utile protocols. However, as noted in Section 2, global reputation management requires a significant number of messages to be exchanged between the peers of the network to compute the global peer reputations.

In order to compare the overhead of decentralized global reputation management with that of our tit-for-tat protocol, we estimated the number of messages needed by an adaptation of the global reputation protocol of [7] to FDML. In this case, every peer expecting a reward after computing or forwarding an update needs to send at least one message to each of her accountability managers. Then, after all peers have received the requested updates, global reputations are computed. The reputation values maintained by the accountability managers are normalized into the range $[0, 1]$ by dividing them by the largest reputation in the network. To that end, it is necessary that accountability managers broadcast all reputation values greater than 1. Finally, a decentralized global reputation computation must be run, which entails a new exchange of messages among the accountability managers.

In contrast, the tit-for-tat based on local reputations does not require additional communication among peers because the local opinion of a peer on another peer is never shared. However, as some good updates are lost in the learning tit-for-tat when a model manager does not have more updates to exchange, we counted as overhead the messages generated to forward these lost updates.

Figure 16 shows the overhead incurred by our tit-for-tat protocol and three versions of the protocol based on decentralized global reputation in a network where all peers are honest. The first global reputation-based protocol rewards with a reputation increase only the peer that generates a good update; the second rewards the generator of a good update and the first forwarder of that good update; finally, the third rewards the generator of a good update and all the forwarders along the path from the generator to the model manager. Every peer disseminated her model among 12 other peers and the number of epochs was 100.

The size of the network did not affect the overhead ratio incurred by any of the four protocols in the comparison. The tit-for-tat protocol added only a modest overhead in comparison with the global reputation-based protocols. Moreover, the greater the number of peers to be rewarded for each good update, the greater the overhead of global reputation-based protocols.

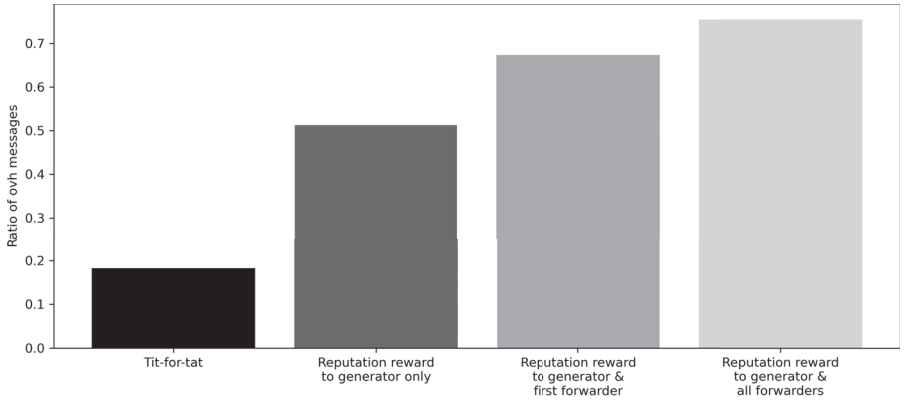


Figure 16: Overhead of the tit-for-tat protocol and three protocols based on decentralized global reputation

7. Conclusions and future work

We have presented a co-utile protocol for FDML. The fact that all participating peers are at the same time model managers and workers facilitates using tit-for-tat and local reputations to achieve co-utility. In this way, co-utility occurs naturally and complex incentives such as decentralized global reputations are not needed.

Our proposed protocol consists of a double tit-for-tat. The first tit-for-tat is used to hide the identity of the peer that computed each update, in order to protect the privacy of peers. In the second tit-for-tat, peers receive updates relevant to their own model, but they receive them from other peers that did not compute those updates. Deviations from the protocol are punished, which increases ostracism for the offending peers. On the one hand, a peer that fails to reciprocate in a tit-for-tat is punished by the non-reciprocated peer. On the other hand, after the second tit-for-tat, bad or duplicate updates can be detected by the model-managing peers, who launch punishment subprotocols aimed as tracing the offending peers. This ensures correctness of the learning process and incentivizes good and non-duplicate updates.

Future research will be devoted to fine-tuning the protocol in order to further reduce its overhead and the proportion of lost good updates. Also, we plan to explore adaptations of the proposed protocol to decentralized computations different from machine learning, such as decentralized anonymization or even multiparty computation among rational peers.

Acknowledgments and disclaimer

Partial support to this work has been received from the European Union's Horizon 2020 research and innovation programme under grant agreement No

871042 ("SoBigData++"), the MCIN / AEI / 10.13039 / 501100011033 / FEDER, UE (project PID2021-123637NB-I00, "CURLING") and the Government of Catalonia (ICREA Acadèmia Prizes to J. Domingo-Ferrer and D. Sánchez and grant 2021SGR-00115). The authors are with the UNESCO Chair in Data Privacy, but the views in this paper are their own and are not necessarily shared by UNESCO.

References

[1] A. N. Bhagoji, S. Chakraborty, P. Mittal and S. Calo. 2019. Analyzing federated learning through an adversarial lens. arXiv 1811.12470v3.

[2] P. Blanchard, E.-M. El-Mhambi, R. Guerraoui and J. Stainer. 2017. Machine learning with adversaries: Byzantine tolerant gradient descent. In: *Advances in Neural Information Processing Systems* 30. NIPS Foundation.

[3] A. Blanco-Justicia, J. Domingo-Ferrer, S. Martínez, D. Sánchez, A. Flanagan and K. E. Tan. 2021. Achieving security and privacy in federated learning systems: survey, research challenges and future directions. *Engineering Applications of Artificial Intelligence* 106: 104468.

[4] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H.B. McMahan, S. Patel, D. Ramage, A. Segal and K. Seth. 2017. Practical secure aggregation for privacy-preserving machine learning. In: *Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security – CCS 2017*, pp. 1175-1191. ACM.

[5] Y. Chen, L. Su and J. Xu. 2017. Distributed statistical machine learning in adversarial settings: Byzantine gradient descent. *Proceedings of the 2017 ACM on Measurement and Analysis of Computing Systems* 1(2):1-25.

[6] I. Colin, A. Bellet, J. Salmon and S. Clémençon. 2016. Gossip dual averaging for decentralized optimization of pairwise functions. In *Proc. of the 33rd International Conference on Machine Learning – ICML 2016*, pp. 1388-1396.

[7] J. Domingo-Ferrer, A. Blanco-Justicia, J. Manjón and D. Sánchez. 2022. Secure and privacy-preserving federated learning via co-utility. *IEEE Internet of Things Journal* 9(5):3988-4000.

[8] J. Domingo-Ferrer, O. Farràs, S. Martínez, D. Sánchez and J. Soria-Comas. 2016. Self-enforcing protocols via co-utile reputation management. *Information Sciences* 367-368:159-175.

[9] J. Domingo-Ferrer, S. Martínez, D. Sánchez and J. Soria-Comas. 2017. Co-utility: self-enforcing protocols for the mutual benefit of participants. *Engineering Applications of Artificial Intelligence* 59:148-158.

[10] M. J. Freedman, K. Nissim and B. Pinkas. 2004. Efficient private matching and set intersection. In *Advances in Cryptology-EUROCRYPT 2004*, pp. 1-19. Springer.

[11] P. Kairouz, H. B. McMahan *et al.* 2019. Advances and open problems in federated learning. arXiv:1912.04977.

[12] S. D. Kamvar, M. T. Schlosser and H. Garcia-Molina. 2003. The EigenTrust algorithm for reputation management in P2P networks. In: *12th International Conference on World Wide Web*, pp. 640-651. ACM.

[13] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins and Y. Yerushalmi. 1999. Web caching with consistent hashing. *Computer Networks* 31(11-16):1203-1213.

[14] A. Koloskova, S. Stich and M. Jaggi. 2019. Decentralized stochastic optimization and gossip algorithms with compressed communication. In *Proc. of the 36rd International Conference on Machine Learning – ICML 2019*, pp. 3478-3487.

[15] X. Lian, C. Zhang, H. Zhang, C.-J. Hsieh, W. Zhang and J. Liu. 2017. Can decentralized algorithms outperform centralized algorithms? A case study for decentralized parallel stochastic gradient descent. In *Advances in Neural Information Processing Systems – NIPS 2017*, pp. 5330-5340. NIPS Foundation.

[16] X. Liang, A. M. Javid, M. Skoglund and S. Chatterjee. 2020. Asynchronous decentralized learning of a neural network. In *IEEE Intl. Conf. on Acoustics, Speech and Signal Processing – ICASSP 2020*, pp. 3947-3951. IEEE.

[17] H. B. McMahan, E. Moore, D. Ramage, S. Hampson and B. Agüera. 2017. Communication-efficient learning of deep networks from decentralized data. In *Proc. of the 20th Intl. Conf. on Artificial Intelligence and Statistics – AISTATS'2017*, pp. 1273-1282.

[18] L. Melis, C. Song, E. De Cristofaro and V. Shmatikov. 2019. Exploiting unintended feature leakage in collaborative learning. In *Proc. of the IEEE Symposium on Security and Privacy*, pp. 691-706. IEEE.

[19] H. K. Mousavi, M. Nazari, M. Takáč and N. Motee. 2019. Multi-agent image classification via reinforcement learning. arXiv:1905.04835.

[20] G. Nadiradze, A. Sabour, D. Alistarh, A. Sharma, I. Markov and V. Aksenov. 2020. SwarmSGD: scalable decentralized SGD with local updates. arXiv:1910.12308.

[21] R. Shokri and V. Shmatikov. 2015. Privacy-preserving deep learning. In *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security – CCS 2015*, pp. 1310-1321. ACM.

[22] H. Tang, X. Lian, M. Yan, C. Zhang and J. Liu. 2018. D^2: Decentralized training over decentralized data. In *Proc. of the 35th Intl. Conf. on Machine Learning – ICML 2018*, pp. 4848-4856.

[23] D.G. Thaler and C.V. Ravishankar. 1998. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking* 6(1):1-14.

[24] P. Vepakomma, O. Gupta, T. Swedish, and R. Raskar. 2018. Split learning for health: distributed deep learning without sharing raw patient data. arXiv: 1812.00564v1.

[25] A. C. Yao. 1982. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science - SFCS 1982*, pp. 160-164. IEEE.

[26] D. Yin, Y. Chen, R. Kannan and P. Bartlett. 2018. Byzantine-robust distributed learning: towards optimal statistical rates. In *Proc. of the 35th Intl. Conf. on Machine Learning – ICML 2018*, pp. 5650-5659.