**To address issue 1**, we use the following ideas. We let a client deposit its coins to the contract as soon as it starts using the service, e.g. in the case of PoR when it uploaded its data and meta data to the server. The mount of deposit includes the service payment ‘’e’’, some extra coins for penalty ‘’p\_1’’ and to cover a dispute resolution ‘’p\_2’’. Also, the sever deposits some coins for penalty ‘’q\_1’’ and to cover a dispute resolution ‘’q\_2’’. In the payment time, after the client receives a proof from the server, it locally verifies the result. If it accepts, for a fixed period, it has a chance to call pay function that sends client’s ‘’e’’ coins plus the server coins to the server (in total e+q\_1+q\_2). The pay function returns the client’s coins for the penalty and dispute resolution back to the client. If the client does not call the pay function, after the fixed time, the server can call the function that would do the same. However, if the client rejects the proof, it raises a dispute and invokes a trusted arbiter who verifies the proof again. It sends its verification result to the contract who transfer coins depending on the arbiter’s message; namely, if it is accepted, all coins go to the server except for client’s p\_2 coins given to the arbiter; otherwise, all goes to the client, except for server’s q\_2 coins given to the arbiter. The reason why the above approach solves problem 1, is that the client pay deposit when it starts using the service, and if acts nothing at the payment time, it would lose its deposit. Also, if it lies about the verification result, it will be double checked by another party; therefore, it will be detected (and penalised), but the server still receives the payment. As evident, if the client and server act rationally, then there will be no need to involve the arbiter. Also, the contract side computation cost is very low. Later, we will show, in general cases, we can further reduce the role of the arbiter (even if a dispute is raised) and in a certain case, i.e. PoR setting, its role can be efficiently played by a smart contract.

**To address issue 2**, we use the following ideas. Instead of trying to hide the information from the public \emph{forever}, we let it become \emph{stale}, to lose its sensitivity, and then it will become publicly accessible. In particular, the client and server agree on the period in which the data should remain hidden, ‘’private time bubble’’. This requires, the parties to postpone calling the above pay function, or raising any dispute to the time when the private time bubble ends (or the bubble bursts). However, the client can still find out if the proof is valid as soon as the server provides it, because the verification is locally performed. To further hide the amount of deposit, we let each party to mask their coins. But this raises another challenge; namely, how can the (mutually untrusted) parties claim back their masking coins after the bubble bursts, while hiding its amount from the public in the private time bubble?

Note that it would not work if they explicitly encode in the contract the amount of masking, as it would reveal the masking coins amount to the public in the begging of the protocol. To address the challenge, we let the client and server, mutually untrusted, to agree on a statement specifying the deposit details (e.g. parties’ coins amount for the service, penalty, or masking). Later, when they call pay function to claim their coins, they also provide the statement to the contract which first efficiently checks the validity of the statement and if it is accepted, it distributes coins according to the statement (and status of the contract).

Therefore, for each client-server interactions there will be a \emph{private time bubble}, in which all information will remain hidden. After a certain time, the bubble is burst, and the information is public. However, at this point the information has lost their value, e.g. in the case of unavailability of PoR the server has patched the vulnerability, or the client has stopped using the service long time ago.