Confirm Activity of Service-Providers in the NuCypher Network

Ghada Almashaqbeh

NuCypher
ghada@nucypher.com

Abstract. In this document we study the confirm activity problem in the NuCypher network. This problem is concerned with devising secure techniques that allow a service-provider, or Ursula, to prove being online and responsive during any time period. We consider two flavors of the problem based on the network operation stage, namely, confirm availability and confirm service activity. The former is about confirming that Ursula is available all the time and replies to all Bob requests, while the latter is about proving that an Ursula has served a specific number of distinct re-encryption requests within a given period. We present several potential solutions for each of these types, discuss the characteristics of these solutions with respect to their security guarantees and efficiency, and provide recommendations to when each of them should be used.

1 Introduction

The NuCypher network [1,2] is a decentralized key management system, encryption, and access control service. It uses threshold proxy re-encryption, namely, the Umbral scheme [3], to delegate access to encrypted data through a public network. This network is composed of semi-trusted re-encryption service-providers, or Ursulas, that implement access control policies created by data owners, or Alices. Alice supplies each Ursula with re-encryption keys allowing a quorum of Ursulas to transform ciphertexts encrypted under Alice's public key into ciphertexts encrypted under the delegatee's, or Bob's, public key. This enables the latter to decrypt the ciphertext without revealing anything about the decryption keys or the raw data to the intermediate service-providers.

Joining the NuCypher network is governed by consensus rules defining the service setup and the monetary incentives paid to provide the service. In order to join the system, Ursula needs to stake an amount of NU tokens (the native token of the network) for a specific period which will be released when the pre-specified staking period is over. Alice chooses an Ursula to hold an access control policy with a probability proportional to the stake this Ursula pledged in the system. Therefore, the stake value influences the service load an Ursula may receive. This stake is also used to punish Ursula financially, by forfeiting part of it for detectable malfeasance.

Alice uses the network by creating an access control policy to be implemented by a set of Ursulas. This policy specifies the re-encryption key fragments each

Ursula will need to answer requests coming from Bob(s), as well as the duration of the policy, which is basically the timeframe during which Bob is authorized to access Alice's data. Furthermore, the policy contract requires Alice to lock an amount of Ether that will be used by the network to pay Ursulas for providing the re-encryption service. As noted, Alice is not exposed to the NU token. All that she needs is an Ethereum wallet, awareness of the NuCypher network architecture to select Ursulas, and knowledge of the NuCypher rules with respect to preparing access control policies.

Bob, on the other hand, does not deal with currency in the system at all. He interacts with the storage network to retrieve the encrypted data, and with Ursulas in the NuCypher network to request the re-encryption service after which he will be able to access the raw data of interest.

Rewards and activity monitoring in the NuCypher network. Ursulas are rewarded for their re-encryption service by using two sources; the first is the service fees collected from the policy owner Alice, while the second is a subsidy in the form of newly minted NU tokens distributed on a schedule enforced by the NuCypher protocol. The latter will be provided during the early stage of the network operation to encourage Ursula's participation and sustain their operations in lieu of a mature fee market.

Currently, the subsidy size in each round or period is computed for each Ursula based on the size of her stake and duration of commitment to the network, and provided that she confirms to be online during this round. Confirming the status of being online is done by calling a simple function in one of the NuCypher contracts, which is like a signal that the calling Ursula is online. Similarly, for service fees, Ursula collects part of the Ether locked by Alice during the policy's duration in proportion to the number of periods up to that point in which Ursula has confirmed her online status.

Motivation. Although rewards computation and activity monitoring are reasonable for the first iteration of the network, we acknowledge that it suffers from two main problems:

- Confirming being online is flawed in the sense that the function call that Ursula makes does not require any input or proofs on providing the service, or even being online and responsive. Ursula can stay offline the majority of the time and not respond to Bobs' requests, then come online merely to call this function. This allows Ursula to collect both subsidy and fees without doing all (or even most) of the necessary work. Even worse, Ursula will be able to get her stake back once it unlocks.
- The revenue computation (both fees and subsidy) is agnostic to the number of requests Ursula serves. Thus, an active Ursula who may serve a huge number of re-encryption requests, and one who does not receive any requests (or more importantly, a malicious/lazy Ursula that does not answer Bob's requests), will both collect the same amount of revenue if they hold identical policies.

Nonetheless, Ursula has an economic incentive to log her availability and provide a satisfactory level of service. If she "cheats" she will risk devaluing her stake since the network will be less useful to users who will leave and cease paying fees. However, we recognize the long-term need for a more intelligent and specialized technique to hold Ursula accountable and prevent her from cheating the system. Then we need to adapt the subsidy and fee computation accordingly to factor in the amount of delivered service, and potentially punish Ursulas if it is insufficient.

Problem statement. The confirm activity problem is defined differently based on the network operation stage. In the first 5 to 8 years after launch, when subsidy rewards are still distributed, we are concerned with the availability of Ursulas and their willingness to serve all requests coming from Bob. In other words, regardless of the number of requests served, the revenue total value (both subsidy and fees) will be computed based on the length of time during which Ursula is online and well-behaved.

On the other hand, in later stages, when subsidies disappear and the only revenue comes from fees, confirm activity will be tied to providing the re-encryption service, in the sense that the payment will be partially computed based on the amount of service provided. This requires Ursula to confirm its activity by proving that she served a given number of distinct re-encryption requests during a given period.

To make the distinction clear, we refer to the first as *confirm availability*, and for the second as *confirm service activity*.

Contributions. In this document, we introduce several solutions for both forms of confirming activity. These solutions differ in the trust, security guarantees, interactivity, and efficiency.

For confirming availability, we employ a detect-and-punish approach embedded within an optimistic service monitoring process. This means that it is only invoked when Bob complains that an Ursula is unresponsive. The proposed solution labels Ursulas as working and gateway Ursulas. The former are contacted in a regular way to handle the re-encryption service, while the latter act as challengers that are contacted only when there is a complaint from Bob so they will attest whether a working Ursula is indeed unavailable. If that is the case, a quorum of gateways (who have challenged the working Ursula and found her unresponsive) will collectively sign a proof-of-unavailability that will cost the misbehaving working Ursula part of her stake. We discuss several aspects related to gateway selection and a variety of potential attacks on this approach along with mitigation techniques.

For confirming service activity, we devise three solutions that allow producing a proof attesting to the statement *"An Ursula has served ω distinct requests correctly"* for some positive integer ω representing the number of served requests. These solutions are refereed to as zero knowledge proof-based (ZKP-based), committee-attestation, and commit/challenge/open-based.

For the *ZKP-based* approach, we utilize the fact that in our network each Ursula already produces a ZKP attesting to the correctness of each re-encryption request she performs [3]. Our solution is to aggregate the proofs of all served requests within a round into one proof attesting to the above statement. This accumulation can be done by using generic techniques like proof carrying data (PCD) [4] or incrementally verifiable computation (IVC) [5]. An alternative, possibly more efficient approach, is to view the above statement as an instance of arithmetic circuit satisfiability problem and combine it a highly efficient ZKP system, e.g., [6]. That is, build a circuit that takes the set of correctness proofs and their count ω as inputs. If the number of correct distinct proofs equals to ω, i.e., the circuit output 1, then a valid proof will be produced. Ursula can submit this proof to the network to claim her payments.

As for the *committee-attestation* approach, it shares a similar theme to confirm availability solution mentioned previously. For each round, a committee of Ursulas (and possibly outsider verifiers) is elected to attest for the work load an Ursula served. In detail, the working Ursula sends all Bob requests she served along with the correctness proofs to this committee. After verifying all the proofs, the committee collectively signs the *"An Ursula has served ω distinct requests correctly"* statement, while replacing ω with the actual number of served requests. The committee (or the working Ursula) submits the signed statement to the network so the working Ursula can claim her payments.

For the *commit/challenge/open* approach, we introduce a mechanism inspired from the Merkle tree-based ZKP system proposed in [7]. For each round a working Ursula constructs a Merkle hash tree of all served requests and their correctness proofs. Then, she uses the root as a random beacon to be fed into an iterative hash process to select n leaf nodes at random to open them (i.e., these are the challenges). Ursula signs the Merkle tree root and the activity proof will be composed of the signed root, the challenge leaf nodes along with their openings, and membership paths of the challenge leaves. Ursula sends this proof to either a sidechain (maintained by round-selected committee) or to the main network to verify. Once verified, Ursula will be rewarded with the service payment.

Lastly, we discuss the trade-offs of each of these solutions in terms of security guarantees, assumptions, and efficiency. We also outline some recommendations regarding which solution should be used at what stage of the network operation.

Organization. We begin with outlining the adopted threat and network model, in Section 2, followed by an overview of some of the cryptographic primitives that we use in Section 3. Then, we present the proposed solutions for confirm availability in Section 4 and those for confirming service activity in Section 5. Lastly, we briefly discuss the trade-offs and requirements of each of these solutions along with some usage recommendations and future work directions in Section 6.

2 Network and Threat Models

This section describes the network and threat models adopted by the proposed solutions. In particular, it introduces a modification for the network model of NuCypher, and it describes the security assumptions and attacker collusion cases considered in this work.

2.1 Network Model

In order for the proposed solutions to work, we need to ensure freshness, integrity, and authenticity of the re-encryption requests (as well as service complaints as we will see later) issued by Bob. This can be achieved by requiring Bob to include a timestamp, or sequence number, in each request to ensure freshness, and to sign the request to ensure integrity and authenticity. The keypair used for the signature should be separate from the keypair Bob uses for proxy re-encryption in the system.

2.2 Threat Model

The goal of this document is to address accounting attacks, namely, holding Ursula accountable for the service it provides and for claiming being available and responsive. In addressing this type of attacks we adopt a threat model that makes the following set of assumptions:

- **Self-interested parties.** We do not place trust in any party and we assume that all participants are self-interested. This means that a party may decide to follow the protocol or deviate from it, either on its own or by colluding with other attackers, and such decision is solely based on what maximizes the financial profits of this party.
- **Attackers' collusion.** If it is profitable, an Ursula may, for example, spin out her own Alice/Bob instances or collude with Alice or Bob. That is, by creating Alice/Bob instances (or colluding with them), Ursula will receive larger number of policies and re-encryption requests. Such a collusion case could be profitable if the subsidy size is influenced by the amount of service delivered. However, this is not the case in NuCypher as mentioned before. Thus, such a collusion scenario is not appealing to a rational Ursula.
  Moreover, we do not consider collusion between Bob and Ursula as a practical threat. This means that cases of Bob pretending that he obtained service from Ursula (while no service has been delivered) is not an issue. We do not suspect this will be of any importance and there is no motivation to do it given that the work Ursula does for any re-encryption request is minimal (as we will see shortly, the ciphertext size is very small since it encrypts a symmetric key instead of the whole data).
  We note that the above non-collusion assumption does not affect the solutions proposed to handle the confirm service activity (Section 5). This means that these solutions provide stronger security guarantees than the confirm

availability solution (Section 4), where they address the issue of potential collusion between Bob and Ursula. In order to make an educated decision of the plausibility of this threat, we need more data about the system operation and the behavior of the participants once subsidies stop in the system.

- **Honest Ursulas.** We also assume that when sampling a subset of Ursulas in the network, at least one of them is honest. This assumption can be achieved by having a global assumption regarding the lower bound of the number of honest Ursulas with respect to the total number of Ursulas in the NuCypher network. (Or it can be achieved by deploying a special entity like an external verifier, that changes on a periodic basis, or one by the NuCypher company, that is trusted to faithfully participate as a member of each samples set. This verifier does not provide any other services in the system.)
- **Efficient adversaries.** We deal with computationally-bounded (or probabilistic polynomial time) adversaries that cannot break secure cryptographic primitives with non-negligible probability.

We also work in the random oracle model where hash functions are modeled as random oracles.

3 Preliminaries

This section provides an overview of several concepts used in the proposed solutions. These include the framework of proof carrying data (PCD), and the collective signing (CoSi) protocol.

3.1 Proof Carrying Data (PCD)

The paradigm of PCD [4] allows proving the correctness of a distributed computation performed by a set of untrusted parties. PCD produces a single proof that attests not only to the correctness of the final result, but also to the correctness of the entire history of intermediate computations that produced this result. Correctness here is defined as a polynomially computable predicate that abstracts the properties, or invariants, that must be satisfied.

To clarify how PCD works, consider a distributed computation task that involves the set of parties shown in Figure 1. Party P_1 starts the computation with its own input, produces an intermediate output with a proof saying that it performed the computation as defined by the protocol. It then sends both the output and the proof to the next party, which is P_2 in this case. Here, P_2 will have its own input x_2, as well as everything it received from P_1 (including the proof) as inputs to the next step of the computation. Similarly, P_2 will produce another intermediate output and a proof. This proof not only attests to the correctness of the computation done by P_2, but also to the history that led to the result y_2. In other words, it implicitly includes, or accumulates, the proof from P_1. The same process continues until the full computation is finished. Anyone can verify

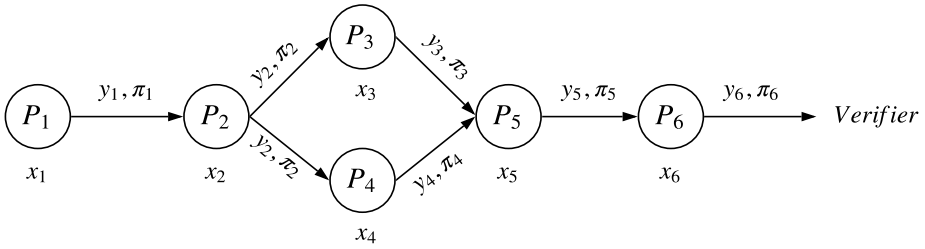

Fig. 1: An example of a PCD application. A distributed computation involving parties P_i, for $i \in \{1, \ldots, 6\}$, each of which has a local input x_i and produces an output y_i with a proof π_i.

the correctness or compliance of the whole distributed protocol by only verifying the last proof π_6 produced by the exit party, which is P_6 in the figure.

As shown, PCDs enable untrusted parties to work with each other in a fully distributed fashion, without overwhelming the system with the storage and verification of individual proofs for each step of the distributed computation, and without re-executing any of the intermediate computations.

3.2 Collective Signing (CoSi)

CoSi [8] is a protocol that enables a set of parties to cosign a statement together while producing a single signature. Thus, both the verification time and the space requirements are just like having a single signer. However, this signature holds all parties accountable for the statement they agreed to sign.

CoSi builds upon Schnorr multi-signatures [9–11], but combines them with communication trees to speed up the signing process, especially when having a large number of cosigners. In what follows, we only present the protocol with a plain communication architecture as it suffices for the confirm service activity solution introduced in this document.

Schnorr signatures work in a group $\mathbb{G}$ of a prime order q and generator g, such that the discrete log is believed to be hard in this group. Take n parties that want to sign a statement together. Each of these parties has a secret $sk_i \in \mathbb{Z}_q$ and a public key $pk_i \in \mathbb{G}$ such that $pk_i = g^{sk_i}$. To sign a message m, one of these parties, say P_1, coordinates the signing process as follows:

1. P_1 prepares a message m and sends it to the rest of the signers.
2. Each party P_i, possibly after verifying m, selects a secret $\tau_i \in \mathbb{Z}_q$ and computes $V_i = g^{\tau_i}$, then it sends V_i back to P_1.
3. P_1 aggregates all random values received from the signers, and its own, by computing $V = \Pi_{i=1}^{n} V_i$.
4. P_1 then computes $c = H(V||m)$, where H is an appropriate hash function and $||$ is a concatenation operator. P_1 then sends c to the rest of the parties.
5. Each party computes a response $r_i = \tau_i - sk_i \cdot c$ and sends it to P_1.

6. Lastly, P_1 computes $r = \sum_{i=1}^{n} r_i$, and outputs the collective signature over m as (c, r).

Verifying the signature proceeds as in classical Schnorr signatures [9] with one difference. An aggregated public key pk is used in the verification process, which is computed as $pk = \Pi_{i=1}^{n} pk_i$.

The work in [8] tackles several issues related to the availability of the signing parties, and optimizing the communication between them. We believe that such techniques can be used in the NuCypher network if we adopt the CoSi-based solution for the confirm service activity issue.

4 Confirm Availability Solution

In this section, we present a solution for confirming availability of Ursulas, which will be used during early operation stage of the network. The proposed solution is called *Optimistic Challenge-based Approach*. It is centered around two important concepts: requiring minimal changes to the current network implementation, and minimizing the additional overhead. We introduce the design details of our solution, after which we discuss some potential security attacks and how to address them.

Overview. The optimistic challenge-based approach is based on a detect-and-punish technique. As long as everything work as expected, no extra measures will be invoked. In other words, investigating the status of a specific Ursula is activated only when Bob does not hear back regarding a re-encryption request he has issued. Thus, Bob proceeds as usual in requesting the service. If any of the contacted Ursulas does not reply, Bob submits a complaint claiming that this Ursula is unavailable. This complaint triggers the challenge phase in the network to verify Bob's claim, and hence, punish Ursula financially (by slashing part of her stake) in case the claim found to be valid.

The proposed scheme labels Ursulas in the network under two classes: Working Ursulas, which is the conventional role of maintaining access policies and providing the re-encryption service. And gateway Ursulas, which is an additional role of handling Bob's complaints. Gateway Ursulas are a subset of the working Ursulas that is selected at random for each round.[1] Hence, each working Ursula will get to serve as a gateway from time to time.

Accordingly, Bob contacts working Ursulas listed in Alice's policy. If any of them does not reply, Bob reaches out to a gateway Ursula and forwards her the request. The gateway in turn mediates the service session by forwarding the request to the working Ursula again and waiting for a reply. If a reply is received, the gateway sends it to Bob and closes the case.

On the other hand, if no reply is received, the gateway notifies other gateways selected for the current round to repeat the mediation process. If none of them receive a reply, they collectively sign (using the CoSi protocol described

[1] A round is the time needed to mine a block or several blocks on the blockchain.

in Section 3.2) a statement against the working Ursula, which we call a proof-of-unavailability. This proof is submitted to the StakeEscrow contract, and after being verified, part of the working Ursula's stake will be slashed as a punishment.

Working Ursulas discovery. In the NuCypher network there is a separation between the role of a staker and a working Ursula. All stakers' Ethereum addresses, along with their stake values, are recorded in one of the NuCypher network smart contracts, namely, the StakeEscrow contract. Each of these stakers will be tied, or bonded, to a working Ursula to provide the re-encryption service by having its address listed next to the staker's address.

For each access policy, Alice delegates the re-encryption service to a designated set of working Ursulas. Each of these Ursulas will be supplied with a re-encryption key fragment that converts a ciphertext encrypted under Alice's public key into a ciphertext fragment encrypted under Bob's public key. Once Bob gets a sufficient number of ciphretext fragment, based on some threshold, he can combine these fragments together to get the complete ciphertext, which he can decrypt.

The StakeEscrow contract does not contain any information about how to reach a specific working Ursula. Thus, and as part of the access policy, Bob receives a map from Alice containing the set of working Ursulas that has the key fragments, along with their Ethereum addresses. To allow the parties to communicate with each other, each participant, i.e., Alice, Bob, Ursula, employs a discovery protocol (more like a gossiping protocol) to discover the IP addresses and ports of the working Ursulas in the network. Hence, Bob uses the map (in association with the network discovery protocol) to reach each of these Ursulas and obtain the service.

The discovery process also includes retrieving the keypair a working Ursula uses for the re-encryption service purposes. We call this keypair a stamp, and we require an Ursula to use her stamp when signing all messages related to the proposed confirm availability protocol.

Gateway Ursulas selection. For each round a set of n gateway Ursulas will be selected. A proof-of-unavailability will be accepted by the network if it is signed by at least t gateways among this set (this threshold is a relaxation of requiring all gateways to sign since it could be the case that not all of them are active).

A simple idea to select the gateways, similar to the deterministic lottery protocol found in [12], is to use a high entropy source of randomness to obtain a random beacon for each round. Then feed this beacon into an iterative hashing process. That is, let the random beacon of the i^{th} round be r_i, and let a hash function be denoted as $H(\cdot)$. To select the first gateway, the hash $h_1 = H(r_i)$ is computed, and then it is mapped to a working Ursula index as listed in the StakeEscrow contract. That is, the working Ursula bonded with the first staker listed in the contract has index 0, the next has index 1, etc. Then, to select the second gateway, hash the previous hash to obtain $h_2 = H(h_1)$ and map the output to a working Ursula index, and so no. In case of a collision, i.e., an

already selected index is selected again, discard it and proceed to the next hash iteration. This process continues until a set of distinct n indices is computed.

It should be noted that since the random beacon is public, although it will be known only when round i starts, any party can compute the set of selected indices. Thus, Bob performs the hash iterative process when needed to obtain the gateways. The selection can be also verified by repeating the hash iterative process by any other party.

Furthermore, note that the selection process may produce Ursulas for which Bob does not have a contact information. Hence, Bob has to discover these Ursulas before being able to submit a complaint. To reduce the delays, and as outlined earlier, Bob starts with complaining to one gateway and involves the rest if the working Ursula does not respond to the challenge from this gateway. Hence, Bob can start with a gateway that he already knows how to reach (if any) and in the meantime works on discovering the rest of the selected gateways (if he does not already have their contact information).

The above protocol assumes working in the random oracle model, meaning that hash functions are modeled as random oracles. Thus, the iterative hashing process is conjectured as producing a random, unbiased output. Although, we believe this is sufficient for our purposes, this can be replaced with more sophisticated approaches to produce random strings (that are then mapped to indices), e.g., a verifiable random function [13].

As for the high entropy source of randomness, there are several potential approaches to fulfill it. We can rely on an external source, e.g., the NIST random beacon [14], or even a combination of multiple sources. Or we can also resort to the random oracle model and assume that block hashes are drawn from a uniform distribution, and thus, use the hash of the block mined in the current round as a random beacon.[2] Alternatively, and to address concerns regarding the low entropy of block hashes, we can use randomness extractors to produce a high entropy beacon from the block hashes [15].

Proof-of-unavailability processing. As mentioned previously, a proof-of-unavailability against a working Ursula is a statement stating that this Ursula was unavailable during a specific round signed by at least t gateways out of the n gateways selected for that round. Such a proof is submitted to the StakeEscrow contract and is verified as follows:

- Reproduce the list of gateways selected for the current round, by using the same process described previously, to verify the signing gateways are legitimate.
- Compute the collective verification key of the cosigners and verify the collective signature over the signed statement (see Section 3.2).

[2] To handle inconsistencies of the blockchain view, the hash of the block that is confirmed at the current round can ebe used. This is the block that was mined y rounds ago where y is the confirmation interval in the underlying cryptocurrency system.

– If all checks pass, meaning that the proof-of-unavailability is valid, the StakeEscrow contract revokes part of the stake bonded to the working Ursula as a punishment.

Note that the above assumes that the StakeEscrow contract is aware of the gateways' stamps (or keys) that are used in signing the proof-of-unavailability. This can be done by either expanding the staker list in the contract to contain not only the Ethereum addresses of working Ursulas, but also their stamps. Another option is to piggyback the stamps (signed by using the key corresponding to the working Ursula's Ethereum address) with the proof-of-unavailability. A more simpler, and efficient, option is to sign the proof-of-unavailability by using the key corresponding to the working Ursula's Ethereum address in the first place, and thus, the StakeEscrow contract will not need an additional information. Which option to use depends merely on efficiency considerations.

Working only during the challenge phase. A potential attack on our scheme is that a working Ursula may operate only during the challenge phase. That is, she ignores any fresh request coming from Bob, and just replies afterwards when receiving the same request again (i.e., when it is being forwarded by the gateways during the challenge phase).

We argue that such an attack is not a practical issue since a rational Ursula will be online eitherway to monitor whether this is a challenge phase or not. Also, the amount of work needed to respond to a re-encryption request is minimal. Hence, there is no incentive at all to follow this strategy, meaning that a rational Ursula will act honestly.

DoS attacks. Another potential issue is exploiting the unavailability complaints to perform a DoS attack against the gateways and working Ursulas. In detail, Bob may issue fake complaints against working Ursulas, even though they are responsive. Alternatively, an attacker (insider or outsider) may replay these complaints, or issue them on behalf of Bob. The goal could be to slash Ursula's stake or to overwhelm the gateways with the challenge process and make them unavailable to perform the re-encryption service (recall that the gateways are originally working Ursulas in the NuCypher network).

This issue can be handled using a collection of techniques as follows:

– Require Bob to sign all the complaints (so no one can impersonate Bob unless they know his signing key), and include a timestamp or a sequence number in each of them (to ensure freshness and prevent replay attacks).
– Increase the cost of issuing complaints. That is, require Bob to do some work in order to produce a valid complaint (in a similar way to proof-of-work, i.e., produce a hash over the complaint with specific number of leading zeros).

Furthermore, since gateways Ursulas are selected fresh at random for each round, an Ursula will get the chance to be only a working Ursula during some rounds. This reduces the impact of gateway DoS attacks for any Ursula as the gateway role is rotating and not fixed.

Unresponsive gateways. There is a chance that the gateways themselves could be unresponsive. This will stall the challenge phase and make it infeasible to prove that some working Ursula is unavailable. We believe that such an issue will be organically handled by the network. That is, the larger the number of Ursulas in the network, the higher the chances of having more honest Ursulas. By increasing the number of selected gateways n, and given that the selection process is random, the chances of having at least t honest gateways in a round will be high.

Moreover, to encourage gateway participation, a potential approach is to incentivize Ursulas to do the work by paying them a small fee. Such a fee could be out of the revoked stake (if any) or paid by Alice from her access policy funds.

It could be the case that all gateways in a round are unresponsive (although we believe that the probability this happens is very low). In this case, it could be viable to perform the challenge phase in a recursive way. In other words, Bob can send an unavailability complaint against the gateways of the current round to the gateways of the previous round. The previous round gateways will in turn challenge the current round gateways by forwarding Bob's complaint to them and sign a proof-of-unavailability if no response is received.

Quantifying the financial punishment. A question that comes to mind is how much should be revoked from Ursula's stake once a proof-of-unavailability is approved? Quantifying this amount should be driven by two factors; the unavailability duration and the utility gain (or profits) of this cheating behavior in terms of the collected fees and subsides during that period.

We can compute this value to be equal to the fees and subsidies that an Ursula is supposed to receive in a round. Hence, the working Ursula will, first, lose her round payments from the network (she will not get paid by Alice or by the network), and second, lose the same amount from her stake. Such a loss will take place even if a single proof-of-unavailability is received against a working Ursula within a round. In fact, the StakeEscrow contract will process only one proof against a working Ursula per round to reduce computational costs in the system.

5 Confirm Service Activity Solutions

As discussed previously, for confirm service activity we need secure solutions that allow an Ursula to prove that she has served a given number of distinct re-encryption requests within a given period. In particular, for every round an Ursula will need to produce a proof attesting to the statement "Ursula has served ω distinct re-encryption requests correctly," for some positive integer ω. In this section, we present three solutions to achieve this goal, which we refer to as zero knowledge proof-based (ZKP-based), committee-attestation, and commit/challenge/open-based. These solutions differ in their security assumptions, guarantees, proof technique, and efficiency. In what follows, we provide the design details of each of them.

5.1 ZKP-based Scheme

In this solution, we utilize the fact that in the NuCypher network each Ursula already produces a ZKP attesting to the correctness of each re-encryption request she performs to be checked by Bob [3]. Such proofs can be aggregated into a single proof attesting to the statement outlined above. This accumulation can be done using two approaches: by using generic techniques like proof carrying data (PCD) [4] or incrementally verifiable computation (IVC) [5], or by using a ZKP system for arithmetic circuits.

For the generic PCD/IVC approach, we have only one prover or computation party, namely, Ursula. For each round, Ursula starts with input x_1, which is a signed and fresh re-encryption request from Bob. It answers this request with a ciphertext fragment $cFrag_{x_1}$ and produces a correctness proof π_1 as defined in the Umbral scheme [3]. Then, when the next request x_2 arrives, which is the input for the next step in the computation, Ursula answers this request as before and produces $cFrag_{x_2}$ and a correctness proof π_2, then it uses $(x_2, cFrag_{x_2}, \pi_2)$ and the previous proof π_1 to produce $\hat{\pi}_2$. $\hat{\pi}_2$ does not only attest to the correctness of $cFrag_{x_2}$, but also attests to the fact that Ursula has served two valid distinct requests until now. The same process is repeated until all requests served in a round are covered. At the end, Ursula produces a single proof $\hat{\pi}_\omega$ along with the number of served requests ω as the final proof. Figure 2 depicts this process pictorially.

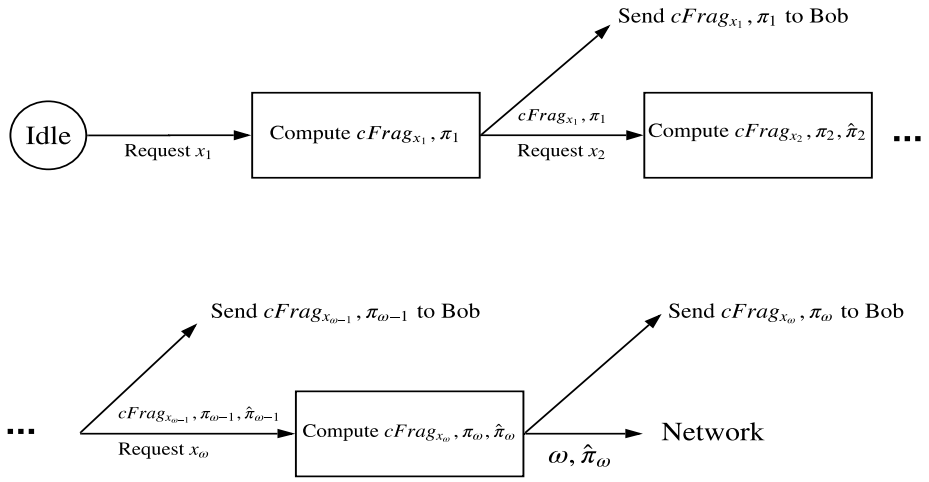

Fig. 2: PCD-based solution diagram. Ursula starts a round in the idle state. When the first request arrives, Ursula responds to the request and starts composing the proofs as new requests arrive. At the end of the round, Ursula informs the network (i.e., verifier) of the number of served requests and a single proof attesting to her claim.

An alternative, possibly more efficient approach, is to view the statement "Ursula has served ω distinct re-encryption requests correctly" as an instance of the arithmetic circuit satisfiability problem and utilize a highly efficient ZKP system, e.g., Bulletproofs [6], to prove its correctness in zero knowledge. That is, build a circuit that takes the set of correctness proofs of the served requests as well as their number ω as inputs, and then output 1 if number of correct distinct proofs equals to ω. Then, invoke a circuit parser [6] to convert the circuit into the Bulletproof format, which allow the prover, Ursula in this case, to produce a single proof.

In both cases, Ursula submits a single proof to the service policy contract. Once the proof is verified, the contract disperses the service fee (in proportion to ω) from Alice's locked Ether to Ursula's account.

5.2 Committee-Attestation Scheme

This solution shares a similar theme to confirm availability solution discussed previously. It relies on electing a committee for each round that will endorse Ursula's claim of serving ω distinct and valid re-encryption requests in a given round. The endorsement is done by employing CoSi (see Section 3.2) to permit the committee to collectively sign the statement "Ursula has served ω distinct re-encryption requests correctly."

In detail, for each round, a committee of Ursulas (and possibly outsider verifiers) is elected to attest for the work load an Ursula served. At the end of a round, each working Ursula establishes a record of all the requests she served along with their individual correctness proofs and the ciphertext fragments she computed. Then, she initiates the CoSi protocol to sign the workload statement above, denoted as m, while replacing ω with the actual number of served requests in the round. Ursula sends m along with the full service record to each member of the committee, which in turn verfiies the correctness of the report as follows:

1. Check that each request is distinct by checking the sequence number/timestamp of the request.
2. Check that each request is valid by verifying Bob's signature over the request. (Here the policy will contain Bob's public key to allow the committee to use the right verification key.)
3. Verify the ZKP correctness proof of each $cFrag$ as in the Umbral scheme.
4. Check that the value of ω inside m agrees with the record.

If everything is fine, each member of the committee and the working Ursula finalize the CoSi signing protocol and produce one signature over m. At the end, the working Ursula publishes m and the collective signature to the network, or the service policy contracts, which in turn verifies correctness as described in Section 3.2.

On the committee election. For each round some reasonable number of Ursulas will be elected by using a similar technique to the ones described in

Section 4, i.e., using the hash iterative process. The selection may also take into account the presence and size of each Ursula's stake, to avoid an attacker spinning up enormous numbers of Ursulas in order to increase the chance of them controlling the entire committee.

The above scheme works under the assumption that at least one Ursula in the elected committee is honest, which pertains to an assumption on the least number of honest Ursulas in the network. If the latter assumption is under threat, one mitigating approach is to increase the committee size. Another approach is to have a changing set of external verifiers, like trusted partners, as part of the elected committee. As long as there is one honest party in the committee, signing fraudulent records will not succeed since this honest party will not agree to do that.

On the public verifiability of the signed records. Having a valid collective signature from a committee (that has at least one honest member) suffices for the correctness of the signed statement. However, it could be better to keep the records that produced the message m available for a while so that any party can verify them (the verifying parties could maintain such a public record for a given period after which the logs can be discarded). We can resort to this option just at the beginning to convince the participants of the trustworthiness of the committee or the validity of the assumption that at least one Ursula in the elected committee is honest.

On compensating the committee. Similar to confirm availability, we need a mechanism to ensure that the elected committee is responsive and willing to handle the endorsement work. Ideally, this can be pictured as a collaborative work, the committee does the work so that others will do the same when the committee members want their workload to be signed. Alternatively, and as mentioned before, the committee can be paid for their work, either by Alice as part of her policy, or by the working Ursulas themselves since they are requesting this endorsement service (however, the latter may complicate the system operation).

5.3 Commit/Challenge/Open-based Scheme

This solution is inspired from the Merkle tree-based ZKP system proposed in [7]. Ursula keeps track of the requests coming from Bob(s) during a time round and assign each of them a unique sequence number. That is, Ursula replies with $cFrag$ and the correctness proof. At the end of the round, Ursula constructs a Merkle tree of all requests, where the requests (and their replies/correctness proofs) are the leaves of the tree ordered by their sequence numbers (recall that we require Bob to include a sequence number in his request to ensure freshness). Ursula signs the root of the tree, denoted as $root$ to produce a signature σ_{root}

To prove correctness of the claimed work amount, Ursula will be challenged to open some leaves in the Merkle tree. She uses the $root$ as a random beacon to be fed into an iterative hash process to select n leaf nodes at random to open them (i.e., these are the challenges). Ursula provides the challenge leaf nodes

along with their membership paths in the tree, along with $(\omega, root, \sigma_{root})$, either to a sidechain (which is maintained by round-elected committee) or to the policy contract to verify.

The verification process involves verifying Ursula's signature on the *root*, verifying the freshness and correctness of the request in each opened leaf, and verifying the membership path of each of these leaves. If everything is fine, Ursula will be paid for serving the claimed ω request.

It should be noted that the security provided here is statistical, driven by the value of n, in the sense that we need to bound the probability by which an Ursula can cheat, such as replicating requests in the tree and adding fabricated ones to increase ω.

6 Conclusion and Future Work

As we mentioned earlier, now in the NuCypher network, we have the naive confirm activity function calling approach, with a financial punishment of slashing the stake of any Ursula who does not log her activity periodically. Our early plans will be to replace this approach with the confirm availability solution introduced in Section 4.

For the confirm service activity, which will be deployed at a later of the network, which solution to adopt will be based on benchmark testing and the development vision of the NuCypher network. The ZKP-based solution (the one utilizing bulletproofs) provides several advantages including non-interactivity, as opposed to the committee-attestation one that needs interaction with the committee. It also supports computational soundness as opposed to the statistical one of the commit/challenge/open solution. Nonetheless, it could be the case that the committee-attestation and the Merkle-tree based solutions are more efficient especially if we incorporate committee attestation or sidechains to support other functionalities in the network.

Another related issue to confirm activity is the service payment strategy in the network. Until now we assume that Alice pays for the service by using the Ether she locks into the policy contract. Other arrangement may emerge in the system like having Bob pay for the requests he issues. Such an arrangement may have an implication on the confirm activity issue, and studying its impact, if it becomes part of the NuCypher network protocol, will be part pf our future work.

References

1. Nucypher. `https://www.nucypher.com/`.
2. Michael Egorov, MacLane Wilkison, and David Nuñez. Nucypher kms: decentralized key management system. *arXiv preprint arXiv:1707.06140*, 2017.
3. David Nuñez. Umbral: A threshold proxy re-encryption scheme. 2018. `https://github.com/nucypher/umbral-doc/blob/master/umbral-doc.pdf`.

4. Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. In *ICS*, volume 10, pages 310–331, 2010.
5. Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Theory of Cryptography Conference*, pages 1–18. Springer, 2008.
6. Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334. IEEE, 2018.
7. Nico Döttling, Russell WF Lai, and Giulio Malavolta. Incremental proofs of sequential work. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 292–323. Springer, 2019.
8. Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. Keeping authorities" honest or bust" with decentralized witness cosigning. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 526–545. Ieee, 2016.
9. Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.
10. Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 390–399, 2006.
11. Silvio Micali, Kazuo Ohta, and Leonid Reyzin. Accountable-subgroup multisignatures. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 245–254, 2001.
12. Ghada Almashaqbeh, Allison Bishop, and Justin Cappos. Microcash: Practical concurrent processing of micropayments. In *International Conference on Financial Cryptography and Data Security*, pages 227–244. Springer, 2020.
13. Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*, pages 120–130. IEEE, 1999.
14. *NIST Random Beacon*. `https://beacon.nist.gov/home`.
15. Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. On bitcoin as a public randomness source. *IACR Cryptology ePrint Archive*, 2015:1015, 2015.