Prerequisites: A note on data availability and erasure coding or Section 5.4 (Selective Share Disclosure) in the Fraud Proofs paper.

So far, we know how to design light clients that can convince themselves that block data is available, and that the network isn't merely selectively releasing chunks to them ("selective share disclosure"). This can be achieved by using onion routing so that sample requests for chunks can be made anonymously, and having sample requests from multiple light clients be sent simultaneously, so that sample requests are unlinkable with each other.

But what if you wanted to prove to someone else

that block data is available? Such a proof can be accepted by a smart contract on the blockchain. This could be handy for sidechain-type constructions such as Plasma, so that the contract on the main chain can verify data availability before accepting new block headers for sidechains.

One approach we can take is that we can run a light client itself as a smart contract. A third party that is observing the blockchain can then convince themselves that data is available by verifying that the smart contract was executed correctly. However the problem is that smart contracts of course cannot make anonymous network requests to check that samples are available.

However, the reason why onion routing provides anonymity is because of the cryptographic aspects of the protocol, rather than the "network-layer" aspects: unless all of the private keys in the nodes in the onion circuit are compromised or corrupted, then the circuit provides anonymity guarantees. So what we can do is construct a protocol where a set of players submit responses (i.e. chunks) to a random number (i.e. sample) received by the network via an onion circuit, and prove that this onion circuit was generated randomly. As long as all the nodes in the circuit aren't all colluding with each other, anonymity is maintained.

To construct the the protocol below, we first make the following assumptions:

1. There is a set of relay nodes and their public keys $\mathsf{relayNodes}$

to pick from known by the chain (which, just as in Tor, must be a Sybil-free set). Each node has its own verifiable number function (VRF) based on these public keys.

1. There is a set of players $\mathsf{players}$

that are willing to participate in the protocol.

1. There is a source of public verifiable randomness available to the blockchain, e.g. using a RANDAO and VDFs.

The protocol can work as follows, for one sample. The process can be repeated for multiple samples. I'm assuming a circuit with three nodes (the minimum); it could be extended to more.

1. A set of players commit to the hash of a secret number on-chain.

2. The chain generates a verifiable random number. This random number is used to pick a random player P

from step 1.

1. P

concatenates their secret with the chain's random number, and gets R_0

. This random number is used as a seed to a deterministic function $\mathsf{pickNode}$

that picks a random relay node $\mathsf{pickNode}(\mathsf{relayNodes}, R_0) = N_1$

from $\mathsf{relayNodes}$

.

1. P

sends N_1

the number R_0

.

1. N_1

computes $\mathsf{VRF}_{N_1}(R_0) = R_1$

and $\mathsf{pickNode}(\mathsf{relayNodes}, R_1) = N_2$

. N_1

sends R_1

to N_2

.

1. N_2

computes \mathsf{VRF}_{N_2}(R_1) = R_2

and \mathsf{pickNode}(\mathsf{relayNodes}, R_2) = N_3

. N_2

sends R_2

to N_3

.

1. N_3

, the exit node, computes \mathsf{VRF}_{N_3}(R_2) = R_3

. N_3

then attempts to download the chunk associated with the number R_3

from the network (e.g. R_3 \mod number_of_chunks

).

1. If N_3

successfully downloads the chunk then it sends it to back to N_2

, who sends it back to N_1

, who sends it back to P

, as well as all of the identities of these nodes and the VRF proofs, which is revealed to P

after the response. P

can then submit this chunk, the Merkle proof that its in the data, all of the identities of N_1, N_2, N_3

, and its secret committed number to a smart contract that can verify all of the VRFs were computed correctly, and accept the data availability proof for that sample.

1. If N_3

was unsuccessful in downloading the chunk, it sends a special “fail” message back using the same method in step 8, in which case the smart contract considers that the chunk is unavailable.

Note that this is no longer an ‘onion’ circuit, but a ‘flat’ circuit where the actual message is generated at the exit node. In order for requests coming from smart contracts to be indistinguishable from requests coming from standard light clients, all light clients should also be using same anonymity network with the same protocol from step 4 and after.

Assuming that not all of N_1, N_2, N_3

are colluding, then the network should not be able to know that the request came from a smart contract, even if P

was dishonest, because P

has no idea what chunk is going to be sampled from the network until after

it receives the response.

The above protocol can be repeated S

times to make S

samples; if all samples receive a successful response then the smart contract can consider the data to be available. A third party that is observing the blockchain can then too convince themselves that the data is available, without having to undergo an interactive challenge-response protocol.

Data unavailability

So we have a protocol that can be used to prove data availability, but what if P

or some relay node drops out of the protocol an doesn't send anything or N_3

sends a fail message despite the chunk actually being available? Just because this happens, that doesn't mean the data is unavailable: you can't prove the absence of data. This is the main challenge of using such a protocol in an application - figuring out what to do in such a case, without allowing an attacker to bias what samples are being made. The solutions may be application-specific.

There are three main cases to think about:

1. P

is byzantine and doesn't respond because it wants to block the protocol, or because it receives a "fail" message but doesn't want to submit it because it doesn't want the smart contract to know the data is unavailable.

1. N_3

, or some intermediate relay node, has a network failure and wasn't able to get the chunk, talk to the next node or respond.

1. N_3

responds with "fail" when it shouldn't, and P

submits the failure to the smart contract.

Case 2 and 3 are less of a risk to the protocol's security because network failures in intermediate and exit nodes theoretically cannot be targeted at P

. For example if N_3

wants to not respond when asked for a certain chunk, it must do so for all requests it receives, because it doesn't know which one came from P

. That problem then boils down to the general problem of how do you incentivise nodes to do their jobs and be reliable. One can attempt to construct protocols that rewards relay nodes for participating in the network and punishes or slashes nodes that are unreliable (e.g. this paper on proof-of-bandwidth for relays), this is a different but important topic.

In case 1 however, P

may intentionally withhold the response from the smart contract. Note that case 1 and 2 are indistinguishable, so we have to deal with them in the same way, even though case 2 can happen when P

is honest. If for example the smart contract wants 30 random samples but only 29 have a response, we cannot simply make the smart contract run another round of the protocol for another random sample, otherwise P

can bias what that final response is. So we have to do all

30 samples again. That means if for example's there's a 99.99% chance of not only landing on available chunks with 30 samples, then on average you'd need to repeat the protocol 10,000 times on-chain for an attacker to win.

To further discourage P

from not responding, we could make the deposit some coins that will be slashed if they time out. However this will also punish them for case 2 failures that are not their fault (but also, those failures cannot be targeted to P

by relay nodes if P

also sends it requests over an onion router). This might not be a problem if P

also gets rewarded for participating, as the reward might out weight the risk of not being able to respond.

In case 3, there is an explicit fail message. What to do in such a case is dependant on the application. In a Plasma-like use case, the smart contract could assume that exit nodes are reliable and reject the block, in which case another block can be mined. The huge caveat is that the exit nodes are reliable and incentivised: if they aren't, then they could choose to censor blocks they don't like. There are ideas and tradeoffs that can be explored to prevent this, such as allowing the smart contract to tolerate one or two failures (e.g. maybe there’s an exit node that’s always bad), but that increases the chance of being fooled into thinking an unavailable block is available.

Such a protocol may be more appropriate for more general use cases that don’t have an immediate time pressure to make sure data is available (let’s say I want to prove to a smart contract that I actually published something to BitTorrent so that I can get paid for it, e.g. the result of a TEE computation), so that if the data was found to be unavailable by the smart contract, it may try again after an interval, e.g. one hour. But it would be important to make sure that there’s sufficient people making sampling requests so that deanonymisation can’t be done by looking at the time when the smart contract made the request to when it was received.

Thanks to Jeff Burdges and Jeff Coleman for discussions that led to this post.