Proof of Space New Definition and Requirements

No Author Given

CryptoNet, Protocol Labs

1 PoS Definition

A *Proof of (Useful) Space* is a protocol that allows a prover to convince a verifier that he has a minimum specified amount of space (ie, used storage) committed to store some data D. More precisely in a PoS protocol we have two main sub-protocols:

- **Initialization (setup phase)**: on public input N (size) and D (data), a *replica* R of size N is created, which is a an encoded version of the data.
  The replica is stored by the prover, while a commitment to R and D is publicly available.
- **Execution (audit phase)**: the prover gets challenged and they answer with a proof that can be valid/invalid. Valid means that any verifier is convinced that the prover stores the replica. This phase can be repeated many times. We parametrize the audits in terms of t, the time between challenge and response (on-chain communication) and d the time between two consecutive challenges

We call honest prover one that store the entire replica; while for the analysis we consider a malicious prover trades storage for computation and regenerates on demand to pass the audit phases.

1.1 A model for PoS

1.2 Syntax

A PoS is a tuple of algorithms $\mathsf{Setup}, \mathsf{Init}, \mathsf{Challenge}, \mathsf{Response}$

- The setup algorithm $\mathsf{pp}_{\mathrm{setup}} \leftarrow \mathsf{Setup}(\mathsf{secprms})$: produces in a trusted manner all public parameters of the scheme (e.g., a CRS)
- $(C^*, \mathsf{aux}_{\mathrm{repl}}) \leftarrow \mathsf{init}(\mathsf{pp}_{\mathrm{setup}}, D)$: produces a commitment C^* and a replica $\mathsf{aux}_{\mathrm{repl}}$ to the replica from the data
- $\mathsf{chlg} \leftarrow \mathsf{Challenge}(\mathsf{pp}_{\mathrm{setup}}, C^*)$
- $\mathsf{resp} \leftarrow \mathsf{Response}(\mathsf{pp}_{\mathrm{setup}}, \mathsf{chlg}, \mathsf{aux}_{\mathrm{repl}})$ // response to audit
- $\mathsf{accept/reject} \leftarrow \mathsf{verify}(\mathsf{pp}_{\mathrm{setup}}, C^*, \mathsf{chlg}, \mathsf{resp})$ // verification step

Defining secprms**.** We use the following tuple of security parameters for the latency model $\mathsf{secprms} = (\epsilon, M, \tau)$ (resp. $\mathsf{secprms} = (\epsilon, M, \rho)$ in the cost model)in our definition

- ϵ: advantage of a malicious provider
- M: a bound on its memory
- τ: a lower bound on its parallel time (latency model); see Definition 2
- ρ: a lower bound related to its "cost" (cost model); see Definition 4

When denoting asymptotics, if it is not important in which security parameter we are denoting them, we write $\mathsf{poly}(\mathsf{secprms})$. We will assume, for example, that the data size is $|D| = \mathsf{poly}(\mathsf{secprms})$.

Remark 1 (Encoding/Decoding). The following is expressed only informally for now and not required for the rest of the ideas currently written in this doc: there should exist an algorithm that goes from auxiliary input (replica) to the actual, i.e. a decoding algorithm. For this to be useful we also require encoding (the initialization stage producing the auxiliary string) to have binding properties.

1.3 Efficiency requirements and other remarks

- The size of the commitment C^* to the replica should be small
- The challenge, as clear from notation, is not computed depending on D
- Ther model in this writeup assumes the initialization is performed honestly; this can be dealt through a probabilistic check. This is currently out of scope of this document.
- Public-verifiability: no secrets to verify auditing

1.4 Correctness

For all security parameter tuples $\mathsf{secprms}$, for all data D, it holds with probability 1 that

$$\mathsf{verify}\left(\mathsf{pp}_{\mathrm{setup}}, C^*, \mathsf{chlg}, \mathsf{Response}\left(\mathsf{pp}_{\mathrm{setup}}, \mathsf{aux}_{\mathrm{repl}}\right)\right) = 1$$

where $(C^*, \mathsf{aux}_{\mathrm{repl}}) \leftarrow \mathsf{init}(\mathsf{pp}_{\mathrm{setup}}, D)$ and $\mathsf{pp}_{\mathrm{setup}} \leftarrow \mathsf{Setup}(\mathsf{secprms})$ and $\mathsf{chlg} \leftarrow \mathsf{Challenge}(\mathsf{pp}_{\mathrm{setup}}, C^*)$.

1.5 An example construction

- Setup algorithm $\mathsf{pp}_{\mathrm{setup}} \leftarrow \mathsf{Setup}(\mathsf{secprms})$:
  1. produces description of a hash function H, SNARK parameters, ...
- $(C^*, \mathsf{aux}_{\mathrm{repl}}) \leftarrow \mathsf{init}(\mathsf{pp}_{\mathrm{setup}}, D)$:
  1. makes a Merkle tree commitment root to D. Call it C_D (this is aka comm_D in other documentation)
  2. produces the last layer L of labels in an appropriate labeled stacked DRG as $\mathsf{hashDataIdToGraph}(\mathsf{pp}_{\mathrm{setup}}, C_D)$
  3. Computes $R = L + D$ (obtained by element-wise sum)
  4. Commit to the vector R through a Merkle tree T_R with root root_R; return root_R as C_{rep}
  5. Computes and returns publicly the pair $C^* = (C_D, C_{\mathsf{rep}})$
  6. Computes and returns locally $\mathsf{aux}_{\mathrm{repl}}$ consisting of: 1) R; 2) the whole Merkle Tree T_R w.r.t. root C_{rep}
- $\mathsf{chlg} \leftarrow \mathsf{Challenge}(\mathsf{pp}_{\mathrm{setup}}, C^*)$:
  1. Sample $\mathcal{J} = \{j_1, \ldots, j_k\}$, k indices between 1 and $|D|$. The number k depends on the security one wants to achieve.
- $\mathsf{resp} \leftarrow \mathsf{Response}(\mathsf{pp}_{\mathrm{setup}}, \mathsf{chlg}, \mathsf{aux}_{\mathrm{repl}})$ // response (online stage of adversary)
  1. Return the opening of R for the indices in $\mathcal{J}$ together with all the corresponding Merkle opening (these can be compressed through a SNARK as well)
- $\mathsf{accept/reject} \leftarrow \mathsf{verify}(\mathsf{pp}_{\mathrm{setup}}, C^*, \mathsf{chlg}, \mathsf{resp})$
  1. Verifies the response step appropriately (checks Merkle paths, etc).

Security game, parametrized by security parameters $\mathsf{secprms} = (\epsilon, M, \tau)$

1. $\mathsf{pp}_{\mathrm{setup}} \leftarrow \mathsf{Setup}(\mathsf{secprms})$
2. $(D, \mathsf{aux}_{\mathsf{A}}) \leftarrow \mathsf{A}(\mathsf{pp}_{\mathrm{setup}})$ // offline stage of adversary(potentially deviating storage provider)
3. $(C^*, \mathsf{aux}_{\mathrm{repl}}) \leftarrow \mathsf{init}(\mathsf{pp}_{\mathrm{setup}}, D)$ // C^* is broadcasted
4. $\mathsf{chlg} \leftarrow \mathsf{Challenge}(\mathsf{pp}_{\mathrm{setup}}, C^*)$
5. $\mathsf{resp} \leftarrow \mathsf{A}(\mathsf{aux}_{\mathsf{A}})$ // response (online stage of adversary)
6. $\mathsf{verify}(\mathsf{pp}_{\mathrm{setup}}, C^*, \mathsf{chlg}, \mathsf{resp})$ // verification step

Fig. 1. Abstract Security Game for PoS

1.6 Security

Definition 1 (Admissible adversary). *An adversary is (ϵ, M)-admissible if in Fig. 1:*

- *it runs in polynomial time in step 2*
- *it has at least ϵ prob of success*
- *it uses at most M memory in storage (defined as the size of $\mathsf{aux}_{\mathsf{A}}$ used at steps 2 and 5)*[1]

Latency-based security

Definition 2 ((ϵ, M, τ)-soundness of PoS). *A PoS is (ϵ, M, τ)-sound in the latency model if, for all (ϵ, M)-admissible algorithms A, their parallel running time in step 5 of Fig. 1 is at least τ.*

Cost-based security We start with an informal treatment:

Security requirement (tradeoffs). [informal]

Consider a party using *too little* space. If they are satisfactorily responding to the challenges at auditing time, then their *incurred cost is substantially higher* than that of the honest party.

Notice that we talk about both "saving up space" and "cost" above. These two items are separated for a reason and, while related, will necessarily coincide.

- Space is what we want to force a party to use: if it is using enough space, it is not deviating. It is just playing by the rules and therefore it is not an "admissible" adversary (see below).
- cost may also include: RAM, hash computations, other expensive operations, etc. We keep it abstract here, but the bottom line is that this needs to be lower than that of the honest party.

[1] Notice that more fine-grained constraints on memory are possible.

Informally, a cost function describes the cost—in USDs or else; in general through a positive integer—of a strategy (honest or adversarial). See example below definition.

Definition 3 (Cost functions for PoS). *We denote the honest cost of a PoS by* cost_{hon} *which maps a tuple of security parameters to a number* $c_{hon} = \mathsf{cost}_{hon}(\mathsf{secprms}) > 0$. *The honest cost function of a PoS depends on the PoS construction. An adversarial cost profile is a family of functions parametrized by an adversary* A *is instead dependent on the adversary and is denoted by* $\mathsf{cost}_{\mathsf{A}}$. *Fixed* A, *it returns a positive number from the security parameters as input* $\mathsf{cost}_{\mathsf{A}}(\mathsf{secprms})$.

Examples for the above: one can instantiate the honest cost function as some linear function in the number of stored bits, e.g., 0.001$ per bit (arbitrary number); the adversarial cost profile could measure, e.g., the number of invocations of the random oracle.

Definition 4 ($(\epsilon, M, \rho, \mathsf{cost}_{\mathbf{hon}}, \{\mathsf{cost}_{\mathsf{A}}\}_{\mathsf{A}})$-soundness of PoS (cost-based)). *Fix a PoS construction and let* cost_{hon} *and* $\{\mathsf{cost}_{\mathsf{A}}\}_{\mathsf{A}}$ *as in Definition 3. We say a PoS construction is* $(\epsilon, M, \rho, \mathsf{cost}_{hon}, \{\mathsf{cost}_{\mathsf{A}}\}_{\mathsf{A}})$*-sound in the cost model if, for all* (ϵ, M)*-admissible algorithms* A, *the adversary's cost is at least a* ρ*-fraction more expensive than the honest strategy, that is* $\mathsf{cost}_{\mathsf{A}}(\mathsf{secprms}) \geq \mathsf{cost}_{hon}(\mathsf{secprms}) + \rho \cdot \mathsf{cost}_{hon}(\mathsf{secprms})$.