Publicly-verifiable proof of storage: a modular construction

Federico Giacon

Ruhr-Universität Bochum
federico.giacon@rub.de

6th BunnyTN, Trent
17 December 2015

RUHR UNIVERSITÄT BOCHUM RUB

Proof of Storage

Proof of Storage (PoS)

A *Proofs of Storage* protocol allows a client to verify that a server is correctly storing a user's file.

A PoS protocol is *publicly-verifiable* when the verification doesn't require any secret parameter.

The parties involved are:

- The *server*, who stores the file.
- The *user*, who wants to store the file.
- A *verifier*, who checks that the file is being stored by the server.

Proof of Storage

Proof of Storage (PoS)

A *Proofs of Storage* protocol allows a client to verify that a server is correctly storing a user's file.

A PoS protocol is *publicly-verifiable* when the verification doesn't require any secret parameter.

The parties involved are:

- The *server*, who stores the file.
- The *user*, who wants to store the file.
- A *verifier*, who checks that the file is being stored by the server.

Proof of Storage

Proof of Storage (PoS)

A *Proofs of Storage* protocol allows a client to verify that a server is correctly storing a user's file.

A PoS protocol is *publicly-verifiable* when the verification doesn't require any secret parameter.

The parties involved are:

- The *server*, who stores the file.
- The *user*, who wants to store the file.
- A *verifier*, who checks that the file is being stored by the server.

What PoS is NOT

Three different problems:

- **Secrecy of the stored file.** The server can read any part of the file: a user must encrypt the file to maintain privacy.
- **Retrieving the file.** The server can refuse to give back the file to the user but still pass the verification, because it is actually storing the file.
- **Recovering a partially corrupted file.** Part of the stored file can become corrupted. The PoS protocol detects the problem, but doesn't restore the file.

What PoS is NOT

Three different problems:

- **Secrecy of the stored file.** The server can read any part of the file: a user must encrypt the file to maintain privacy.
- **Retrieving the file.** The server can refuse to give back the file to the user but still pass the verification, because it is actually storing the file.
- **Recovering a partially corrupted file.** Part of the stored file can become corrupted. The PoS protocol detects the problem, but doesn't restore the file.

What PoS is NOT

Three different problems:

- **Secrecy of the stored file.** The server can read any part of the file: a user must encrypt the file to maintain privacy.
- **Retrieving the file.** The server can refuse to give back the file to the user but still pass the verification, because it is actually storing the file.
- **Recovering a partially corrupted file.** Part of the stored file can become corrupted. The PoS protocol detects the problem, but doesn't restore the file.

What PoS is NOT

Three different problems:

- **Secrecy of the stored file.** The server can read any part of the file: a user must encrypt the file to maintain privacy.
- **Retrieving the file.** The server can refuse to give back the file to the user but still pass the verification, because it is actually storing the file.
- **Recovering a partially corrupted file.** Part of the stored file can become corrupted. The PoS protocol detects the problem, but doesn't restore the file.

Proof of Storage
Additional requirement

We can trivially obtain a PoS protocol if the user stores a hash h of its file f and the server sends back the whole file, f'.

`Verify`(f'): if $H(f') == h$ return *true*; else return *false*;

We require the protocol be efficient in term of bandwidth: the communication complexity must be much lower than the size of the file.

Applications

- **Cloud storage:** the user can verify that their files are correctly stored by the cloud provider: otherwise the server might remove files that are unlikely to be accessed (e.g., old backups).
- **Automated, trustless payment for file storage:** with public verifiability a trusted third party (or a smart contract) can verify that the server is storing the file and authorize the payment, without requiring any trust between the user and the server.

(pk, sk, st)

$$(f', st) \leftarrow \texttt{Encode}_{sk}(f)$$

$$\longrightarrow$$

$$c$$

$$\longrightarrow$$

$$\longleftarrow$$

$$\pi \leftarrow \texttt{Prove}_{pk}(f', c)$$

(pk, f')

$$b = \texttt{Verify}_{pk}(st, c, \pi)$$

Security

The protocol is *correct*:

$$\text{The server knows } f' \implies \text{the verification succeeds.}$$

The protocol is *secure*:

$$\mathcal{A} \text{ can verify} \implies \text{we can ``extract'' the file } f \text{ from } \mathcal{A}.$$

The result

Ateniese, Kamara, Katz – *ASIACRYPT 2009*

We can build a correct and secure *publicly-verifiable PoS protocol* based on the hardness of factoring in the random oracle model.

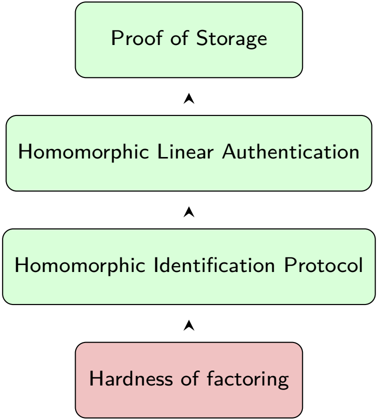
Additional properties

- Unlimited challenges.
- Public verifiability.

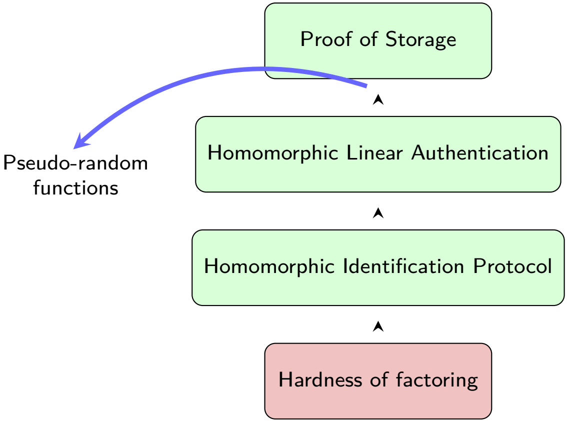
And with respect to the file size:

- Communication complexity: $\mathcal{O}(1)$.
- Server storage: the file f and a overhead $\mathcal{O}(1)$.
- Client storage: $\mathcal{O}(1)$.

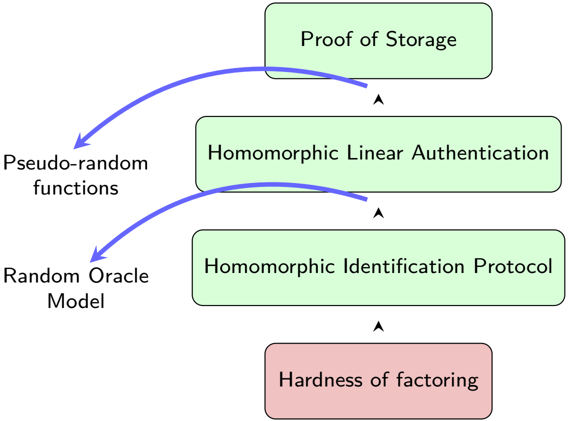
Modular construction



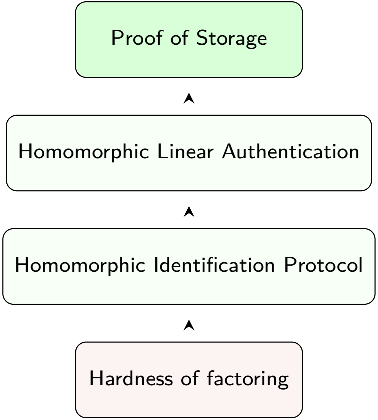
Modular construction



Modular construction



Proof of Storage



Homomorphic Linear Authentication

$\vec{f} = (f_1, \ldots, f_N)$

$$(\vec{t}', st) \leftarrow \mathtt{Tag}_{sk}(\vec{f}) \longrightarrow$$

$$\xrightarrow{\vec{c}}$$

$$\longleftarrow \tau \leftarrow \mathtt{Auth}_{pk}(\vec{f}, \vec{t}, \vec{c})$$

$$b = \mathtt{Verify}_{pk}(st, \mu, \vec{c}, \tau)$$

Homomorphic Linear Authentication

$$b = \mathtt{Verify}_{pk}(st, \mu, \vec{c}, \tau)$$

Correctness

$$\mathtt{Verify}_{pk}\left(st, \sum_i c_i f_i, \vec{c}, \tau\right) = 1$$

Security

A polynomial adversary cannot forge a valid authentication proof for $\mu \neq \sum_i c_i f_i$.

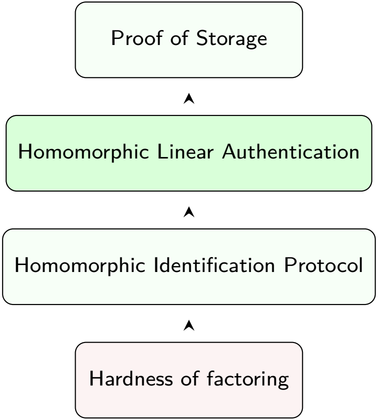
HLA $\Rightarrow$ PoS

The file is split: $\vec{f} = (f_1, \ldots, f_N)$, with $f_i \in \mathbb{Z}_p$.
The server stores a file with HLA tags: $\vec{f}' = (\vec{f}, \vec{t})$.

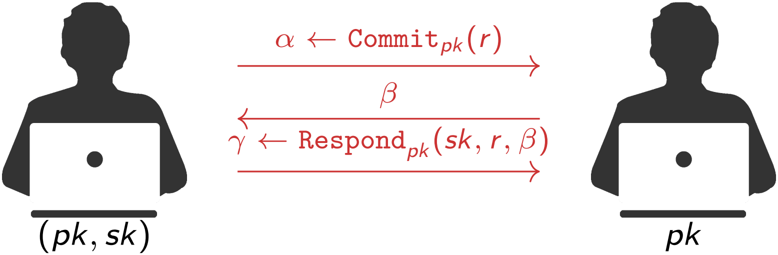
The request procedure consists in:

- Share a key K for the pseudo-random function F: the commitment is $c_i = F_K(i) \in \mathbb{Z}_p$.
- Using the HLA protocol we compute $\tau \leftarrow \mathtt{Auth}_{pk}(\vec{f}, \vec{t}, \vec{c})$ and set $\mu = \sum_i f_i c_i$. The PoS proof is $\pi = (\mu, \tau)$.
- To verify we use `Verify` of the HLA protocol.

Homomorphic Linear Authentication



Verify that the user knows the secret key *sk* without revealing additional information.



$$b = \texttt{Verify}_{pk}(\alpha, \beta, \gamma)$$

Homomorphic Identification Protocol

For an homomorphic protocol we add $\texttt{Combine}_1$ and $\texttt{Combine}_3$.

Correctness

If $\{(\alpha_i, \beta_i, \gamma_i)\}_{i=1}^{n}$ is a set of valid IP transcript ($\texttt{Verify} = 1$) and $\vec{c}$ is a vector:

$$\texttt{Verify}_{pk}\left(\texttt{Combine}_1(\vec{c}, \vec{\alpha}), \sum_i c_i \beta_i, \texttt{Combine}_3(\vec{c}, \vec{\gamma})\right).$$

Security

A polynomial adversary $\mathcal{A}$ has negligible probability to output a string $(\vec{c}, \mu', \gamma')$ such that:

- $\mu' \neq \sum_i c_i \beta_i$.
- $\texttt{Verify}_{pk}\left(\texttt{Combine}_1(\vec{c}, \vec{\alpha}), \mu', \gamma'\right) = 1$.

HIP $\Rightarrow$ HLA

The file is split: $\vec{f} = (f_1, \ldots, f_N)$, with $f_i \in \mathbb{Z}_p$. Implicitly $\beta_i = f_i$.

$\mathtt{Tag}_{sk}(\vec{f})$: We take a random element st and we fix $r_i = H(st, i)$ and $\alpha_i = \mathtt{Commit}_{pk}(r_i)$. We compute $\gamma_i = \mathtt{Respond}_{pk}(sk, r_i, f_i)$. The tag is $\vec{t} = (\gamma_1, \ldots, \gamma_n)$.

$\mathtt{Auth}_{pk}(\vec{f}, \vec{t}, \vec{c})$: Output the combined tag $\tau = \mathtt{Combine}_3(\vec{c}, \vec{t})$.

$\mathtt{Verify}_{pk}(st, \mu, \vec{c}, \tau)$: Output the combined verification $\mathtt{Verify}_{pk}(\mathtt{Combine}_1(\vec{c}, \vec{\alpha}), \mu, \tau)$.

Homomorphic Identification Protocol

Proof of Storage

↑

Homomorphic Linear Authentication

↑

Homomorphic Identification Protocol

↑

Hardness of factoring

Instantiation

Quadratic residuosity: $N = pq$, $\mathcal{J}_N^{+1}$ is the set of element in $\mathbb{Z}_N$ with Jacobi symbol $+1$, and $\mathcal{QR}_N$ is the set of quadratic residues mod N.

$y \xleftarrow{\$} \mathcal{QR}_N$; $pk = (N, y)$; $sk = (p, q)$.

$\texttt{Commit}_{pk}(r)$: output $\alpha = r$ as element of $\mathcal{J}_N^{+1}$.

$\texttt{Respond}_{pk}(sk, r, \beta)$: output γ, a random 2^{3k}-th root of $\pm r y^{\beta}$.

$\texttt{Verify}_{pk}(\alpha, \beta, \gamma)$: output 1 if and only if $\gamma^{2^{3k}} = \pm \alpha y^{\beta}$ and $\beta < 2^{3k}$.

$\texttt{Combine}(\vec{c}, \vec{x})$: output $\prod_i x_i^{c_i}$.

Thank you for your attention.