Public Ledger for Sensitive Data

Riccardo Longo

April 26th 2021

DeCifris Athesis

Sensitive Data after GDPR

- Personal data often needs to be shared with various Service Providers
- Sensitive data must be carefully managed
- Modern privacy regulation (GDPR) gives some rights to the data owner:
  - control over data sharing
  - ability to revoke access (right to be forgotten)
- Compliance requires *privacy by design*:
  - need-to-know policy
  - principle of least priviledge

Decentralized Storage

- Data Management is not easy:
  - sharing and availability
  - backups
  - consistency
  - access management and confidentiality
- Decentralized storage (*Cloud*) offloads various issues
- Availability and Integrity are improved, but Confidentiality becomes harder

End-To-End Encryption

- Cloud providers facilitate data storage and sharing
- Data management is possible without access to actual contents
- Cloud provider only sees/stores/shares encrypted data
- Data owner manages sharing by giving decryption keys only to rightful recipients
- Keys are shared directly from data owners to service providers that need access to data, so the cloud provider never sees cleartext

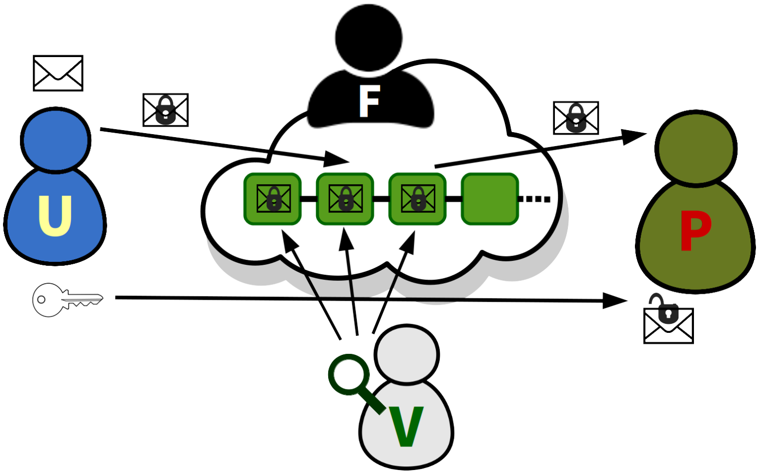
Access Control and Key-Management

- To granularly control access each file should have its own unique key
- Data owners should have exclusive control over the keys
- Local key-management is hard (the more keys, the harder it gets):
  - Availability (bakcups, sharing with recipients)
  - Confidentiality
- Key Encapsulation Mechanisms (KEM) to the rescue:
  - only one key (Key-Encrypting Key or KAK) to manage locally
  - per-file keys stored encrypted on the cloud
- KEM paired with asymmetric cryptography allows practical sharing of encrypted data

Access Revocation

- Data owner should be able to revoke access
- Revocation = prevention of further access to shared data
- Local copy problem:
  - sharing recipient saves a local copy of data
  - revocation becomes useless because local access is always available and its prevention is not enforceable
  - large scale data duplication however is expensive
  - therefore revocation should aim at making further access as hard as maintaining a local copy
- Sharing is usually carried out through access keys
- Revocation invalids such keys e.g. through data re-encryption

Simple Model



Data Integrity

- Many application require trusted data:
  - certified origin
  - not modified / tampered with after creation
- Append-only systems (blockchains) help against tampering
- Data verifiability hindered by encryption
- Revocation and re-encryption apparently prevent public verifiability

Verifying Encrypted Data and Revocation

- The goal is to enable an independent verifier to check integrity
- In general the verifier should not be able to access sensitive data
- The verification must therefore be carried out on the ciphertext
- If we re-encrypt to revoke access to shared keys the integrity can no longer be checked by any independent verifier
- Idea: split the encryption

Split Encryption (1)

- The data D is encrypted with key K_1 obtaining the ciphertext C
- The key K_1 is encrypted with the key K_2 obtaining the encapsulation E_1
- C and E_1 are stored on the cloud
- To share access to the data D the key K_2 is shared
- With K_2 it is possible to recover K_1 from E_1, and then D from C

Split Encryption (2)

- To revoke access re-encrypt K_1 with $\tilde{K}_2$ obtaining $\tilde{E}_1$ and substitute E_1 with on the cloud
- K_2 cannot decrypt $\tilde{E}_1$, so it can no longer recover K_1 and therefore D
- C remains unchanged so an independent verifier can still check its integrity
- The revocation is effective if K_1 is as expensive as D to maintain locally, so it makes sense to retrieve it from the cloud
- K_2 can be a small key, but each file should have a distinct, random key, and the Data Owner must appropriately store, safeguard and manage them

One-Time-Pad

- D is a string of bits D_i, $1 \leq i \leq \ell$
- K_1 is a string of bits $K_{1,i}$, $1 \leq i \leq \ell$
- The ciphertext C is computed bit-by-bit as $C_i = D_i \oplus K_{1,i}$, for $1 \leq i \leq \ell$
- If K_1 is random then we have a perfect cipher
- Note that D and K_1 have the same length, moreover whereas D may be compressible, K_1 is not, so it is not more convenient to store locally

Efficient Storage and Revocation (1)

- A direct implementation of the method introduced weighs down storage considerably:
  - both D and E_1 have to be stored, so every file takes up double the space
  - to implement time-constrained access and revoke everything periodically each E_1 has to be updated
- To increase efficiency E_1 may be shared between multiple files $D^{(j)}$
- Different keys $K_2^{(j)}$ derive from the same E_1 different keys $K_1^{(j)}$
- These derived keys $K_1^{(j)}$ can be safely used to encrypt $D^{(j)}$ into $C^{(j)}$

Efficient Storage and Revocation (2)

- The cloud has to store the individual $C^{(j)}$ and a single E_1
- Revocation is performed through a single update of E_1 into $\tilde{E}_1$
- To share again access to $D^{(j)}$ compute a new $\tilde{K}_2^{(j)}$ that derives from $\tilde{E}_1$ the same $K_1^{(j)}$ as before
- To enable this computation and ease key management for the data Owner we add another level of encryption

Efficient Storage and Revocation (3)

- On the cloud are saved E_1 and $(C^{(j)}, E_2^{(j)})$ for each j
- The data owner has a single private key S that can derive $K_2^{(j)}$ from $E_2^{(j)}$ for each of their files
- All files are revoked in a single pass updating E_1 into $\tilde{E}_1$ and $E_2^{(j)}$ into $\tilde{E}_2^{(j)}$
- The update is performed in such a way that:
  - after the update, using S you derive from $\tilde{E}_2^{(j)}$ the keys $\tilde{K}_2^{(j)}$
  - using $\tilde{K}_2^{(j)}$, you derive from $\tilde{E}_1$ the first-level keys $K_1^{(j)}$
  - these $K_1^{(j)}$ are the same you could derive before the update from E_1 with $K_2^{(j)}$

The File Keeper

- The cloud is managed by the File Keeper F
- Its main job is to update E_1 and the $E_2^{(j)}$
- We include in $C^{(j)}$ some extra data that allows an independent verifier to check that E_1 and $E_2^{(j)}$ remain coherent even after the update
- F only sees encrypted data and has never access to decryption keys
- The threats that its misbehaviour may pose are DOS and revocation reversing
- Each misbehaviour is immediately detectable by any independent verifier, so there are multiple mitigation techniques

Updating Masking Shards Protocol: Actors

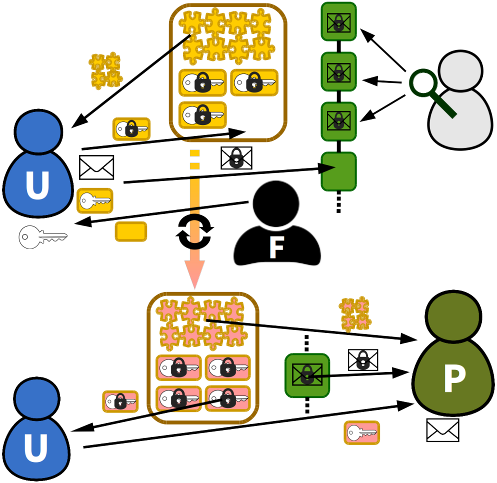
- users U_ℓ publish encrypted data
- file keeper F maintains the public ledger
- service provider P requests access to users

Encryption flow

- F publishes the *masking shards*
- U_ℓ combines a personal key with the shards to encrypt the files
- This key is encapsulated (encrypted) and stored on the *updating ledger* alongside the masking shards
- Encrypted data is stored in the *append-only ledger*, its integrity is guaranteed via chains of hash digests
- U_ℓ *unlocks* the encapsulated key, so P can combine it with the masking shards and decrypt the file
- The updating ledger is periodically updated (re-encrypted) by F, so unlocked keys no longer work

- E_1
- $K_1^{(j)}$
- $E_2^{(j)}$
- C
- $K_2^{(j)}$
- $E \to \tilde{E}$

Working Diagram



Pairing And Bilinear Groups

- Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be groups of the same prime order p, and e a pairing
- A **Pairing** is a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ with the following properties:
  - **Bilinearity**: $\forall g \in \mathbb{G}_1, h \in \mathbb{G}_2, \forall a, b \in \mathbb{Z}_p$, $e(g^a, h^b) = e(g, h)^{ab}$.
  - **Non-degeneracy**: for g_1, g_2 generators of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively, $e(g_1, g_2) \neq 1_{\mathbb{G}_T}$.
- $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are **Bilinear Groups** if the conditions above hold and the group operations in $\mathbb{G}_1, \mathbb{G}_2$, and $\mathbb{G}_T$ and e are efficiently computable
- Commonly implemented with the group of points of an elliptic curve over a finite field
- Tate and Weil pairings, in their non-degenerate version

Bilinear Decisional Diffie-Hellman Assumption

- Let $\alpha, \beta, \gamma, z \in \mathbb{Z}_p$ be chosen at random
- Let $\mathcal{B}(g_1, g_2, A = g_1^{\alpha}, B = g_2^{\beta}, C = g_2^{\gamma}, T) \to \{0, 1\}$ be an algorithm that distinguishes between

$$T = e(g_1, g_2)^{\alpha\beta\gamma} \qquad \text{and} \qquad T = e(g_1, g_2)^{z}$$

  outputting respectively 1 and 0
- The advantage of $\mathcal{B}$ is:

$$Adv_{\mathcal{B}} = \Big| \Pr\left[\mathcal{B}(g_1, g_2, A, B, C, e(g_1, g_2)^{\alpha\beta\gamma}) = 1\right] - \Pr\left[\mathcal{B}(g_1, g_2, A, B, C, e(g_1, g_2)^{z}) = 1\right] \Big|$$

- The **BDDH** assumption states that no probabilistic polynomial-time algorithm $\mathcal{B}$ has a more than **negligible advantage**

Random Strings from the Target Group

- To construct the keys for the One-Time-Pad we need bitstrings derived from random elements of $\mathbb{G}_T$ to be indistinguishable from strings uniformly distributed in $(\mathbb{F}_2)^\delta$

Definition (Uniform Mapping)

$\phi : \mathbb{G}_T \to (\mathbb{F}_2)^\delta$ is a *uniform mapping* of $\mathbb{G}_T$ of size δ if it is efficiently computable and no probabilistic polynomial-time algorithm $\mathcal{B} : (\mathbb{F}_2)^\delta \to \{0, 1\}$ has more than negligible advantage:

$$Adv_{\mathcal{B}} = \Big| \Pr[\mathcal{B}(\phi(g)) = 1] - \Pr[\mathcal{B}(s) = 1] \Big|$$

when g is chosen uniformly at random in $\mathbb{G}_T$ and s is chosen uniformly at random in $(\mathbb{F}_2)^\delta$.

- In practice ϕ is a hash function

Setup

- In *Updating Masking Shards Protocol* participate a file keeper F, a set of users $\{U_\ell\}_{1\leq\ell\leq N}$ and a service provider P
- Bilinear groups $\mathbb{G}_1, \mathbb{G}_2$ of prime order p are chosen according to a security parameter κ, along with generators $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2$.
- Let e be their pairing and $\mathbb{G}_T$ be the target group (of the same order p), with uniform mapping ϕ of size δ
- Let $|B|$ be the desired block length, then $l = |B|/\delta$ is the maximum number of shards in a block

File Keeper

- F chooses uniformly at random exponents $u_i \in \mathbb{Z}_p$ for $1 \le i \le I$
- F chooses a random time-key $s_0 \in \mathbb{Z}_p$ and publishes the initial masking shards:

$$\varepsilon_{i,0} = g_1^{u_i s_0} \qquad 1 \le i \le I.$$

- F securely saves the value s_0 but it can forget the exponents u_i
- F periodically updates the shards by choosing at time t_{j+1} a new random time-key $s_{j+1} \in \mathbb{Z}_p$ and computing for $1 \le i \le I$:

$$\varepsilon_{i,j+1} = \left(\varepsilon_{i,j}\right)^{\frac{s_{j+1}}{s_j}} = \left(g_1^{u_i s_j}\right)^{\frac{s_{j+1}}{s_j}} = g_1^{u_i s_{j+1}}.$$

User

- Each user U_ℓ chooses two private exponents $\mu_\ell, \nu_\ell \in \mathbb{Z}_p$ and publishes the public key:

$$q_\ell = g_2^{\mu_\ell}$$

- U_ℓ wants to encrypt a file $m_b^{(\ell)}$ at a time t_j, to publish it in the b-th block of the ledger
- U_ℓ divides the file $m_b^{(\ell)}$ in I_b pieces $m_{b,i}$, for $1 \leq i \leq I_b$, of equal length δ, where $I_b \delta$ is the length of $m_b^{(\ell)}$
- U_ℓ requests an encryption token $k_{\ell,j}$
- F takes the public key q_ℓ of the user and computes:

$$k_{\ell,j} = q_\ell^{\frac{1}{s_j}} = g_2^{\frac{\mu_\ell}{s_j}} .$$

Encryption

- U_ℓ chooses a random exponent $k_b \in \mathbb{Z}_p$ to calculate the encrypted shards:

$$\begin{aligned} c_{b,i} &= m_{b,i} \oplus \phi\left(e(\varepsilon_{i,j}, (k_{\ell,j})^{k_b})\right) \qquad 1 \le i \le l_b \\ &= m_{b,i} \oplus \phi\left(e\left(g_1^{u_i s_j}, g_2^{\frac{k_b \mu_\ell}{s_j}}\right)\right) \\ &= m_{b,i} \oplus \phi\left(e(g_1, g_2)^{u_i k_b \mu_\ell}\right) \end{aligned}$$

- U_ℓ calculates the encapsulated key:

$$K_{\ell,j,b} = (k_{\ell,j})^{\frac{v_\ell k_b}{\mu_\ell}} = \left(g_2^{\frac{\mu_\ell}{s_j}}\right)^{\frac{v_\ell k_b}{\mu_\ell}} = g_2^{\frac{v_\ell k_b}{s_j}}$$

- U_ℓ can forget the exponent k_b once this key has been computed

Control Shard

- Let $\bar{\imath} \equiv b \mod l$.
- F computes the control shard as:

$$c_b^* = \phi\left(e(\varepsilon_{\bar{\imath},j}, K_{\ell,j,b})\right) = \phi\left(e\left(g_1^{u_{\bar{\imath}} s_j}, g_2^{\frac{k_b v_\ell}{s_j}}\right)\right) = \phi\left(e(g_1, g_2)^{u_{\bar{\imath}} k_b v_\ell}\right)$$

- U_ℓ sends to F:
  - the digest $H\left(m_b^{(\ell)}\right)$
  - the ciphertext $c_b^{(\ell)}$
  - the control shard c_b^*
  - the encapsulated key $K_{\ell,j,b}$
- F inserts $c_b^{(\ell)}, H\left(m_b^{(\ell)}\right), c_b^*$, in the *append-only* public chain
- F inserts $K_{\ell,j,b}$ in the *updating ledger*

Key Update

- F has to periodically update not only the masking shards but also the encapsulated keys:

$$
\begin{aligned}
K_{\ell,j+1,b} &= (K_{\ell,j,b})^{\frac{s_j}{s_{j+1}}} \\
&= \left(g_2^{\frac{v_\ell k_b}{s_j}} \right)^{\frac{s_j}{s_{j+1}}} \\
&= g_2^{\frac{v_\ell k_b}{s_{j+1}}}
\end{aligned}
$$

- $s_{j+1} \in \mathbb{Z}_p$ is the same time-key used to update the shards
- s_j could be forgotten once the update has been completed

Key Unlocking

- Let P be a service provider that needs access to the file $m_b^{(\ell)}$, and therefore asks for permission to the file owner U_ℓ
- To grant access U_ℓ computes an unlocked key valid for the current time t_j
- U_ℓ retrieves from the updating ledger the encapsulated key $K_{\ell,j,b}$ and calculates:

$$\begin{aligned}\overline{K}_{\ell,j,b} &= (K_{\ell,j,b})^{\frac{\mu_\ell}{v_\ell}} \\ &= \left(g_2^{\frac{v_\ell k_b}{s_j}}\right)^{\frac{\mu_\ell}{v_\ell}} \\ &= g_2^{\frac{\mu_\ell k_b}{s_j}}\end{aligned}$$

Decryption

- With the unlocked key, P can decrypt the encrypted shards by computing:

$$
\begin{aligned}
m'_{b,i} &= c_{b,i} \oplus \phi\left(e\left(\varepsilon_{i,j}, \overline{K}_{\ell,j,b}\right)\right) \qquad 1 \leq i \leq l_b \\
&= m_{b,i} \oplus \phi\left(e(g_1, g_2)^{u_i k_b \mu_\ell}\right) \oplus \phi\left(e\left(g_1^{u_i s_j}, g_2^{\frac{\mu_\ell k_b}{s_j}}\right)\right) \\
&= m_{b,i} \oplus \phi\left(e(g_1, g_2)^{u_i k_b \mu_\ell}\right) \oplus \phi\left(e(g_1, g_2)^{u_i \mu_\ell k_b}\right) \\
&= m_{b,i}
\end{aligned}
$$

- Afterwards, P can compare the hash digest of the decrypted message $H(m'_{b,1} \parallel \ldots \parallel m'_{b,l_b})$ with the digest included in the append-only chain

Updating Ledger

It contains:

- the current time t_j
- the masked shards and their index:

$$(\varepsilon_{i,j}, i)_{1 \leq i \leq I}$$

- the encapsulated keys and the index of the data block where the corresponding encrypted pieces are stored:

$$(k_{\ell,j,b}, b)_{b \geq 1}$$

All these elements are kept constantly updated

Append-Only Ledger

Each data block B_b contains:

- the ciphertext, the digests of the original cleartext, and the control shard:

$$D_b = \left(c_b^{(\ell)}, H\left(m_b^{(\ell)} \right), c_b^* \right);$$

- the hash of the previous data block $H(B_{b-1})$.
- a cryptographic warranty of immutability of the block involving the digest

$$d_b = H(H(B_{b-1}), D_b)$$

Control Shards

- the index of the control shard covers the whole range $1 \leq i \leq I$
- these pieces are needed to check the integrity of the updating ledger
- In fact let $\bar{\imath} = b \mod I$, then for every time t_j it should hold:

$$
\begin{aligned}
c_b^* &= \phi\left(e(\varepsilon_{\bar{\imath},j}, K_{\ell,j,b})\right) \\
&= \phi\left(e\left(g_1^{u_{\bar{\imath}} s_j}, g_2^{\frac{k_b v_\ell}{s_j}}\right)\right) \\
&= \phi\left(e(g_1, g_2)^{u_{\bar{\imath}} k_b v_\ell}\right)
\end{aligned}
$$

- Any observer could check the coherence of the updating ledger (and consequently the behaviour of F)

Immutability of the block

- Immutability can exploit a pre-existing blockchain, embedding d_b in its blocks
- There are different approaches to achieve stand-alone immutability of the static block:
  - the signature of the user (owner of the data) on d_b
  - a proof of work involving d_b
  - a signature made by a third party (or a group or multi-party signature) on d_b
- All have pros and cons to their adoption, the optimal choice is probably a combination of the three

Lightweight Chain

- Exclude the actual encrypted data from the blocks, retaining only their digests
- bulk of data could be stored in distributed databases
- hashes are kept on the ledger to guarantee the integrity
- blocks are much smaller, so the chain can be widely replicated
- more actors could perform controls
- more difficult to forge

Shrunk Block

The shrunk block contains:

- the hash digest of the encrypted file, and the digest of the cleartext, and the control shard:

$$D'_b = \left(H\left(c_b^{(\ell)}\right), H\left(m_b^{(\ell)}\right), c_b^*\right)$$

- the hash of the previous block $H(B_{b-1})$
- a cryptographic warranty on

$$d'_b = H(H(B_{b-1}), D'_b)$$

Security Model

The goals of the protocol is to achieve the following security properties:

- **End-to-end encryption**: the File Keeper F must not be able to read the plaintext message at any time.
- **One-time access**: a Service Provider P should be able to read a plaintext message at the time t *if and only if* authorized by the file owner with an unlocked key for the time t.

Updating Masking Shards Protocol Security

Theorem (Security against Outsiders and Service Providers)

If an adversary can break the scheme, then a simulator can be constructed to play the BDDH game with non-negligible advantage.

Theorem (Security Against the File Keeper)

If an adversary can break the scheme, then a simulator can be constructed to play with non-negligible advantage:

- *the BDDH game if F is honest but curious*
- *the IDDH game in $\mathbb{G}_2$ if F is malicious*
- *the DDH game in $\mathbb{G}_2$ if F is malicious but we add a ZKP to the protocol (after the encryption token generation)*

Any question?

riccardolongomath@gmail.com

Thank You