Protocole SÉRÉNITÉ : Opposabilité Conditionnelle de Preuves Censurées

MAKEU TENKU STELY BELVA

4 décembre 2025

Table des matières

1 Introduction **3**

2 Fondements théoriques **3**
2.1 Opposabilité conditionnelle 3
2.2 Objectifs du protocole 3
2.3 Notations mathématiques 3

3 Protocole SÉRÉNITÉ **4**
3.1 Acteurs 4
3.2 Structure du document publié 4
3.3 Formalisation mathématique 4
3.3.1 Étape 1 : Publication 4
3.3.2 Étape 2 : Révocation conditionnelle 5

4 Analyse de sécurité **5**
4.1 Confidentialité (C) 5
4.2 Fiabilité (R) 5
4.3 Opposabilité conditionnelle (O) 5

5 Cas d'usage **5**

6 Implémentation Python du protocole SÉRÉNITÉ **6**

7 Conclusion **8**

1 Introduction

Dans un contexte où la protection de la confidentialité des documents numériques est cruciale, le protocole **SÉRÉNITÉ** propose une solution basée sur le trilemme CRO :

- **C (Confidentialité)** : protéger les informations sensibles dans un document publié.
- **R (Fiabilité)** : garantir que le document original peut être reconstruit fidèlement.
- **O (Opposabilité conditionnelle)** : assurer la valeur juridique de la preuve uniquement après une procédure de vérification contrôlée.

Le protocole résout le problème de l'opposabilité juridique des preuves censurées tout en préservant la vie privée des parties.

2 Fondements théoriques

2.1 Opposabilité conditionnelle

Définition : Soit D_{Original} le document original et $D_{\text{Censuré}}$ le document publié après censure. L'opposabilité conditionnelle stipule que la valeur légale de $D_{\text{Censuré}}$ n'est établie que si la reconstruction produit le document original et satisfait l'égalité cryptographique :

$$H(D_{\text{Reconstruit}}) = H(D_{\text{Original}}) \tag{1}$$

où H est une fonction de hachage cryptographique sécurisée (SHA-3-256 par exemple).

2.2 Objectifs du protocole

- **Fiabilité (R) :** assurer que $H(D_{\text{Original}})$ n'est vérifiable que via une reconstruction réversible.
- **Confidentialité (C) :** masquer les données sensibles contenues dans $D_{\text{Censuré}}$.
- **Opposabilité (O) :** conditionnée par la vérification cryptographique de l'égalité $H(D_{\text{Reconstruit}}) = H(D_{\text{Original}})$.

2.3 Notations mathématiques

$$\begin{aligned}
D_{\text{Original}} &\in \{0,1\}^* \quad \text{(document complet)} \\
D_{\text{Censuré}} &\in \{0,1\}^* \quad \text{(document publié)} \\
H &: \{0,1\}^* \to \{0,1\}^{256} \quad \text{(fonction de hachage)} \\
Enc_K(M) &: \text{chiffrement symétrique sous clé } K \\
Dec_K(C) &: \text{déchiffrement correspondant} \\
\text{MerkleTree}(B_1, \ldots, B_m) &\to R \quad \text{(racine Merkle)} \\
SK_i &: \text{share } i \text{ dans un schéma Shamir } (t, n) \\
t, n &: \text{seuil et nombre total d'autorités} \\
\text{Sign}_{sk}(M) &: \text{signature numérique de } M \text{ avec clé privée } sk
\end{aligned}$$

3 Protocole SÉRÉNITÉ

3.1 Acteurs

— **Auteur** : crée D_{Original}.

— **Publieur** : publie $D_{\text{Censuré}}$.

— **Autorités de révocation** : n entités (ex. juges ou administrateurs) qui détiennent des parts du secret.

— **Vérificateur / Tribunal** : récupère les parts pour établir l'opposabilité.

3.2 Structure du document publié

Le document publié contient :

DCensuré = {blocs non sensibles, ciphertexts des blocs sensibles, Merkle root, meta chiffrée, signature}

3.3 Formalisation mathématique

3.3.1 Étape 1 : Publication

(a) Diviser D_{Original} en m blocs : $D_{\text{Original}} = \{B_1, \ldots, B_m\}$.

(b) Pour chaque bloc sensible B_j, générer une clé aléatoire K_j et calculer :

$$C_j = Enc_{K_j}(B_j) \tag{2}$$

(c) Remplacer les blocs sensibles par C_j et construire $D_{\text{Censuré}}$.

(d) Construire la racine Merkle :

$$R = \text{MerkleTree}(D_{\text{Censuré}}) \tag{3}$$

(e) Calculer le hachage original :

$$H_{\text{Original}} = H(D_{\text{Original}}) \tag{4}$$

(f) Préparer les métadonnées :

$$\text{meta} = \{H_{\text{Original}}, R, \text{timestamp}, \text{auteur}\} \tag{5}$$

(g) Générer une clé maître K_{meta} et chiffrer les métadonnées :

$$E_{\text{meta}} = Enc_{K_{\text{meta}}}(\text{meta}) \tag{6}$$

(h) Distribuer K_{meta} via un schéma Shamir (t, n) :

$$SK_i = \text{ShamirShare}_i(K_{\text{meta}}), \quad i = 1, \ldots, n \tag{7}$$

Chaque SK_i est chiffré sous la clé publique de l'autorité correspondante.

(i) Signer l'ensemble :

$$\sigma_{\text{author}} = \text{Sign}_{sk_{\text{author}}}(R || H(E_{\text{meta}})) \tag{8}$$

3.3.2 Étape 2 : Révocation conditionnelle

(a) Collecter au moins t parts SK_i signées par les autorités.

(b) Reconstruire la clé maître K_{meta} :

$$K_{\text{meta}} = \text{ShamirReconstruct}(SK_1, \ldots, SK_t) \tag{9}$$

(c) Déchiffrer les métadonnées :

$$\text{meta} = Dec_{K_{\text{meta}}}(E_{\text{meta}}) \tag{10}$$

(d) Récupérer les clés K_j des blocs sensibles et reconstruire $D_{\text{Reconstruit}}$:

$$D_{\text{Reconstruit}} = \bigcup_{j=1}^{m} B_j \tag{11}$$

(e) Vérifier l'égalité cryptographique :

$$H(D_{\text{Reconstruit}}) \stackrel{?}{=} H_{\text{Original}} \tag{12}$$

4 Analyse de sécurité

4.1 Confidentialité (C)

Les fragments sensibles sont chiffrés par des clés individuelles K_j ; la clé maître K_{meta} est distribuée via un schéma Shamir (t, n). Si moins de t autorités coopèrent, les blocs sensibles restent confidentiels.

4.2 Fiabilité (R)

La reconstruction complète de D_{Original} est garantie par :

$$H(D_{\text{Reconstruit}}) = H(D_{\text{Original}}) \tag{13}$$

et par la signature de l'auteur sur la racine Merkle et le hash des métadonnées.

4.3 Opposabilité conditionnelle (O)

L'opposabilité légale n'est établie que si le processus de reconstruction est réalisé avec succès et que l'égalité $H(D_{\text{Reconstruit}}) = H(D_{\text{Original}})$ est vérifiée.

5 Cas d'usage

— **Journalisme d'investigation** : publication de documents censurés protégeant les sources.

— **Lanceurs d'alerte** : divulgation partielle sécurisée, reconstruction possible sous contrôle légal.

— **Archivage légal** : documents accessibles sous conditions strictes pour vérification.

6 Implémentation Python du protocole SÉRÉNITÉ

Le protocole **SÉRÉNITÉ** peut être simulé avec Python pour gérer le chiffrement des blocs sensibles, le Merkle Tree, le partage de la clé maître via Shamir et la reconstruction conditionnelle.

```
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric.ed25519 import
   Ed25519PrivateKey
from secretsharing import SecretSharer
import merkletools
import os
import json
import base64
import hashlib

# Fonctions utilitaires
def hash_data(data: bytes) -> str:
    digest = hashlib.sha3_256(data).hexdigest()
    return digest

def aes_encrypt(key: bytes, plaintext: bytes) -> bytes:
    aesgcm = AESGCM(key)
    nonce = os.urandom(12)
    ct = aesgcm.encrypt(nonce, plaintext, None)
    return nonce + ct

def aes_decrypt(key: bytes, ciphertext: bytes) -> bytes:
    nonce, ct = ciphertext[:12], ciphertext[12:]
    aesgcm = AESGCM(key)
    return aesgcm.decrypt(nonce, ct, None)

# Cr ation d'un document exemple
DOriginal = [
    {"index":0, "content": b"Texte public 1", "sensitive": False},
    {"index":1, "content": b"Nom secret", "sensitive": True},
    {"index":2, "content": b"Texte public 2", "sensitive": False},
    {"index":3, "content": b"Num ro confidentiel", "sensitive":
       True},
]

# Chiffrement des blocs sensibles
DCensur  = []
secret_keys_map = {}
for block in DOriginal:
    if block["sensitive"]:
        K = AESGCM.generate_key(bit_length=256)
        C = aes_encrypt(K, block["content"])
        DCensur .append({"index": block["index"], "content":
           base64.b64encode(C).decode(), "placeholder": True})
        secret_keys_map[block["index"]] = base64.b64encode(K).
```

```
            decode()
    else:
        DCensur .append({"index": block["index"], "content": block
            ["content"].decode(), "placeholder": False})

# Cr ation du Merkle Tree
mt = merkletools.MerkleTools(hash_type="sha3_256")
for block in DCensur :
    mt.add_leaf(block["content"], True)
mt.make_tree()
M_root_cens = mt.get_merkle_root()

# Hachage du document original
doc_bytes = b"".join([block["content"] if not block["sensitive"]
    else base64.b64decode(aes_encrypt(AESGCM.generate_key(bit_length
    =256), block["content"])) for block in DOriginal])
HOriginal = hash_data(doc_bytes)

# M tadonn es et cl  ma tre
meta = {"HOriginal": HOriginal, "MerkleRoot": M_root_cens, "author"
    : "Belva Makeu"}
K_meta = AESGCM.generate_key(bit_length=256)
meta_json = json.dumps(meta).encode()
E_meta = aes_encrypt(K_meta, meta_json)

# Shamir Secret Sharing
n, t = 5, 3
hex_key = K_meta.hex()
shares = SecretSharer.split_secret(hex_key, t, n)

# Signature Ed25519
sk = Ed25519PrivateKey.generate()
sig = sk.sign(E_meta)

# Publication
publication = {
    "DCensur ": DCensur ,
    "MerkleRoot": M_root_cens,
    "E_meta": base64.b64encode(E_meta).decode(),
    "Shares": shares,
    "Signature": base64.b64encode(sig).decode()
}
print(json.dumps(publication, indent=4))

# Reconstruction conditionnelle
selected_shares = shares[:t]
reconstructed_hex = SecretSharer.recover_secret(selected_shares)
K_meta_reconstructed = bytes.fromhex(reconstructed_hex)
meta_decrypted = aes_decrypt(K_meta_reconstructed, base64.b64decode
    (publication["E_meta"]))
meta_final = json.loads(meta_decrypted)
```

```
print(meta_final)
```

Listing 1 – Implémentation Python du protocole SÉRÉNITÉ

7 Conclusion

Le protocole **SÉRÉNITÉ** est une solution innovante permettant la publication de documents censurés tout en garantissant une opposabilité conditionnelle. Le trilemme CRO est respecté grâce à :

— Le chiffrement des données sensibles pour la confidentialité (C),

— La reconstruction réversible pour la fiabilité (R),

— La dépendance à la révocation par un seuil d'autorités pour l'opposabilité (O).