MÁSTER INTERUNIVERSITARIO EN CIBERSEGURIDAD

Information Security
Lab Assignment II
Verifiable Secret Sharing (VSS)

Aarón Cela Riveiro

In this lab assignment, I implemented a **Verifiable Secret Sharing (VSS)** scheme using **KZG Polynomial Commitments**. The system allows a dealer to distribute a secret among n participants with a reconstruction threshold t, so that any group of at least $t + 1$ users can recover the secret, while smaller groups cannot. The scheme is based on polynomial sharing and includes cryptographic verification of each share through commitment proofs.

The project is organized in two Python programs: `dealer.py`, responsible for generating the shares and polynomial commitment, and `client.py`, which verifies and reconstructs the original secret. The implementation uses the `ckzg` library to generate polynomial commitments and proofs of evaluation using the KZG scheme. Share data is exchanged via a shared file named `shares.txt`.
Full implementation available at https://github.com/ronezz/SI/tree/main/LAB2.

File: dealer.py

This program performs the Sharing phase of the VSS. It builds a random polynomial of degree t with constant term equal to the secret, commits to the polynomial using KZG, produces per-index evaluation proofs, verifies them locally, and stores the shares (index, value, proof) in a text file along with the commitment.

1. Configuration and Field Primitives

```python
import sys, hashlib, secrets, ckzg

SHARES_FILE = "shares.txt"
DEFAULT_SETUP_PATH = "trusted_setup.txt"
FR_MOD = int("73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff00000001
    ", 16)
BYTES_PER_FE = 32
FIELD_ELEMENTS_PER_BLOB = 4096
PRIMITIVE_ROOT_OF_UNITY = 7

# Finite field
def fr(x): return x % FR_MOD
def int_to_fe_bytes(x): return fr(x).to_bytes(BYTES_PER_FE, "big")
```

Listing 1: Constants and field helpers

2. Polynomial Construction and Evaluation

A random polynomial $\varphi(x)$ of degree t is generated so that $\varphi(0) = s$. A Horner evaluation is used for efficiency.

```python
# Polynomial operations
def random_poly_with_secret(s, t):
    return [fr(s)] + [secrets.randbelow(FR_MOD) for _ in range(t)]

def poly_eval(coeffs, x):
    y = 0
    for c in reversed(coeffs):
        y = (y * x + c) % FR_MOD
    return y
```

Listing 2: Random polynomial with secret and evaluation

3. Domain Evaluation to Blob (KZG Input)

The polynomial is evaluated over a roots-of-unity domain to create a blob compatible with KZG commitments.

```python
# Domain evaluation -> blob for KZG
def _compute_roots_of_unity(n):
    assert (FR_MOD - 1) % n == 0
    w = pow(PRIMITIVE_ROOT_OF_UNITY, (FR_MOD - 1) // n, FR_MOD)
    roots = [1]
    for _ in range(1, n):
        roots.append((roots[-1] * w) % FR_MOD)
    return roots

def _bit_reverse(seq):
    n, w = len(seq), (len(seq) - 1).bit_length()
    def rb(k): return int(f"{k:0{w}b}"[::-1], 2)
    return [seq[rb(i)] for i in range(n)]

def coeffs_to_blob(coeffs):
    domain = _compute_roots_of_unity(FIELD_ELEMENTS_PER_BLOB)
    evals = [poly_eval(coeffs, x) for x in domain]
    evals = _bit_reverse(evals)
    return b"".join(int_to_fe_bytes(e) for e in evals)
```

Listing 3: Roots of unity and blob conversion

4. Secret Encoding and Shares File Writer

```python
# Secret from input string
def secret_from_string(text):
    return int.from_bytes(hashlib.sha256(text.encode()).digest(), "big") %
        FR_MOD

# Write shares to file
def save_shares_txt(commitment_hex, t, rows):
    with open(SHARES_FILE, "w") as f:
        f.write(f"commitment={commitment_hex}\n")
        f.write(f"threshold={t}\n")
        for i, y, proof in rows:
            f.write(f"i={i} y=0x{y.hex()} proof=0x{proof.hex()}\n")
```

Listing 4: Hash-to-field and shares writer

5. Sharing Phase: Commitment, Witnesses, and Output

```python
# Sharing...
def main():
    print("[Dealer] Sharing secret...")
    if len(sys.argv) != 4:
        print("Usage: python dealer.py N T SECRET")
        sys.exit(1)

    n = int(sys.argv[1]); t = int(sys.argv[2]); secret_text = sys.argv[3]
    assert n > t >= 0
    s = secret_from_string(secret_text)
    coeffs = random_poly_with_secret(s, t)
    setup = ckzg.load_trusted_setup(DEFAULT_SETUP_PATH, 0)
    blob = coeffs_to_blob(coeffs)

    # Commit
    commitment = ckzg.blob_to_kzg_commitment(blob, setup)
    commitment_hex = "0x" + commitment.hex()

    rows = []
    for i in range(1, n + 1):
        z = int_to_fe_bytes(i)

        # CreateWitness
        proof, y = ckzg.compute_kzg_proof(blob, z, setup)

        # VerifyEval
        if not ckzg.verify_kzg_proof(commitment, z, y, proof, setup):
            raise RuntimeError(f"Invalid proof at share {i}")

        rows.append((i, y, proof))

    save_shares_txt(commitment_hex, t, rows)
    print(f"[Dealer] {n} shares saved to {SHARES_FILE}")
    print(f"[Dealer] threshold t={t}")
    print(f"[Dealer] shared secret hash={s}")
```

Listing 5: Main sharing logic: commit and generate verified shares

File: `client.py`

This program performs the Reconstruction phase. It parses the shares file, verifies each share against the KZG commitment, keeps only valid shares, and uses Lagrange interpolation at $x = 0$ to recover the secret with any $t + 1$ valid points.

1. Configuration and Helpers

```python
import sys, ckzg

SHARES_FILE = "shares.txt"
DEFAULT_SETUP_PATH = "trusted_setup.txt"
FR_MOD = int("73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff00000001", 16)
BYTES_PER_FE = 32

# Field conversion helpers
def fr(x): return x % FR_MOD
def int_to_fe_bytes(x): return fr(x).to_bytes(BYTES_PER_FE, "big")
def fe_bytes_to_int(b): return int.from_bytes(b, "big") % FR_MOD
```

Listing 6: Constants and field conversion helpers

2. Lagrange Interpolation at Zero

```python
# Lagrange interpolation
def lagrange_at_zero(points):
    s = 0
    for i, (x_i, y_i) in enumerate(points):
        num, den = 1, 1
        for j, (x_j, _) in enumerate(points):
            if i == j:
                continue
            num = (num * (-x_j % FR_MOD)) % FR_MOD
            den = (den * ((x_i - x_j) % FR_MOD)) % FR_MOD
        li0 = num * pow(den, FR_MOD - 2, FR_MOD) % FR_MOD
        s = (s + y_i * li0) % FR_MOD
    return s
```

Listing 7: Compute $\phi(0)$ from t+1 valid points

3. Parse Shares File

```python
# Load shares
def load_shares_txt():
    with open(SHARES_FILE) as f:
        lines = [l.strip() for l in f if l.strip()]
    commitment_hex = lines[0].split("=")[1]
    t = int(lines[1].split("=")[1])
    shares = []
    for line in lines[2:]:
        parts = dict(x.split("=") for x in line.split())
        shares.append((
            int(parts["i"]),
            bytes.fromhex(parts["y"][2:]),
            bytes.fromhex(parts["proof"][2:])
        ))
    return commitment_hex, t, shares
```

Listing 8: Read commitment, threshold, and rows from `shares.txt`

4. Reconstruction Phase: Verify and Interpolate

```python
# Reconstruction
def main():
    if len(sys.argv) != 1:
        print("Usage: python client.py")
        sys.exit(1)


    commitment_hex, t, shares = load_shares_txt()
    print(f"[Client] Reconstructing the secret with {t} shares...")
    setup = ckzg.load_trusted_setup(DEFAULT_SETUP_PATH, 0)
    commitment = bytes.fromhex(commitment_hex[2:])

    good = []
    for i, y, proof in shares:
        z = int_to_fe_bytes(i)

        # VerifyEval
        if ckzg.verify_kzg_proof(commitment, z, y, proof, setup):
            good.append((i, fe_bytes_to_int(y)))

    if len(good) < t + 1:
        raise RuntimeError(f"Not enough valid shares: {len(good)} < {t+1}")

    good.sort()
    used = good[:t + 1]

    # Interpolation
    secret = lagrange_at_zero(used)

    print(f"[Client] reconstructed secret = {secret}")
    print(f"[Client] t={t} shares_usadas={len(used)} idx={[i for i,_ in used]}")
```

Listing 9: Verify KZG proofs and reconstruct the secret

Shares File Format (`shares.txt`)

The dealer stores the output in a plain text file. The first two lines contain the KZG commitment (hex) and the threshold t. Each subsequent line contains the tuple $\langle i, \varphi(i), w_i \rangle$ encoded as index, evaluation bytes, and the evaluation proof.

```
commitment=0x<commitment-hex>
threshold=t
i=1 y=0x<32-byte-enc> proof=0x<proof-hex>
i=2 y=0x<32-byte-enc> proof=0x<proof-hex>
...
```

Listing 10: Example layout of `shares.txt`

Tests and Validation

This section presents a series of tests carried out to verify the correct behaviour of the implemented Verifiable Secret Sharing (VSS) scheme. The objective is to validate the correctness of the sharing and reconstruction phases, as well as the system's robustness in the presence of corrupted or malicious participants (Byzantine faults).

Test 1: Basic secret sharing and reconstruction

In this initial test, the dealer distributes a secret among 5 participants with threshold 2, and the client attempts to reconstruct it using the shares written to `shares.txt`. No tampering is applied in this case.

```
PS C:\Users\aaron\Documents\MUNICS2\SI\Practicas\LAB-2\p-2> python dealer.py 5 2 "Testing our VSS practice..."
[Dealer] Sharing secret...
[Dealer] 5 shares saved to shares.txt
[Dealer] threshold t=2
[Dealer] shared secret hash=3274120820012603764869263942926919353940716031661395115415661777999191324190
PS C:\Users\aaron\Documents\MUNICS2\SI\Practicas\LAB-2\p-2> python .\client.py
[Client] Reconstructing the secret with 2 shares...
[Client] reconstructed secret = 3274120820012603764869263942926919353940716031661395115415661777999191324190
[Client] t=2 shares_usadas=3 idx=[1, 2, 3]
PS C:\Users\aaron\Documents\MUNICS2\SI\Practicas\LAB-2\p-2>
```

Figure 1: Correct sharing and reconstruction with valid shares

Test 2: Detection of corrupted shares

In this test, we simulate the presence of Byzantine participants by manually modifying the value of some shares in `shares.txt`. The integrity of each share is validated using `VerifyEval` against the polynomial commitment generated by the dealer.

```
shares.txt
commitment=0x9745bd855bd13d2263b4ff878e2ce127ccc5ac2770ceb593b05d5561e13c94bee6c521fe5c6ac710c66c99f04c6df429
threshold=2
i=1 y=0x1937da9bacb7c732fe14996243a6f6c5408806f46dfe0155d745369ce305c8e7 proof=0x82c524a886533064dc931fc783ef168b6d5ab0587d648beabecbf4c2ae8e7db1561e54a6b1160be5
i=2 y=0x567246f681709c94899a415d71232b67f59b5b4707dac45c7b53a0eaebf11174 proof=0xa679dcae04f6cfbade4c4271f2e0155370399a1be869c604c4b57fca9db3eff4b5de5a15e7605ad5
i=3 y=0x2a1479ea59eab1f4a61e9fc0a8f4385b1cad8a295a87c8db2ad54437a7e8ca68 proof=0x8d6b5cbe04331b2b1e9d3b2dd4633d758cb1b1204e8df2c718d4885b0321408889bdf4a8fcf344e1
i=4 y=0x080c1aca5fc3849b86db8c93f4bbf5a4097c379e66036ad0e5ca208216ecf3c4 proof=0xb44d3c90f6fecd693d13aff137d59a50673de7cd0f8573901463beec943d0c4e326e9b40a1988fa
i=5 y=0x6446d0e9bc9891d15f0adfdf5e1c3b480fc507a92a4c063cac3235c938fd8d89 proof=0xa607fbc08630f0736d79d5764dd21c09494aa166ccf4241b6184a41c2edd0f5e4a8118a9566a2c1a

PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL   powershell

PS C:\Users\aaron\Documents\MUNICS2\SI\Practicas\LAB-2\p-2> python dealer.py 5 2 "Testing our VSS system with corrupting nodes..."
[Dealer] Sharing secret...
[Dealer] 5 shares saved to shares.txt
[Dealer] threshold t=2
[Dealer] shared secret hash=40822142169232086648031044482557906626927313075957419506785236993016995442883
PS C:\Users\aaron\Documents\MUNICS2\SI\Practicas\LAB-2\p-2>
```

Figure 2: Initial dealer's share

The client identifies the modified shares as invalid and discards them during verification. Since at least $t + 1$ valid shares remain available, the reconstruction still succeeds. Note that we manually tampered with the last hexadecimal digit of the first two shares to simulate malicious behaviour.

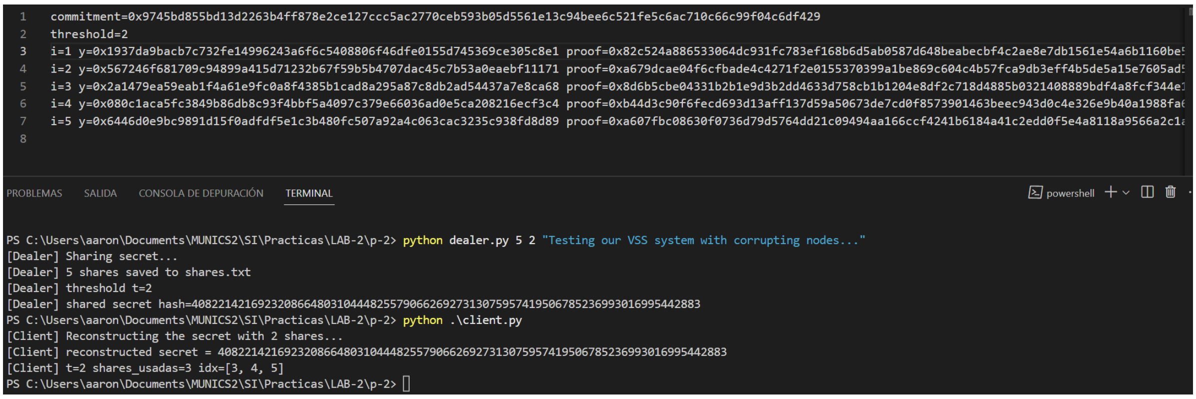

Figure 3: Invalid shares discarded; secret successfully recovered

Test 3: Threshold enforcement

In this test, we evaluate the threshold property by progressively removing valid shares. When exactly $t + 1$ valid shares are available, reconstruction remains possible.

Figure 4: Reconstruction with exactly $t + 1$ valid shares

However, when additional shares are removed or corrupted so that fewer than $t+1$ valid shares remain, the reconstruction process fails as expected.

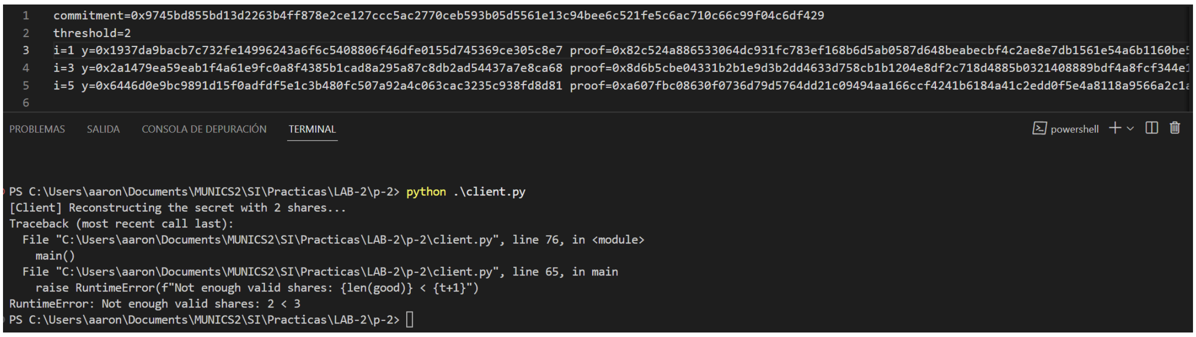

Figure 5: Reconstruction failure when fewer than $t+1$ valid shares remain