Example 01

BLS Signature (SageMath code)

```
# 1. Use a smaller pairing-friendly curve for demonstration
p =
0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9feffffff
fffaaab
F = GF(p)
E = EllipticCurve(F, [0, 4])  # y² = x³ + 4 (BLS12-381 curve parameters)

# 2. Find a valid generator point (this works for BLS12-381)
Gx =
0x17F1D3A73197D7942695638C4FA9AC0FC3688C4F9774B905A14E3A3F171BAC586C55E83FF97A1AEFF
B3AF00ADB22C6BB
Gy =
0x08B3F481E3AAA0F1A09E30ED741D8AE4FCF5E095D5D00AF600DB18CB2C04B3EDD03CC744A2888AE
40CAA232946C5E7E1
G = E(Gx, Gy)

# 3. Generate keys
print("\nGenerating keys...")
order = G.order()
sk = ZZ.random_element(order)  # Private key
pk = sk * G                    # Public key
print(f"Private key: {sk}")
print(f"Public key: {pk}")

# 4. Hash message to curve (simplified)
print("\nSigning message...")
message = "Hello"
h = int(hashlib.sha256(message.encode()).hexdigest(), 16) % order
H = h * G
signature = sk * H
print(f"Signature: {signature}")

# 5. Verification using Weil pairing
print("\nVerifying...")
lhs = G.weil_pairing(signature, order)
rhs = pk.weil_pairing(H, order)
print(f"Verification {'passed' if lhs == rhs else 'failed'}")

print("\nDone!")
```

```
Generating keys...
Private key: 41774967760637195051855541726800770157152756773339356030013515528590704449461
Public key: (1988004570477638025749209290148942951486990970443593902656779027940171459133257663289868553513468885494110790828009 : 17
29325471966442529858459352629108848378753048741362578895161405415264813223073934324717384998096857394687185896963 : 1)

Signing message...
Signature: (3634372867025134779537492762017842587895103140040590366827922268729919685175689481154386115491074391834038984755437 : 703
4244807966585747925829746149927016897009525961190331630943618598208570708610402842966366681474008906950392669858 : 1)

Verifying...
Verification passed

Done!
```

Example 02

Tripartite Diffie-Hellman key exchange

```
# 1. Curve Setup (Supersingular curve for pairing-friendly properties)
p = 103  # Prime ≡ 3 mod 4
F = GF(p)
E = EllipticCurve(F, [1, 0])  # y² = x³ + x
print(f"Curve order: {E.order()}")

# 2. Find prime subgroup
r = 13  # Subgroup order
cofactor = E.order() // r

# 3. Generate G1 (on base curve)
G1 = E(0)
while G1 == E(0):
   P = E.random_point()
   G1 = cofactor * P
assert r*G1 == E(0)
print(f"G1: {G1}")

# 4. Create extension field and curve
R.<x> = PolynomialRing(F)
F2.<i> = GF(p^2, modulus=x^2 + 1)
E2 = EllipticCurve(F2, [1, 0])  # Same curve over Fp²

# 5. Generate G2 (must be on E2)
G2 = E2(0)
while G2 == E2(0):
   P = E2.random_point()
   G2 = cofactor * P
assert r*G2 == E2(0)
print(f"G2: {G2}")

# 6. Tripartite DH Protocol
```

```
print("\nTripartite Diffie-Hellman:")

# Private keys
a = ZZ.random_element(r)
b = ZZ.random_element(r)
c = ZZ.random_element(r)
print(f"Private keys: a={a}, b={b}, c={c}")

# Public values (all points on E2 for pairing)
aG1 = a * E2(G1)  # Convert G1 to E2
bG1 = b * E2(G1)
cG1 = c * E2(G1)
aG2 = a * G2
bG2 = b * G2
cG2 = c * G2

# Compute shared keys
print("\nComputing shared keys...")
try:
    # Alice: e(bG1, cG2)^a
    alice_shared = bG1.weil_pairing(cG2, r)^a

    # Bob: e(aG1, cG2)^b
    bob_shared = aG1.weil_pairing(cG2, r)^b

    # Carol: e(aG1, bG2)^c
    carol_shared = aG1.weil_pairing(bG2, r)^c

    # Verification
    print(f"Alice's key: {alice_shared}")
    print(f"Bob's key: {bob_shared}")
    print(f"Carol's key: {carol_shared}")
    print(f"\nAll match: {alice_shared == bob_shared == carol_shared}")

    # True shared secret
    shared_secret = E2(G1).weil_pairing(G2, r)^(a*b*c)
    print(f"True secret: {shared_secret}")

except Exception as e:
    print(f"Error: {str(e)}")
    print("Note: For better performance, use optimized pairing libraries")
```

```
.....
Curve order: 104
G1: (18 : 44 : 1)
G2: (9*i + 15 : 101*i + 26 : 1)

Tripartite Diffie-Hellman:
Private keys: a=10, b=8, c=10

Computing shared keys...
Alice's key: 25*i + 71
Bob's key: 25*i + 71
Carol's key: 25*i + 71

All match: True
True secret: 25*i + 71
sage:
```

Example 03

Zero-Knowledge Proof of the quadratic equation

```
# 1. Curve Setup (same as working tripartite DH example)
p = 103  # Prime ≡ 3 mod 4
F = GF(p)
E = EllipticCurve(F, [1, 0])  # y² = x³ + x
print(f"Curve order: {E.order()}")

# 2. Find prime subgroup
r = 13  # Subgroup order
cofactor = E.order() // r

# 3. Generate G1 (on base curve)
G1 = E(0)
while G1 == E(0):
   P = E.random_point()
   G1 = cofactor * P
assert r*G1 == E(0)
print(f"G1: {G1}")

# 4. Create extension field and curve
R.<x> = PolynomialRing(F)
F2.<i> = GF(p^2, modulus=x^2 + 1)
E2 = EllipticCurve(F2, [1, 0])  # Same curve over Fp²
```

```
# 5. Generate G2 (on extended curve)
G2 = E2(0)
while G2 == E2(0):
    P = E2.random_point()
    G2 = cofactor * P
assert r*G2 == E2(0)
print(f"G2: {G2}")

# 6. Pairing function
def pairing(P, Q):
    return P.weil_pairing(Q, r)

# 7. Zero-Knowledge Proof for x^2 - x - 42 = 0

# Prover knows x = 7 (solution)
x = 7
assert x^2 - x - 42 == 0

# Convert G1 to E2 for pairing operations
G1_E2 = E2(G1)

# Prover computes commitments
xG1 = x * G1_E2  # Commitment in E2
xG2 = x * G2     # Proof in E2

# Verifier checks the equation in the exponent:
# e(xG1, xG2) * e(-G1, xG2) * e(-42*G1, G2) == 1

term1 = pairing(xG1, xG2)
term2 = pairing(-G1_E2, xG2)
term3 = pairing(-42*G1_E2, G2)

lhs = term1 * term2 * term3

print("\nZero-Knowledge Proof Verification:")
print(f"Pairing products: {term1} * {term2} * {term3} = {lhs}")
print("Proof valid (==1):", lhs == F2(1))

# Test with wrong value
x_wrong = 5
xG1_wrong = x_wrong * G1_E2
xG2_wrong = x_wrong * G2
term1_wrong = pairing(xG1_wrong, xG2_wrong)
term2_wrong = pairing(-G1_E2, xG2_wrong)
```

```
term3_wrong = pairing(-42*G1_E2, G2)

lhs_wrong = term1_wrong * term2_wrong * term3_wrong
print("\nTest with wrong x=5:")
print("Verification (should not be 1):", lhs_wrong == F2(1))
```

```
Curve order: 104
G1: (32 : 56 : 1)
G2: (8*i : 56*i + 47 : 1)

Zero-Knowledge Proof Verification:
Pairing products: 45*i + 6 * 78*i + 71 * 45*i + 6 = 1
Proof valid (==1): True

Test with wrong x=5:
Verification (should not be 1): False
sage:
```