

# The Alberta Buck - Identity - Cryptography Example (v1.0)

Perry Kundert

2026-04-10



This document has two parts. First, a data flow analysis shows where each piece of identity data resides – Ethereum contract storage, Holochain DHT, or local wallet – at each step of the protocol, using mermaid sequence diagrams to trace registration, approval, transfer, and regulatory compliance flows. Second, a complete worked cryptographic example implements every step in Python on the `alt_bn128` (BN254) curve via `py_ecc`, with counter-examples demonstrating what each verification check looks like when it fails.

The scenario is a transfer from Private Alice to Public Bob (an AMM pool). Alice obtains a Pointcheval-Sanders signed identity from an issuer, derives a fresh unlinkable credential via the Identity Fountain, registers it on-chain (~235K gas NIZK verification), and re-encrypts her identity for Bob with a Chaum-Pedersen proof of correct re-encryption (~29K gas). Bob verifies Alice's identity using the two-channel design: the on-chain ElGamal ciphertext provides cryptographic binding, while Holochain delivers the plaintext `identity_data`. The AMM pool scenario demonstrates how this proof chain protects Bob under subpoena – even 18 months and 500,000 transactions later – and why Alice cannot hide from regulatory recovery despite the Identity Fountain's unlinkability guarantees. (PDF, Text)

# Contents

|           |  |           |
|-----------|--|-----------|
| <b>1</b>  | <b>Data Flow: Where Everything Lives</b>                             | <b>4</b>  |
| 1.1       | Data Residency . . . . .   | 4         |
| 1.2       | Registration: Credential Goes On-Chain . . . . .                     | 4         |
| 1.3       | Approve: Re-Encryption with Trust Verification . . . . .             | 5         |
| 1.4       | Transfer: Everything On-Chain . . . . .                              | 6         |
| 1.5       | Two-Channel Verification and Malfeasance Detection . . . . .         | 7         |
| 1.6       | The Four Transfer Modes . . . . .                                    | 8         |
| 1.7       | The AMM Pool Scenario: Retroactive Identity Recovery . . . . .       | 9         |
| <b>2</b>  | <b>Setup: The alt_bn128 Curve and Helpers</b>                        | <b>13</b> |
| <b>3</b>  | <b>Issuer Key Generation</b>   | <b>14</b> |
| 3.1       | Same Operation via the Wallet API . . . . .                          | 14        |
| <b>4</b>  | <b>KYC Ceremony – Issuer Signs Alice’s Identity</b>                  | <b>15</b> |
| 4.1       | Alice’s Plaintext Identity . . . . .                                 | 15        |
| 4.2       | Compute Identity Scalar and PS Signature . . . . .                   | 15        |
| 4.3       | Verify the PS Signature (Issuer Self-Check) . . . . .                | 16        |
| 4.4       | Same Operation via the Wallet API . . . . .                          | 17        |
| <b>5</b>  | <b>The Identity Fountain – Alice Derives a Fresh Credential</b>      | <b>17</b> |
| 5.1       | Rerandomize the PS Signature . . . . .                               | 17        |
| 5.2       | Generate Fresh Identity Key Pair and ElGamal Encryption . . . . .    | 18        |
| 5.3       | NIZK Proof: Binding PS Signature to ElGamal Ciphertext . . . . .     | 19        |
| <b>6</b>  | <b>Registration – On-Chain Verification</b>                          | <b>20</b> |
| 6.1       | Verify the NIZK Proof (What the Contract Computes) . . . . .         | 20        |
| 6.2       | Same Operation via the Wallet API . . . . .                          | 23        |
| <b>7</b>  | <b>Bob Binds His Pool as a Public-Identity Contract</b>              | <b>23</b> |
| 7.1       | Same Operation via the Wallet API . . . . .                          | 24        |
| <b>8</b>  | <b>Alice Approves Bob – Re-Encryption + Chaum-Pedersen Proof</b>     | <b>25</b> |
| 8.1       | Re-Encrypt Alice’s Identity for Bob . . . . .                        | 25        |
| 8.2       | Chaum-Pedersen Proof of Correct Re-Encryption . . . . .              | 26        |
| 8.3       | Verify Chaum-Pedersen Proof (What the Contract Computes) . . . . .   | 27        |
| 8.4       | Same Operation via the Wallet API . . . . .                          | 28        |
| <b>9</b>  | <b>Transfer – Bilateral Identity Check</b>                           | <b>29</b> |
| <b>10</b> | <b>Bob Verifies Alice’s Identity (Two-Channel Design)</b>            | <b>30</b> |
| 10.1      | Counter-example: Tampered Identity Data . . . . .                    | 31        |
| <b>11</b> | <b>Receipt Construction</b>  | <b>31</b> |
| <b>12</b> | <b>Summary: What Each Party Knows</b>                                | <b>32</b> |
| <b>13</b> | <b>Appendix: The Identity Fountain – Unlinkability Demonstration</b> | <b>32</b> |

|  |           |
|--|-----------|
| <b>14 Appendix: Small-Group Proofs</b>                           | <b>34</b> |
| 14.1 Setup and ElGamal . . . . .                                 | 34        |
| 14.2 NIZK: Proof of Knowledge (Registration) . . . . .           | 35        |
| 14.3 Chaum-Pedersen: Proof of Same Plaintext (Approve) . . . . . | 36        |

# 1 Data Flow: Where Everything Lives

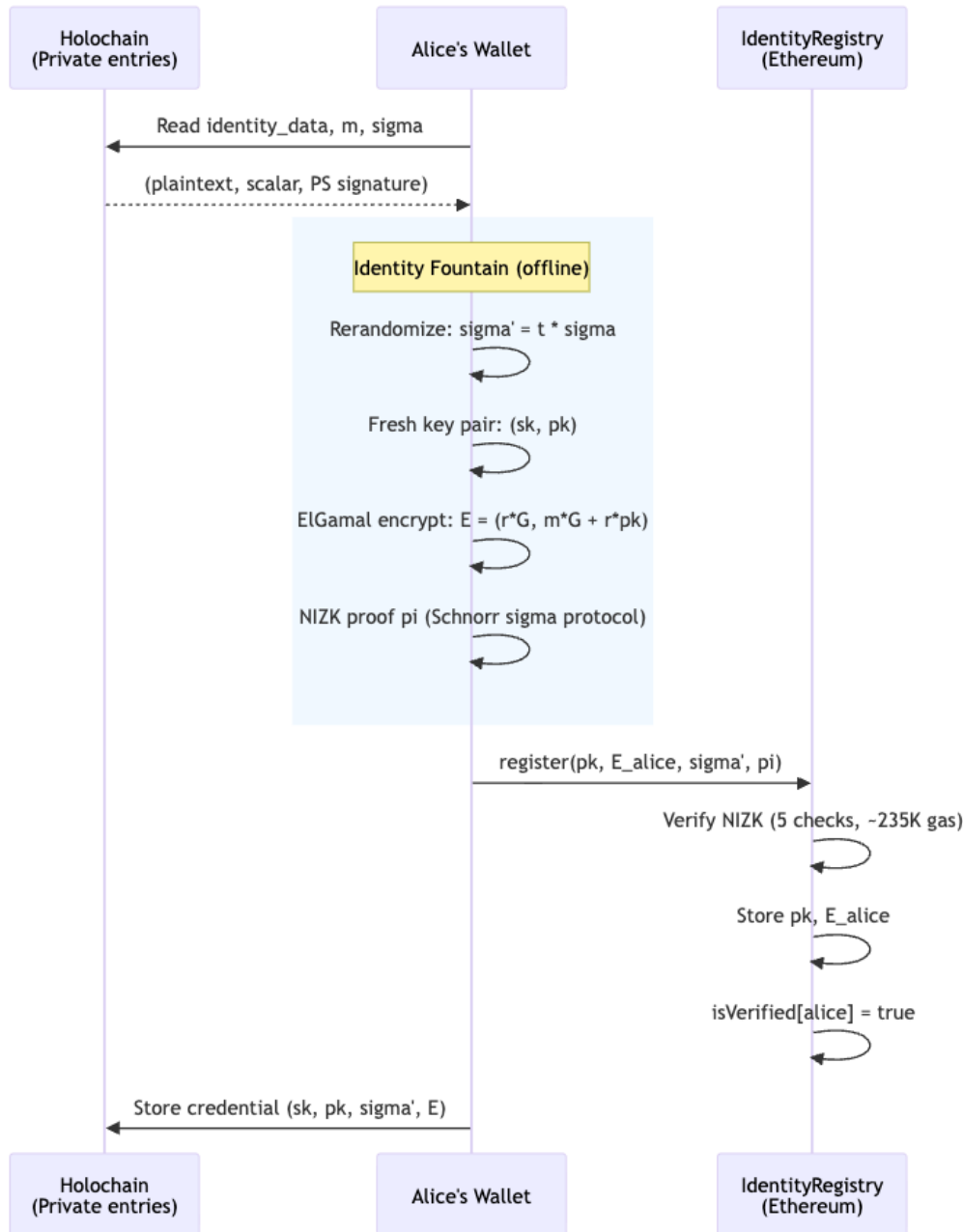
The Alberta Buck identity system spans two networks (Ethereum and Holochain) and a local device (the wallet). The tables and mermaid sequence diagrams below map every piece of identity data to its storage location, visibility, and the protocol step that consumes it – tracing flows through registration, approval, transfer, and two-channel verification, including the trust checkpoint where wallets detect malfeasance, and the AMM pool scenario where a public pool operator responds to a regulatory subpoena months after a swap. The key result: all cryptographic proofs are permanently on-chain, so regulatory compliance requires only Bob’s secret key and an archival Ethereum node – not Alice’s cooperation.

## 1.1 Data Residency

| Data                                  | Location                 | Visibility        | When Used                            |
|---------------------------------------|--------------------------|-------------------|--------------------------------------|
| identity_data = plaintext             | Holochain Private entry  | Owner only        | Approve (sent to counterparty)       |
| $m = H(\text{identity\_data})$        | Holochain Private entry  | Owner only        | Registration, Approve                |
| PS signature $\sigma$                 | Holochain Private entry  | Owner only        | Registration (rerandomized)          |
| Secret key $sk$                       | Holochain Private entry  | Owner only        | Approve (decrypt + re-encrypt)       |
| Rerandomized $\sigma'$                | IdentityRegistry         | Public (on-chain) | Registration verification            |
| Public key $pk$                       | IdentityRegistry         | Public (on-chain) | Approve, Transfer                    |
| ElGamal ciphertext $E_{\text{alice}}$ | IdentityRegistry         | Public (on-chain) | Approve (read by contract)           |
| NIZK proof $\pi$                      | IdentityRegistry (tx)    | Public (on-chain) | Registration verification            |
| Re-encrypted $E_{\text{bob}}$         | BUCK ERC-20 (approve)    | Public (on-chain) | Two-channel verification             |
| Chaum-Pedersen proof                  | BUCK ERC-20 (approve tx) | Public (on-chain) | Approve verification                 |
| $\text{\_receiptFragments[a][b]}$     | BUCK ERC-20 storage      | Public (on-chain) | Transfer (bilateral check)           |
| isPublicIdentity flag                 | IdentityRegistry         | Public (on-chain) | Transfer (receipt-fragment fallback) |
| Public identity_data                  | IdentityRegistry or DHT  | Public            | Anyone (no approve needed)           |
| Holochain Warrants                    | Holochain DHT            | Public            | Malfeasance detection                |
| Canonical receipt                     | Holochain Private entry  | Each party only   | Dispute resolution                   |

## 1.2 Registration: Credential Goes On-Chain

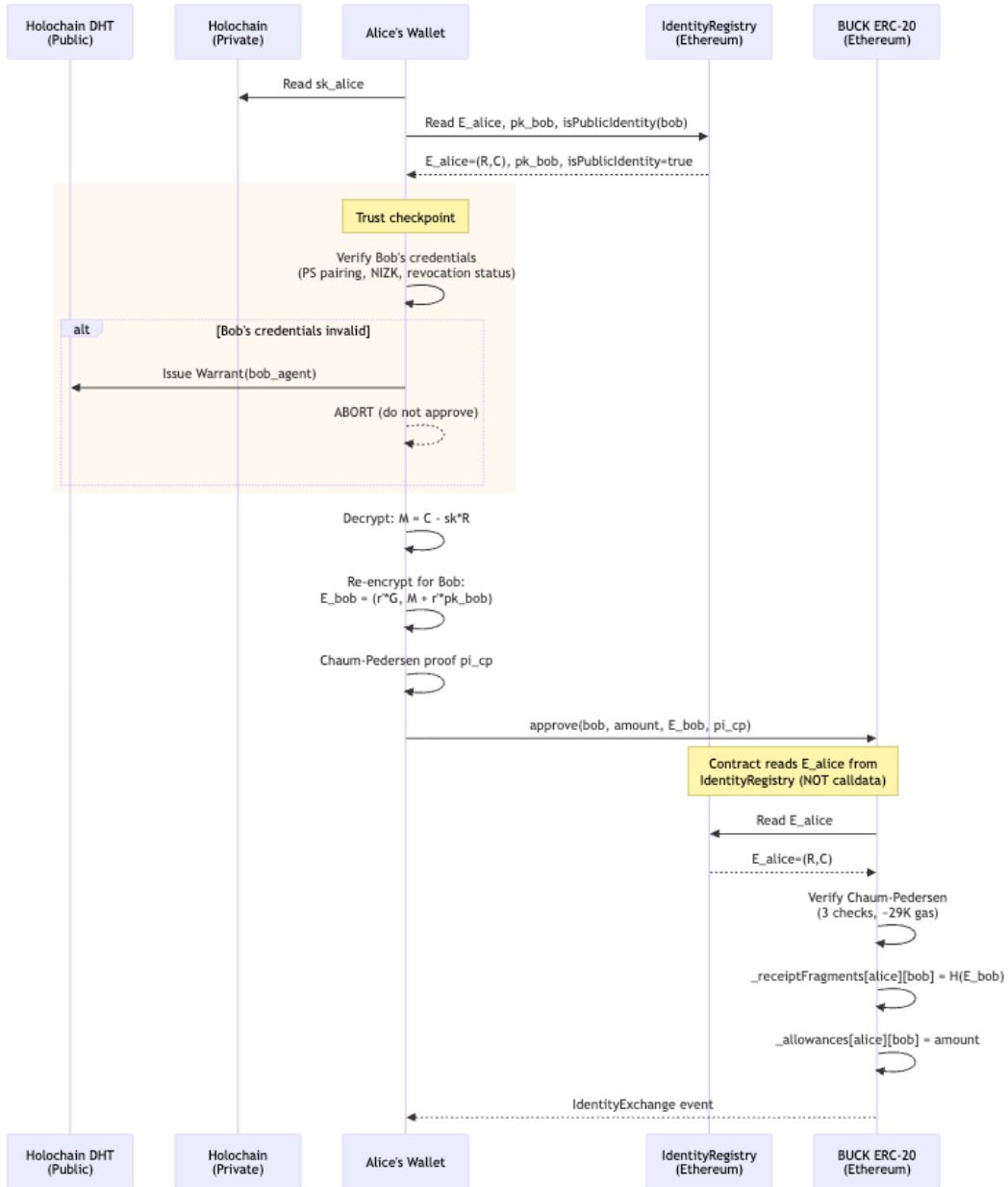
Alice’s wallet reads her Private entries, derives a credential entirely offline, then submits it to the IdentityRegistry. The contract verifies the NIZK proof ( $\sim 235K$  gas) and stores the public artifacts. The secret key and plaintext identity *never* touch Ethereum.



### 1.3 Approve: Re-Encryption with Trust Verification

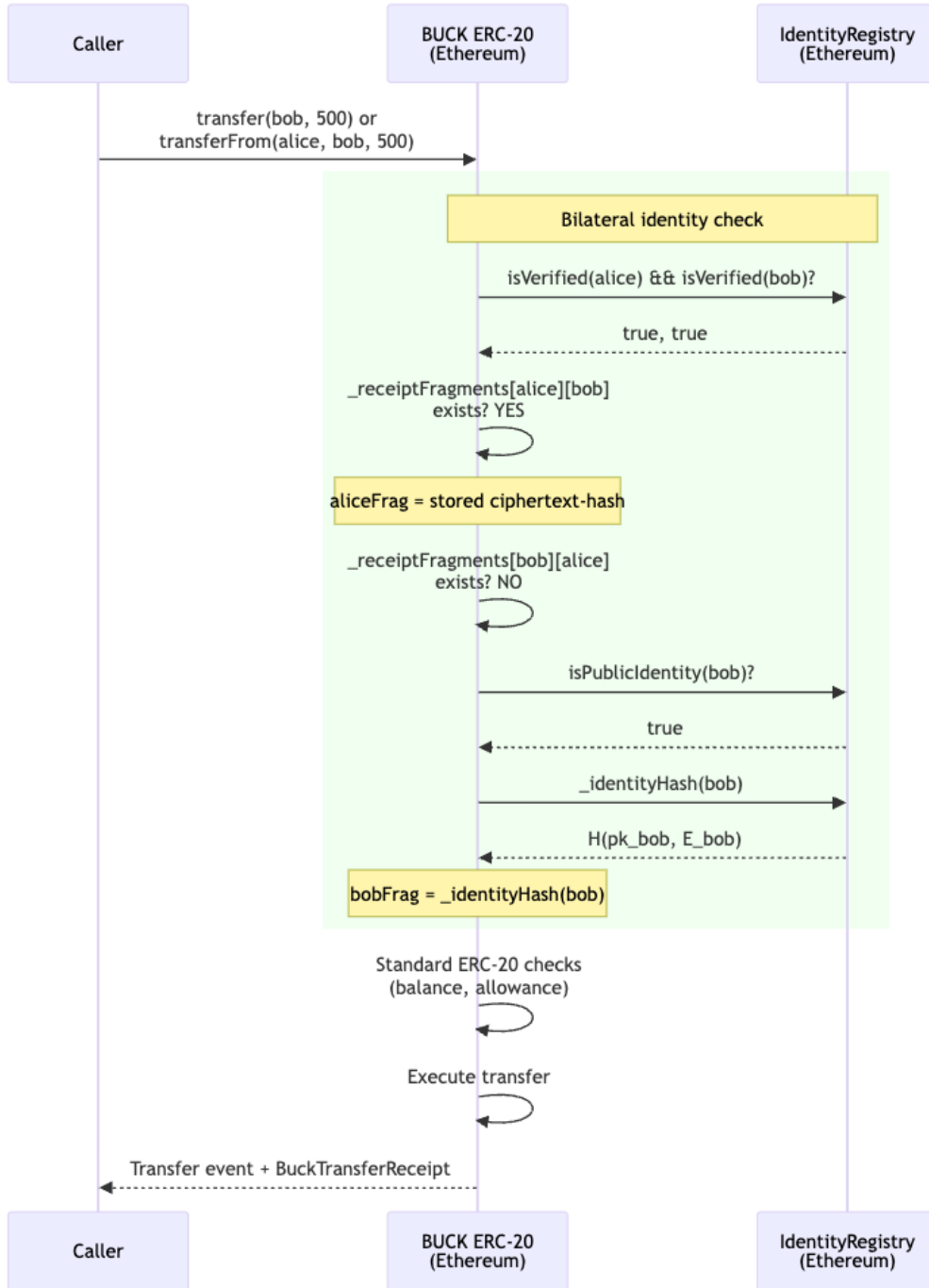
When Alice wants to transact with Bob, her wallet first *verifies Bob's credentials* before re-encrypting. This is the critical trust checkpoint: Alice's wallet reads Bob's on-chain data, checks his PS signature (via the pairing equation) and public key, and only then produces the re-encrypted ciphertext.

If anything is wrong – invalid PS signature, mismatched NIZK proof, revoked credential – Alice simply *does not call approve*. No transaction, no risk. If Alice's wallet detects provable malfeasance (e.g., a forged PS signature from a revoked issuer), it can issue a Holochain **Warrant** against Bob's agent. Multiple independent wallets detecting the same fraud produce consensus: the warranted agent becomes untrusted on the DHT.



## 1.4 Transfer: Everything On-Chain

By the time `transfer` or `transferFrom` is called, all proofs have already been verified. The EVM code only checks boolean flags and stored receipt fragments – no cryptography runs during the transfer itself. Public-Identity contracts (those bound via `bindContract(target, pk, E, isPublicIdentity_=true)`) need not have a per-pair receipt fragment: the receipt falls back to the bound `_identityHash` for that side.

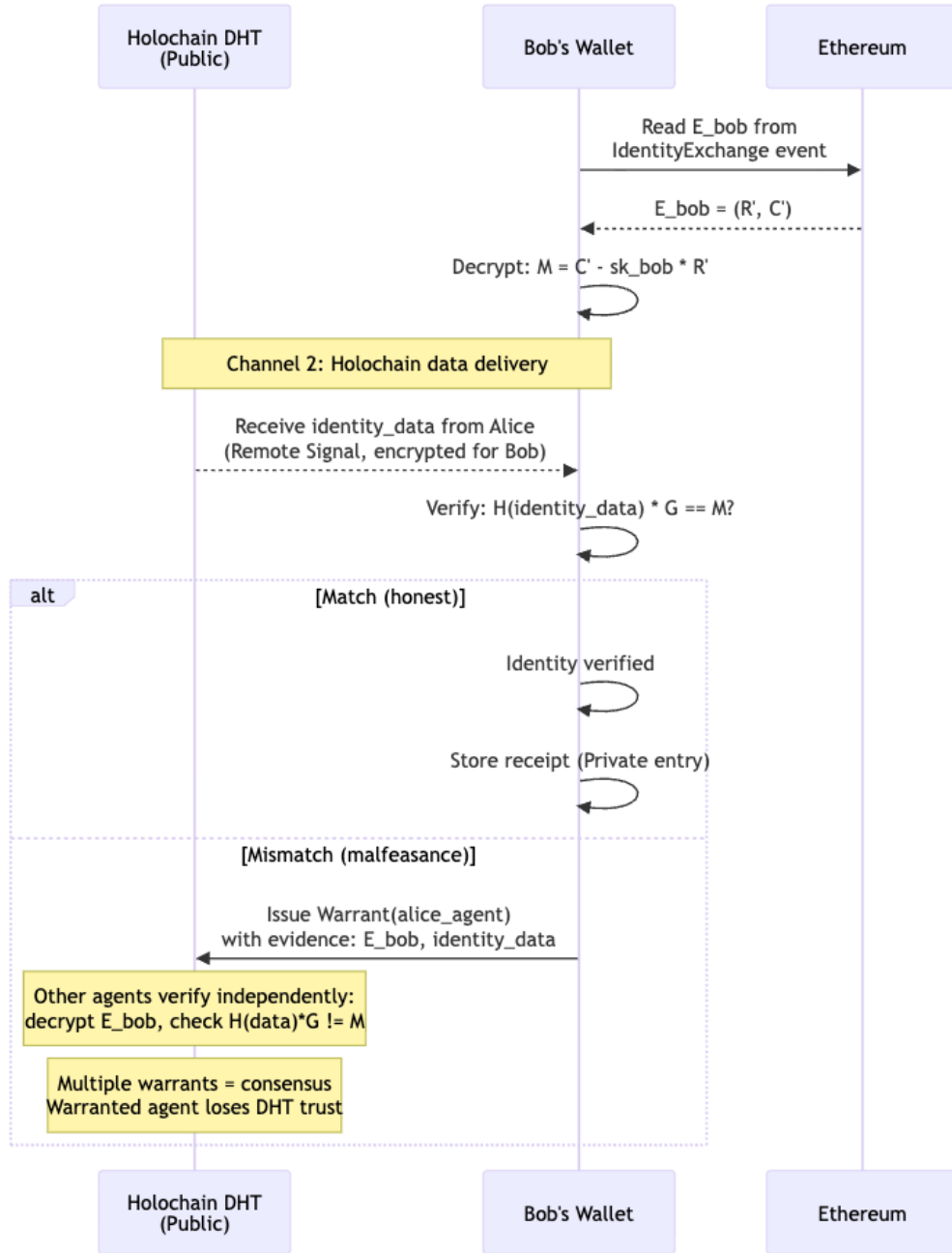


### 1.5 Two-Channel Verification and Malfeasance Detection

After the transfer, Bob verifies Alice’s identity using two independent channels. Channel 1 (Ethereum) provides the binding proof; Channel 2 (Holochain) provides the content. Neither channel alone is sufficient – this is what makes the design tamper-evident.

If Bob detects that the off-chain `identity_data` doesn’t match the on-chain point `M`, this is *provable* malfeasance: Bob holds the ciphertext (on-chain, timestamped) and the mismatched data (signed by Alice’s Holochain agent). Bob issues a Warrant; other agents independently verify the

evidence. Since Holochain agents sign their source chain entries, Alice cannot deny having sent the bad data.



## 1.6 The Four Transfer Modes

The amount of pre-work depends on the visibility of each party:

| Sender  | Receiver | Who Calls <code>approve</code> | Holochain Work        | On-Chain Gas        |
|---------|----------|--------------------------------|-----------------------|---------------------|
| Private | Private  | Both (Alice→Bob, Bob→Alice)    | Both exchange via DHT | 2 x ~29K            |
| Private | Public   | Alice only                     | Alice sends via DHT   | 1 x ~29K            |
| Public  | Private  | Bob only                       | Bob sends via DHT     | 1 x ~29K            |
| Public  | Public   | Neither                        | None                  | 0 (standard ERC-20) |

In every case, the `transfer` / `transferFrom` call itself does *no* cryptography – it only reads the boolean results of prior `approve` calls and the `isPublicIdentity` flag bound at `bindContract` time. All proof verification happens during `approve`.

## 1.7 The AMM Pool Scenario: Retroactive Identity Recovery

Consider Bob’s AMM liquidity pool – a public BUCK account that processes swaps automatically. Over its lifetime, a million private accounts call `approve(pool, amount, E_pool, proof)` and then swap via `transferFrom`. Bob’s pool contract never examines any counterparty’s identity at the time of the swap. It doesn’t need to: the EVM already verified every proof.

Eighteen months later, a regulator subpoenas Bob for Alice’s identity – account #500,000 – in connection with a fraud investigation. How does Bob comply, and how does the on-chain record prove his pool wasn’t complicit?

### 1.7.1 What the Chain Already Guarantees

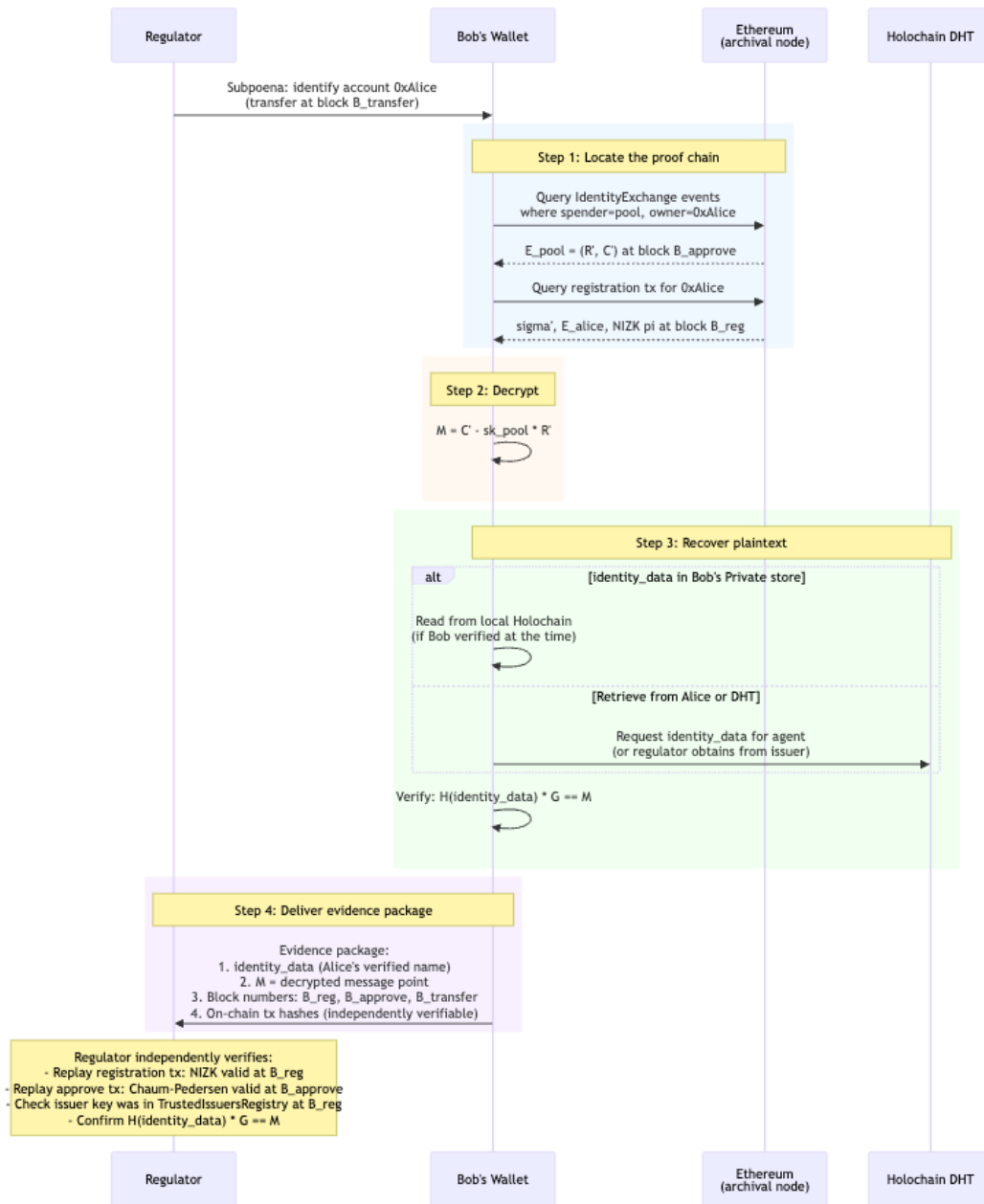
Every `approve` that succeeded is an *immutable proof receipt*. The EVM verified the Chaum-Pedersen proof at that block height; the IdentityRegistry verified Alice’s NIZK at registration time. These facts are encoded in the transaction receipts themselves – the transactions *could not have succeeded* without valid proofs.

The chain of cryptographic commitments is:

1. **Issuance:** A trusted issuer (in `TrustedIssuersRegistry`) signed Alice’s identity scalar  $m$  with a PS signature  $\sigma$ . The issuer’s public key was on-chain and valid at that epoch.
2. **Registration:** The IdentityRegistry verified Alice’s NIZK proof – binding  $\sigma'$  (rerandomized PS signature) to  $E_{\text{alice}}$  (ElGamal ciphertext). The NIZK proves the *same*  $m$  appears in both. The transaction succeeded at block  $B_{\text{reg}}$ ; the NIZK was valid at that block.
3. **Approve:** The BUCK contract verified Alice’s Chaum-Pedersen proof – proving  $E_{\text{pool}}$  encrypts the *same* message point  $M$  as her registered  $E_{\text{alice}}$ . The contract read  $E_{\text{alice}}$  from the IdentityRegistry (not from `calldata`), so Alice couldn’t substitute a different ciphertext. The transaction succeeded at block  $B_{\text{approve}}$ .
4. **Transfer:** The bilateral identity check passed. The pool was bound under a Public Identity (`isPublicIdentity(pool) = true` via `IdentityRegistry.bindContract`), and Alice had a valid `_receiptFragments[alice][pool]` from her `approve`. The pool side fell back to the bound `_identityHash(pool)`. The swap executed at block  $B_{\text{transfer}}$ .

None of these steps required Bob to be online, to run a Holochain node, or to examine Alice’s identity. The EVM did it for him.

## 1.7.2 Bob Responds to the Subpoena



## 1.7.3 Why Bob Is Protected

Bob's defense is the proof chain itself. He doesn't need to argue that he checked Alice's identity – the *contract* checked it, and the Ethereum transaction log is the evidence:

- **Registration tx succeeded**  $\Rightarrow$  NIZK was valid  $\Rightarrow$  a trusted issuer certified  $m$  at that epoch.
- **Approve tx succeeded**  $\Rightarrow$  Chaum-Pedersen was valid  $\Rightarrow E_{\text{pool}}$  encrypts the same  $M$  as  $E_{\text{alice}}$ .
- **Bob decrypts**  $E_{\text{pool}}$  to  $M$ ;  $M = H(\text{identity\_data}) \cdot G$ .

The regulator can independently replay these transactions against an archival node. No trust in Bob is required – the math is self-certifying.

Even in adversarial scenarios, Bob is covered:

| Scenario                             | Why Bob is protected   |
|--------------------------------------|--|
| Alice used a stolen identity         | The <i>issuer</i> certified it; Bob’s contract verified the issuer’s PS signature was valid at that epoch. Liability falls on the issuer’s KYC process.      |
| Alice’s issuer key was later revoked | The registration tx is timestamped. The issuer key was in TrustedIssuersRegistry at block $B_{\text{reg}}$ .   |
| Alice’s identity epoch expired       | Epoch expiry is enforced at registration time. If the epoch was valid then, the proof stands.  |
| Alice’s Holochain agent is offline   | Bob needs only $E_{\text{pool}}$ (on-chain) and <code>sk_pool</code> (his own key). The plaintext can be obtained from the issuer if Alice is uncooperative. |
| Alice denies the transaction         | The approve tx is signed by Alice’s Ethereum account. Non-repudiation is inherent in the blockchain.   |

#### 1.7.4 Data Availability Over Time

The critical question: will Bob still have what he needs 18 months later?

| Data                                    | Availability   | Notes                                      |
|---|--|--|
| $E_{\text{pool}}$ ( $R', C'$ )          | Permanent (Ethereum event log)   | IdentityExchange event                     |
| $E_{\text{alice}}$                      | Permanent (IdentityRegistry storage)                                     | Until account is closed                    |
| $\sigma', \text{NIZK } \pi$             | Permanent (registration tx calldata)                                     | Archival node required                     |
| Chaum-Pedersen proof                    | Permanent (approve tx calldata)  | Archival node required                     |
| <code>sk_pool</code> (Bob’s secret key) | Bob’s custody  | Standard key management                    |
| <code>identity_data</code> (plaintext)  | Holochain DHT (if agent is online) or Bob’s Private store (if he cached) | May require issuer cooperation as fallback |
| Issuer public key history               | TrustedIssuersRegistry (on-chain)  | Includes revocation timestamps             |

Everything except the plaintext `identity_data` is permanently on-chain. The plaintext is the one piece that lives off-chain – but Bob only needs it to produce a *human-readable name* for the regulator. The cryptographic proof chain stands without it:  $M$  is recoverable from  $E_{\text{pool}}$  alone, and  $M$  is what the issuer certified.

If Alice’s Holochain agent is long gone, the regulator can obtain `identity_data` from the issuer (who has it from the original KYC ceremony) and verify  $H(\text{identity\_data}) \cdot G = M$  independently. The issuer doesn’t need Bob’s cooperation; the regulator doesn’t need Alice’s.

#### 1.7.5 Why $M$ Is the Same Across All Accounts – And Why That’s Correct

A natural concern: if Alice derives ten accounts from the Identity Fountain, and each counterparty who decrypts their re-encrypted ciphertext recovers the same  $M = m \cdot G$ , doesn’t that break unlinkability?

No. The Identity Fountain’s guarantee is about *on-chain unlinkability* – what a passive observer (Eve) can learn from the public blockchain data alone, without any secret keys. That guarantee is airtight:

- $\sigma'_1$  and  $\sigma'_2$  are *statistically* unlinkable (information-theoretic, not merely computational)
- $E_1$  and  $E_2$  are IND-CPA secure under independent keys
- $pk_1$  and  $pk_2$  are independent random points

No algorithm, no amount of computation, can correlate Alice’s accounts from on-chain data.

The identity scalar  $m = H(\text{identity\_data})$  is *intentionally invariant* because it *is* Alice’s identity. If each account used a different  $m$ , the two-channel verification would break: Bob couldn’t verify that the `identity_data` he received matches what the issuer certified. The Identity Fountain protects the *credential wrapping* ( $\sigma'$ ,  $pk$ ,  $E$ ), not the identity itself.

Think of it as giving Alice a new disguise for each account – different keys, different signature, different ciphertext. The disguises fool passive surveillance (Eve). They don’t fool someone Alice voluntarily shows her face to (Bob, during `approve`).

| Observer                | Can link accounts? | Why?   |
|-------------------------|--------------------|--|
| Eve (passive, on-chain) | NO                 | All public artifacts are independent random values |
| Bob (authorized)        | Learns one account | By design – the point of <code>approve</code>      |
| Bob + Carol (colluding) | Yes, via $M$       | But both already know Alice’s name                 |
| Issuer (on-chain only)  | NO                 | Can’t extract $M$ without a secret key             |
| Issuer (given $M$ )     | YES                | By design – regulatory compliance path             |

The privacy boundary is clear: *before* `approve`, Alice’s accounts are unlinkable to everyone. *After* `approve`, the specific counterparty learns Alice’s identity for that account – and only that counterparty (plus anyone with whom they share  $M$ , but they already shared Alice’s name via the `identity_data` exchange).

### 1. Alice Tries to Hide by Not Publishing `identity_data`

Alice might try to game the system: complete the on-chain `approve` (re-encryption + Chaum-Pedersen proof), satisfying the ERC-20’s bilateral identity check, but never publish her `identity_data` to the Holochain DHT. She hopes that if a subpoena arrives, Bob won’t be able to produce a human-readable name.

This doesn’t work. Bob still holds the cryptographic thread:

- Bob decrypts  $E_{\text{pool}}$  to recover  $M = m \cdot G$ . This requires only Bob’s secret key and the on-chain ciphertext – Alice’s cooperation is unnecessary.
- The regulator takes  $M$  to the issuer. The issuer computes  $m_i \cdot G$  for each identity they have issued and finds the match. The issuer has Alice’s `identity_data` from the original KYC ceremony.
- The issuer confirms: this  $M$  belongs to Alice Johnson.

Alice’s attempt to hide by not publishing is futile: the on-chain  $E_{\text{pool}}$  and the issuer’s records are sufficient. Alice cannot retroactively alter either.

### 2. Bob’s Wallet Detects Non-Publication

Bob’s wallet (Holochain hApp) can proactively monitor for counterparties who complete the on-chain `approve` but fail to publish the required DHT entries. The expected protocol is:

- (a) Alice calls `approve(pool, amount, E_pool, proof)` on Ethereum.
- (b) Alice’s wallet publishes a DHT entry containing her `identity_data`, encrypted for Bob’s Holochain agent, and keyed by the `IdentityExchange` event hash.
- (c) Bob’s wallet watches for the `IdentityExchange` event and looks up the corresponding DHT entry within a grace period.

If step 2 never happens – the on-chain `approve` succeeded but no DHT entry appears – Bob’s wallet can:

- Flag the account as non-compliant (Alice approved on-chain but didn’t complete the Holochain identity exchange).
- Issue a Holochain **Warrant** against Alice’s agent, citing the on-chain tx hash as evidence of protocol violation.
- Report the account to pool operators or compliance systems.
- Optionally, Bob’s contract could revoke the approval (`reset_receiptFragments[alice][pool]`), blocking future swaps until Alice completes the full protocol.

The warrant is independently verifiable: any Holochain agent can check that the `IdentityExchange` event exists on Ethereum but the corresponding DHT entry does not. Multiple independent warrants produce DHT-level consensus that Alice’s agent is non-compliant.

Even if Alice never publishes, Bob’s regulatory position is unchanged. He can still decrypt  $M$  from the on-chain ciphertext and hand it to the regulator, who obtains `identity_data` from the issuer. Alice’s non-publication delays the human-readable identification by one step (issuer lookup) but cannot prevent it.

## 2 Setup: The `alt_bn128` Curve and Helpers

All operations use BN254, the curve behind Ethereum’s `ecAdd` (0x06), `ecMul` (0x07), and `ecPairing` (0x08) precompiles. The `py_ecc.bn128` library provides exactly these primitives. Helper functions for random scalar generation, `keccak256` hashing, and point formatting are defined here; all subsequent code blocks share a single `:session` `abid` so that variables persist across steps.

```
import secrets
from py_ecc.bn128 import bn128_curve as bc, bn128_pairing as bp
from web3 import Web3

G1      = bc.G1          # Generator of G_1
G2      = bc.G2          # Generator of G_2
Z1      = bc.Z1          # Point at infinity (G_1 identity)
ORDER   = bc.curve_order # Curve group order (prime)
mul     = bc.multiply    # Scalar multiplication in G_1 or G_2
add     = bc.add         # Point addition
neg     = bc.neg         # Point negation
eq      = bc.eq         # Point equality
pairing = bp.pairing    # Optimal Ate pairing: G_2 x G_1 -> G_T

def rand():
    """Random non-zero scalar mod ORDER."""
    return secrets.randbelow(ORDER - 1) + 1

def keccak(*args):
    """keccak256 over concatenated big-endian 32-byte words, reduced mod ORDER."""
    data = b''.join(v.to_bytes(32, 'big') if isinstance(v, int)
```

```

        else str(v).encode() for v in args)
    h = Web3.solidity_keccak(['bytes'], [data])
    return int.from_bytes(h, 'big') % ORDER

def pt(P, label=""):
    """Format a G_1 point for display."""
    if bc.is_inf(P):
        return f"{label}0 (infinity)"
    return f"{label}({int(P[0]) % 10**6:>6d}..., {int(P[1]) % 10**6:>6d}...)"

print(f"BN254 curve order: {ORDER}")
print(f"G_1 generator:      {pt(G1)}")
print(f"G_2 generator:      (pair of FQ2 elements)")

BN254 curve order: 21888242871839275222246405745257275088548364400416034343698204186575808495617
G_1 generator:      ( 1...,      2...)
G_2 generator:      (pair of FQ2 elements)

```

### 3 Issuer Key Generation

The identity issuer (e.g., ATB Financial acting as Alberta’s KYC authority) generates a Pointcheval-Sanders key pair: secret key  $(x, y)$  (two scalars), public key  $(X, Y)$  (two points in  $G_2$ , published on-chain in the `TrustedIssuersRegistry`). This key pair is the root of trust for all identities the issuer certifies; every credential in the system traces back to a PS signature under this key.

```

# Issuer's PS secret key
isk_x = rand()
isk_y = rand()

# Issuer's PS public key (in G_2, published on-chain in TrustedIssuersRegistry)
IPK_X = mul(G2, isk_x)
IPK_Y = mul(G2, isk_y)

print("Issuer PS key pair generated.")
print(f"  sk = (x, y)          -- two secret scalars")
print(f"  PK = (X, Y) in G_2     -- published in TrustedIssuersRegistry")

Issuer PS key pair generated.
  sk = (x, y)          -- two secret scalars
  PK = (X, Y) in G_2  -- published in TrustedIssuersRegistry

```

#### 3.1 Same Operation via the Wallet API

The reference implementation in `alberta_buck.wallet.issuer` packages the same key generation behind a typed `Issuer` class. Wrapping the scalars produced above into an `Issuer` lets every later step in this document call the wallet API while keeping the keys identical to the raw-crypto example.

```

from alberta_buck.wallet import Issuer, PSKeyPair

ATB_ADDR = 0xa7b0000000000000000000000000000000000000000000000000000000000000a7b00
issuer_obj = Issuer(
    issuer_id="atb-financial-ca",
    issuer_addr=ATB_ADDR,
    keypair=PSKeyPair(sk_x=isk_x, sk_y=isk_y, pk_X=IPK_X, pk_Y=IPK_Y),
)
print(f"Wallet API: {issuer_obj.issuer_id!r} @ 0x{issuer_obj.issuer_addr:040x}")
print(f"  pk_X, pk_Y match the raw G_2 points above.")

Wallet API: 'atb-financial-ca' @ 0xa7b0000000000000000000000000000000000000000000000000000000000000a7b00
  pk_X, pk_Y match the raw G_2 points above.

```

## 4 KYC Ceremony – Issuer Signs Alice’s Identity

Alice presents her identity documents to the issuer, who verifies them, constructs a canonical `identity_data` JSON structure (sorted keys, no whitespace), and computes the identity scalar  $m = \text{keccak256}(\text{identity\_data}) \bmod \text{ORDER}$ . The issuer then produces a PS signature  $\sigma = (h, (x + m \cdot y) \cdot h)$  binding  $m$  to the issuer’s secret key. The bilinear pairing equation verifies the signature, and a counter-example confirms that any change to `identity_data` (producing a different  $m$ ) causes verification to fail.

### 4.1 Alice’s Plaintext Identity

```
import json

# Alice’s verified identity (canonical encoding -- field order matters)
alice_identity_data = json.dumps({
    "given_name": "Alice",
    "family_name": "Johnson",
    "jurisdiction": "Alberta, Canada",
    "id_type": "Alberta Identity Card",
    "id_number": "AIC-2026-4839201",
    "date_of_birth": "1992-03-15",
    "issuer_id": "atb-financial-ca",
    "issued_at": "2026-01-20T14:30:00Z",
    "epoch": 42
}, sort_keys=True, separators=(',', ':'))

print("Alice’s identity_data (canonical JSON):")
print(f" {alice_identity_data[:72]}...")
```

```
Alice’s identity_data (canonical JSON):
{"date_of_birth":"1992-03-15","epoch":42,"family_name":"Johnson","given_...
```

### 4.2 Compute Identity Scalar and PS Signature

The PS signature is  $\sigma = (h, (x + m \cdot y) \cdot h)$  where  $h$  is a random point in  $G_1$ .

```
# Identity scalar: m = keccak256(identity_data) mod ORDER
m_alice = keccak(alice_identity_data)

# PS signature: pick random h, compute sigma = (h, (x + m*y) * h)
t_h = rand()
h = mul(G1, t_h) # random G_1 point
sigma_2 = mul(h, (isk_x + m_alice * isk_y) % ORDER)
sigma = (h, sigma_2)

print(f"Identity scalar m = H(identity_data):")
print(f" m = {m_alice}")
print(f"PS signature sigma:")
print(f" sigma_1 = {pt(sigma[0])}")
print(f" sigma_2 = {pt(sigma[1])}")

Identity scalar m = H(identity_data):
m = 6446313962173165014188361468977035479170665192834402473818305319846425860127
PS signature sigma:
sigma_1 = (706798..., 83433...)
sigma_2 = (24094..., 464101...)
```

### 4.3 Verify the PS Signature (Issuer Self-Check)

The PS verification equation uses a bilinear pairing:

$$e(\sigma_1, X + m \cdot Y) = e(\sigma_2, g_2)$$

**What this proves:** The issuer's signature is mathematically valid – it binds the identity scalar  $m$  to the issuer's secret key. If the signature passes, Alice can be sure the issuer actually certified her identity (no one else knows  $(x, y)$ ).

```
# Verify: e(sigma_1, X + m*Y) == e(sigma_2, g_2)
lhs = pairing(add(IPK_X, mul(IPK_Y, m_alice)), sigma[0])
rhs = pairing(G2, sigma[1])
assert lhs == rhs, "PS signature verification FAILED"
print("PS signature verified: e(sigma_1, X + m*Y) == e(sigma_2, g_2)")

# The issuer gives Alice: (m, sigma, identity_data)
# Alice stores all three in her Holochain Private entry
print(f"\nIssuer gives Alice:")
print(f"  m           = {m_alice}")
print(f"  sigma          = (sigma_1, sigma_2)  -- PS signature")
print(f"  identity_data  = '{alice_identity_data[:40]}...')

PS signature verified: e(sigma_1, X + m*Y) == e(sigma_2, g_2)

Issuer gives Alice:
m           = 6446313962173165014188361468977035479170665192834402473818305319846425860127
sigma       = (sigma_1, sigma_2)  -- PS signature
identity_data = '{"date_of_birth":"1992-03-15","epoch":42...'
```

**How to read the output:** "PS signature verified" means both sides of the pairing equation produce the same element in  $G_T$ . If  $m$  were wrong (e.g., tampered identity data), the two pairings would produce *different*  $G_T$  elements and the assertion would fail. A counter-example follows.

#### 4.3.1 Counter-example: Wrong Identity Scalar

If Eve tries to forge a signature using a different  $m$ , the pairing equation fails. This is what a rejected verification looks like:

```
# Tampered m -- what if someone changes one field in identity_data?
m_fake = keccak("tampered_identity_data")
lhs_bad = pairing(add(IPK_X, mul(IPK_Y, m_fake)), sigma[0])
rhs_good = pairing(G2, sigma[1])
ok = (lhs_bad == rhs_good)
print(f"PS verify with wrong m: {ok}")
print(f" (The two G_T elements differ -- the pairing detects the mismatch.)")
print(f" Any change to identity_data produces a different m, which")
print(f" makes the pairing equation fail. The signature is unforgeable.")

PS verify with wrong m: False
(The two G_T elements differ -- the pairing detects the mismatch.)
Any change to identity_data produces a different m, which
makes the pairing equation fail. The signature is unforgeable.
```

## 4.4 Same Operation via the Wallet API

`Issuer.issue(...)` bundles the canonicalization,  $m$  computation, and PS signing into one call. `ps_verify(...)` performs the same pairing equation as the raw block above; the counter-example uses the same primitive against a tampered  $m$ .

```
from alberta_buck.wallet import ps_verify, identity_scalar
ALICE_ADDR = 0xa11ce0000000000000000000000000000000000000000000000000000000000000a11ce

alice_fields = {
    "given_name": "Alice",
    "family_name": "Johnson",
    "jurisdiction": "Alberta, Canada",
    "id_type": "Alberta Identity Card",
    "id_number": "AIC-2026-4839201",
    "date_of_birth": "1992-03-15",
    "issuer_id": "atb-financial-ca",
    "issued_at": "2026-01-20T14:30:00Z",
    "epoch": 42,
}
cred = issuer_obj.issue(alice_fields, ALICE_ADDR)
# Wallet computes m the same way: m = keccak256(canonical) mod ORDER
assert cred.m == identity_scalar(cred.canonical) == m_alice
# PS verification, same pairing equation as the raw block:
ok = ps_verify(issuer_obj.pk_X, issuer_obj.pk_Y, cred.sigma, cred.m)
ok_bad = ps_verify(issuer_obj.pk_X, issuer_obj.pk_Y, cred.sigma, m_fake)
print(f"Wallet API: cred.m == m_alice : {cred.m == m_alice}")
print(f" ps_verify(sigma, m_alice) : {ok}")
print(f" ps_verify(sigma, m_fake) : {ok_bad}")
```

```
Wallet API: cred.m == m_alice : True
ps_verify(sigma, m_alice) : True
ps_verify(sigma, m_fake) : False
```

## 5 The Identity Fountain – Alice Derives a Fresh Credential

Alice wants to create a new Ethereum account. Her Holochain wallet runs the Identity Fountain entirely offline, producing four artifacts: a rerandomized PS signature  $\sigma'$  (statistically unlinkable to the original), a fresh key pair  $(sk, pk)$ , an ElGamal encryption of  $m$  under  $pk$ , and a NIZK sigma-protocol proof binding the PS signature to the ciphertext. Each step is demonstrated below: rerandomization preserves signature validity while destroying correlation, and ElGamal decryption recovers the correct message point. A counter-example confirms that the wrong secret key yields an unrelated random point.

### 5.1 Rerandomize the PS Signature

Alice picks a random scalar  $t$  and computes  $\sigma' = (t \cdot \sigma_1, t \cdot \sigma_2)$ . This is a valid PS signature on the same  $m$  – but *statistically unlinkable* to the original  $\sigma$ .

**What this proves:** Rerandomization produces a fresh, valid PS signature that cannot be correlated with the original – even by the issuer.

```
# Rerandomize: sigma' = (t * sigma_1, t * sigma_2)
t_rerand = rand()
sigma_p_1 = mul(sigma[0], t_rerand)
sigma_p_2 = mul(sigma[1], t_rerand)
sigma_p = (sigma_p_1, sigma_p_2)

# Verify sigma' is still a valid PS signature on m
```

```

lhs = pairing(add(IPK_X, mul(IPK_Y, m_alice)), sigma_p[0])
rhs = pairing(G2, sigma_p[1])
assert lhs == rhs, "Rerandomized PS signature FAILED"
assert not bc.is_inf(sigma_p[0]), "sigma'_1 must not be point at infinity"
print("Rerandomized PS signature verified.")
print(f" sigma'_1 = {pt(sigma_p[0])}")
print(f" sigma'_2 = {pt(sigma_p[1])}")
print(f"\nOriginal and rerandomized signatures are unlinkable:")
print(f" sigma_1 = {pt(sigma[0])}")
print(f" sigma'_1 = {pt(sigma_p[0])}")
print(f" (No algorithm can correlate them.)")

```

```

Rerandomized PS signature verified.
sigma'_1 = (435556..., 806524...)
sigma'_2 = (502712..., 527449...)

```

```

Original and rerandomized signatures are unlinkable:
sigma_1 = (706798..., 83433...)
sigma'_1 = (435556..., 806524...)
(No algorithm can correlate them.)

```

**How to read the output:** Two things to check: (1) "Rerandomized PS signature verified" confirms  $\sigma'$  still satisfies the pairing equation. (2) Compare the coordinate fragments of `sigma_1` vs. `sigma'_1` – they share no visible pattern. This isn't coincidence: the rerandomization is *statistical* (information-theoretic), not merely computational.

## 5.2 Generate Fresh Identity Key Pair and ElGamal Encryption

Alice generates a new key pair on `alt_bn128` and encrypts her identity scalar  $m$  under her own public key using ElGamal:

$$E_{\text{alice}} = (r \cdot G, m \cdot G + r \cdot pk_{\text{alice}})$$

**What this proves:** Alice can encrypt her identity under her own public key and recover it. ElGamal decryption yields the *point*  $M = m \cdot G$ , not the scalar  $m$  (solving the discrete log is infeasible).

```

# Fresh identity key pair for this account
sk_alice = rand()
pk_alice = mul(G1, sk_alice)

# ElGamal encryption of m under pk_alice
r_enc = rand()
R_alice = mul(G1, r_enc) # R = r * G
M_alice = mul(G1, m_alice) # M = m * G (message point)
C_alice = add(M_alice, mul(pk_alice, r_enc)) # C = m*G + r*pk
E_alice = (R_alice, C_alice)

print(f"Fresh key pair for this Ethereum account:")
print(f" sk_alice = {sk_alice}")
print(f" pk_alice = {pt(pk_alice)}")
print(f"\nElGamal ciphertext E_alice = (R, C):")
print(f" R = r*G = {pt(R_alice)}")
print(f" C = m*G+r*pk = {pt(C_alice)}")
print(f"\nMessage point M = m*G:")
print(f" M = {pt(M_alice)}")

# Verify Alice can decrypt her own ciphertext
M_check = add(C_alice, neg(mul(R_alice, sk_alice))) # C - sk*R = M
assert eq(M_check, M_alice), "ElGamal self-decryption FAILED"
print(f"\nAlice decrypts: C - sk*R = {pt(M_check)} (matches M)")

```

```

Fresh key pair for this Ethereum account:
  sk_alice = 13429079965490741492569193501572007864312167093356585818547301549935110836187
  pk_alice = (587319..., 958871...)

ElGamal ciphertext E_alice = (R, C):
  R = r*G = (535490..., 86183...)
  C = m*G+r*pk = (894443..., 534845...)

Message point M = m*G:
  M = (739132..., 937433...)

Alice decrypts: C - sk*R = (739132..., 937433...) (matches M)

```

**How to read the output:** The last two lines show the decrypted point  $C - sk \cdot R$  and the original  $M$ . Their coordinate fragments must match – this confirms the ciphertext is well-formed. If Alice used the wrong secret key, the decrypted point would be a random, unrelated point (shown in the counter-example below).

### 5.2.1 Counter-example: Wrong Secret Key

An attacker who doesn't know `sk_alice` cannot decrypt to the correct  $M$ . The result is an unrelated random point:

```

sk_eve = rand() # Eve's key, not Alice's
M_wrong = add(C_alice, neg(mul(R_alice, sk_eve)))
print(f"Eve decrypts with wrong key:")
print(f"  M_wrong = {pt(M_wrong)}")
print(f"  M_real = {pt(M_alice)}")
print(f"  Match: {eq(M_wrong, M_alice)}")
print(f"  (Completely different point -- no information about m is leaked.)")

```

```

Eve decrypts with wrong key:
  M_wrong = (194251..., 517317...)
  M_real = (739132..., 937433...)
  Match: False
  (Completely different point -- no information about m is leaked.)

```

### 5.3 NIZK Proof: Binding PS Signature to ElGamal Ciphertext

Alice must prove (without revealing  $m$  or  $r$ ) that:

- (a)  $\sigma'$  is a valid PS signature on  $m$
- (b)  $E_{\text{alice}}$  encrypts the same  $m$

This is a Schnorr-family sigma protocol. The PS relation is *linear* in  $m$ , so no Groth-Sahai proofs or SNARKs are needed.

```

# === PROVER (Alice, offline) ===
# Commit: pick random blinding scalars
m_tilde = rand() # blinding for m
r_tilde = rand() # blinding for r

# Commitment for PS equation (a): A_ps = m_tilde * sigma'_1
# (This will be checked against the pairing equation)
A_ps = mul(sigma_p[0], m_tilde)

# Commitments for ElGamal equations (b) and (c):
T_C = add(mul(G1, m_tilde), mul(pk_alice, r_tilde)) # m~*G + r~*pk

```

```

T_R = mul(G1, r_tilde)                                # r~*G

# Fiat-Shamir challenge: hash all public inputs + commitments
# (In Ethereum: includes registrant_address for domain separation)
e = keccak(
    sigma_p[0][0], sigma_p[0][1], # sigma'_1
    sigma_p[1][0], sigma_p[1][1], # sigma'_2
    R_alice[0], R_alice[1],       # E_alice.R
    C_alice[0], C_alice[1],       # E_alice.C
    pk_alice[0], pk_alice[1],     # pk_alice
    A_ps[0], A_ps[1],             # PS commitment
    T_C[0], T_C[1],               # ElGamal C commitment
    T_R[0], T_R[1],               # ElGamal R commitment
)

# Responses
s_m = (m_tilde + e * m_alice) % ORDER
s_r = (r_tilde + e * r_enc) % ORDER

pi = (e, s_m, s_r, A_ps, T_C, T_R)
print("NIZK proof pi = (e, s_m, s_r, A_ps, T_C, T_R)")
print(f" challenge e = {e}")
print(f" response s_m = {s_m}")
print(f" response s_r = {s_r}")

NIZK proof pi = (e, s_m, s_r, A_ps, T_C, T_R)
challenge e = 19408665787197568013438050849908992331627024433538315193292703486036442177804
response s_m = 989722230010420555156753637767192651761201022118101473590839669595608882526
response s_r = 13509042462376563756590884605508345748991673110712709754959944090053895438866

```

## 6 Registration – On-Chain Verification

Alice submits  $(pk, E_{\text{alice}}, \sigma', \pi)$  to the IdentityRegistry contract. The contract runs five verification checks using the `ecPairing`, `ecMul`, and `ecAdd` precompiles (~235K gas total): two ElGamal consistency checks, one PS pairing product check, a Fiat-Shamir challenge reconstruction, and a non-triviality guard. All five must pass for `isVerified[alice]` to be set. The full verification runs below, followed by a counter-example where Alice tries to register an ElGamal ciphertext containing a different  $m$  than her PS signature – the ElGamal consistency check catches the mismatch.

### 6.1 Verify the NIZK Proof (What the Contract Computes)

**What this proves:** The same  $m$  appears in both the PS signature and the ElGamal ciphertext, without revealing  $m$  or  $r$ . Five checks run:

- (b) ElGamal C consistency: the committed  $m$  is inside  $C$
- (c) ElGamal R consistency: the committed  $r$  is inside  $R$
- (a) PS pairing product: the same  $m$  satisfies the PS equation
- (d) Fiat-Shamir: the challenge was honestly derived (not chosen after the fact)
- (e) Non-triviality:  $\sigma'_1 \neq \mathcal{O}$  (prevents trivial forgery)

```

# === VERIFIER (IdentityRegistry contract, on-chain) ===
# Unpack proof
e_v, s_m_v, s_r_v, A_ps_v, T_C_v, T_R_v = pi

```

```

# --- Check (b): ElGamal consistency ---
# Verify:  $s_m * G + s_r * pk_{alice} == e * C + T_C$ 
lhs_C = add(mul(G1, s_m_v), mul(pk_alice, s_r_v))
rhs_C = add(mul(C_alice, e_v), T_C_v)
assert eq(lhs_C, rhs_C), "ElGamal C check FAILED"
print("(b) ElGamal C check passed:  $s_m * G + s_r * pk == e * C + T_C$ ")

# Verify:  $s_r * G == e * R + T_R$ 
lhs_R = mul(G1, s_r_v)
rhs_R = add(mul(R_alice, e_v), T_R_v)
assert eq(lhs_R, rhs_R), "ElGamal R check FAILED"
print("(c) ElGamal R check passed:  $s_r * G == e * R + T_R$ ")

# --- Check (a): PS signature via pairing ---
# Reconstruct PS commitment:  $s_m * \sigma'_1$  should equal  $A_{ps} + e * \sigma'_2$ 
# relative to the pairing equation.
# Verify pairing product:
#  $e(s_m * \sigma'_1, Y) * e(-A_{ps}, Y) * e(e * \sigma'_1, X)$ 
#  $* e(-e * \sigma'_2, g_2) == 1$ 
# Which simplifies to:
#  $e(s_m * \sigma'_1 - A_{ps}, Y) * e(e * \sigma'_1, X) * e(-e * \sigma'_2, g_2) == 1$ 
#
# This is equivalent to checking:
#  $e(A_{ps}, Y) * e(e * \sigma'_2, g_2) == e(s_m * \sigma'_1, Y) * e(e * \sigma'_1, X)$ 

P1 = mul(sigma_p[0], s_m_v)           #  $s_m * \sigma'_1$ 
P2 = add(A_ps_v, mul(sigma_p[1], e_v)) #  $A_{ps} + e * \sigma'_2$ 
P3 = mul(sigma_p[0], e_v)           #  $e * \sigma'_1$ 

# Pairing check:  $e(P1, Y) == e(P2, Y) * e(P3, X) / e(P3, X)$ 
# More directly: verify the original PS equation holds through the proof
#  $e(s_m * \sigma'_1 - e * \sigma'_2, Y) == e(A_{ps} + e * \sigma'_1, X) \dots$  etc.
#
# Simplified: check that the reconstructed pairing equation holds:
#  $e(\sigma'_1, X + m * Y) = e(\sigma'_2, g_2)$  via the proof responses
#
# The verifier checks:  $e(s_m * s'_1, Y) * e(e * s'_1, X) == e(A_{ps}, Y) * e(e * s'_2, g_2)$ 
lhs_pair = pairing(IPK_Y, P1) * pairing(IPK_X, P3)
rhs_pair = pairing(IPK_Y, P2) * pairing(G2, mul(sigma_p[1], e_v))
# Actually let's use the direct verification approach:
# We know  $s_m = m + e * m$ . The commitment  $A_{ps} = m * \sigma'_1$ .
# So  $s_m * \sigma'_1 = m * \sigma'_1 + e * m * \sigma'_1 = A_{ps} + e * m * \sigma'_1$ 
# The verifier doesn't know  $m$ , but can check via pairing:
#  $e(s_m * \sigma'_1, Y) * e(e * \sigma'_1, X) * e(-e * \sigma'_2 - A_{ps}, g_2\_placeholder)$ 
# This gets complex in code. Let's use the direct pairing product check.

# Direct approach: verify  $e(\sigma'_1, X + m * Y) = e(\sigma'_2, g_2)$ 
# by extracting  $m$  from the sigma protocol responses:
# The key insight: the SAME  $s_m$  that satisfies (b) must also satisfy (a).
# If checks (b) and (c) pass, we know  $s_m = m + e * m$  for the  $m$  in  $E_{alice}$ .
# Now check (a): does this same  $m$  satisfy the PS equation?
#
# Reconstruct:  $m * \sigma'_1 = (s_m * \sigma'_1 - A_{ps}) / e$ 
# But we avoid division. Instead verify the pairing product equation:
#  $e(s_m * \sigma'_1 - A_{ps}, Y) == e(e * \sigma'_2, g_2) * e(-e * \sigma'_1, X)$ 
# (rearranging  $e(\sigma'_1, X + m * Y) = e(\sigma'_2, g_2)$  with substitution)

LHS_point = add(mul(sigma_p[0], s_m_v), neg(A_ps_v)) #  $s_m * s'_1 - A_{ps} = e * m * s'_1$ 
RHS_neg = add(mul(sigma_p[1], e_v), neg(mul(sigma_p[0], e_v))) #  $e * s'_2 - e * s'_1$ 
# not quite right -- use the clean verification instead

# Clean pairing product check (what ecPairing precompile computes):
#  $e(s_m * \sigma'_1, Y) * e(-A_{ps}, Y) * e(e * \sigma'_1, X) * e(-e * \sigma'_2, g_2) == 1$ 
check = (
    pairing(IPK_Y, mul(sigma_p[0], s_m_v)) #  $e(s_m * s'_1, Y)$ 
    * pairing(IPK_Y, neg(A_ps_v)) #  $e(-A_{ps}, Y)$ 
    * pairing(IPK_X, mul(sigma_p[0], e_v)) #  $e(e * s'_1, X)$ 
    * pairing(G2, neg(mul(sigma_p[1], e_v))) #  $e(-e * s'_2, g_2)$ 

```

```

)
assert check == bp.FQ12.one(), "PS pairing check FAILED"
print("(a) PS pairing check passed:")
print("    e(s_m*s'_1, Y) * e(-A_ps, Y) * e(e*s'_1, X) * e(-e*s'_2, g_2) == 1")

# --- Fiat-Shamir challenge reconstruction ---
e_reconstructed = keccak(
    sigma_p[0][0], sigma_p[0][1],
    sigma_p[1][0], sigma_p[1][1],
    R_alice[0], R_alice[1],
    C_alice[0], C_alice[1],
    pk_alice[0], pk_alice[1],
    A_ps_v[0], A_ps_v[1],
    T_C_v[0], T_C_v[1],
    T_R_v[0], T_R_v[1],
)
assert e_reconstructed == e_v, "Fiat-Shamir challenge mismatch"
print("(d) Fiat-Shamir challenge verified (binds all public inputs)")

# --- sigma'_1 != 0 check ---
assert not bc.is_inf(sigma_p[0]), "sigma'_1 is point at infinity (trivial forgery)"
print("(e) sigma'_1 != 0 (non-trivial signature)")

print("\n==> Registration PASSED. isVerified(alice) = true")

(b) ElGamal C check passed: s_m*G + s_r*pk == e*C + T_C
(c) ElGamal R check passed: s_r*G == e*R + T_R
(a) PS pairing check passed:
    e(s_m*s'_1, Y) * e(-A_ps, Y) * e(e*s'_1, X) * e(-e*s'_2, g_2) == 1
(d) Fiat-Shamir challenge verified (binds all public inputs)
(e) sigma'_1 != 0 (non-trivial signature)

==> Registration PASSED. isVerified(alice) = true

```

**How to read the output:** All five checks must print "passed". If any prints "FAILED", the proof is invalid. A counter-example follows showing what happens when Alice tries to register a ciphertext containing a *different*  $m$  than the one in her PS signature.

### 6.1.1 Counter-example: ElGamal Encrypts a Different $m$

If Alice (or an attacker) tries to encrypt one  $m$  in ElGamal but prove a PS signature on a *different*  $m$ , the ElGamal consistency check detects the mismatch:

```

# Forge: encrypt a fake m_fake in ElGamal, but use the real PS sigma
m_fake = rand()
r_fake = rand()
R_fake = mul(G1, r_fake)
C_fake = add(mul(G1, m_fake), mul(pk_alice, r_fake))

# Try to build a NIZK proof binding sigma' (real m) to E_fake (fake m)
m_t = rand(); r_t = rand()
A_fake = mul(sigma_p[0], m_t)
T_C_f = add(mul(G1, m_t), mul(pk_alice, r_t))
T_R_f = mul(G1, r_t)
e_f = keccak(
    sigma_p[0][0], sigma_p[0][1], sigma_p[1][0], sigma_p[1][1],
    R_fake[0], R_fake[1], C_fake[0], C_fake[1],
    pk_alice[0], pk_alice[1],
    A_fake[0], A_fake[1], T_C_f[0], T_C_f[1], T_R_f[0], T_R_f[1])
s_m_f = (m_t + e_f * m_alice) % ORDER # uses real m for PS...
s_r_f = (r_t + e_f * r_fake) % ORDER # ...but fake r for ElGamal

# Check (b): s_m*G + s_r*pk == e*C_fake + T_C?

```

```

lhs = add(mul(G1, s_m_f), mul(pk_alice, s_r_f))
rhs = add(mul(C_fake, e_f), T_C_f)
ok = eq(lhs, rhs)
print(f"Forged proof -- ElGamal C check: {ok}")
print(f" (The check FAILS because s_m encodes m_real but C_fake")
print(f"   encrypts m_fake. The NIZK forces both to be identical.)")

```

```

Forged proof -- ElGamal C check: False
(The check FAILS because s_m encodes m_real but C_fake
encrypts m_fake. The NIZK forces both to be identical.)

```

## 6.2 Same Operation via the Wallet API

The wallet's Identity Fountain is three calls (`rerandomize_for_registration`, `identity_keygen`, `elgamal_encrypt`) plus a registration NIZK round-trip (`registration_prove` / `registration_verify`). The wallet's Fiat-Shamir transcript binds the registrant Ethereum address for replay protection – the only substantive difference from the raw NIZK above.

```

from alberta_buck.wallet import (
    rerandomize_for_registration, identity_keygen, elgamal_encrypt,
    registration_prove, registration_verify,
)
from alberta_buck.wallet.bn254 import rand_scalar

sigma_w_p, _ = rerandomize_for_registration(cred)
wallet_kp    = identity_keygen()
r_w         = rand_scalar()
E_w        = elgamal_encrypt(mul(G1, cred.m), wallet_kp.pk, r_w)

proof_w = registration_prove(
    sigma_w_p, cred.m, r_w, wallet_kp.pk, E_w, ALICE_ADDR,
)
ok = registration_verify(
    sigma_w_p, E_w, wallet_kp.pk,
    issuer_obj.pk_X, issuer_obj.pk_Y, proof_w, ALICE_ADDR,
)
# Replay to a different registrant address must be rejected.
ok_replay = registration_verify(
    sigma_w_p, E_w, wallet_kp.pk,
    issuer_obj.pk_X, issuer_obj.pk_Y, proof_w, ALICE_ADDR ^ 1,
)
print(f"Wallet API: registration_verify (correct registrant) : {ok}")
print(f"      registration_verify (replayed registrant) : {ok_replay}")

Wallet API: registration_verify (correct registrant) : True
      registration_verify (replayed registrant) : False

```

## 7 Bob Binds His Pool as a Public-Identity Contract

Bob operates a Uniswap-style AMM liquidity pool. Bob himself is a registered EOA, but the *pool contract* is the entity that holds and moves BUCK on his behalf. At deployment time Bob (or the `BuckAwareDeployer.deployAndBind` helper, atomically) calls `IdentityRegistry.bindContract(pool, pk_bob, E_bob, isPublicIdentity_=true)`, publishing Bob's (pk, E) against the pool's address and marking the pool `isPublicIdentity = true`. Bob's `identity_data` remains world-readable in the `IdentityRegistry` under his own EOA.

Counterparties of the pool need not wait for the pool to call `approve` back: when no per-pair receipt fragment exists for the pool side, `transfer` falls back to the bound `_identityHash(pool)` for the receipt event. The bilateral identity check still requires both parties to be `isVerified`; the public-side fallback only relaxes the receipt-fragment requirement, not the verification requirement.



```

    "given_name": "Bob",
    "family_name": "Smith",
    "jurisdiction": "Alberta, Canada",
    "id_type": "Corporate Registration",
    "id_number": "AB-CORP-2026-00182",
    "date_of_birth": "1985-07-22",
    "issuer_id": "atb-financial-ca",
    "issued_at": "2026-02-01T09:00:00Z",
    "epoch": 42,
}
cred_bob = issuer_obj.issue(bob_fields, BOB_ADDR)
assert cred_bob.m == m_bob
assert ps_verify(issuer_obj.pk_X, issuer_obj.pk_Y, cred_bob.sigma, cred_bob.m)
print(f"Wallet API: cred_bob.m == m_bob : {cred_bob.m == m_bob}")
print(f" ps_verify(cred_bob.sigma, m_bob) : True")
print(f" Issuer log now has 2 entries: {len(issuer_obj.issuance_log())}")

```

```

Wallet API: cred_bob.m == m_bob : True
ps_verify(cred_bob.sigma, m_bob) : True
Issuer log now has 2 entries: 2

```

## 8 Alice Approves Bob – Re-Encryption + Chaum-Pedersen Proof

Alice wants to swap BUCK for USDT on Bob’s pool. She calls `approve(pool, amount, E_pool, proof)` to re-encrypt her identity under the pool’s bound public key (Bob’s `pk_bob`) and prove the re-encryption is correct via a Chaum-Pedersen proof (29K gas). `Approve` always requires CP, even though the pool is a Public-Identity contract -- this captures Alice’s per-pair consent, not just the registry-derivable identity hash; see [\[\[https://perry.kundert.ca/range/fin/approve Is Always CP-Bound/\]\]](https://perry.kundert.ca/range/fin/approve%20Is%20Always%20CP-Bound/) for the full rationale. The pool itself need not reciprocate with its own `=approve=`; the bilateral identity check in `=transfer=` passes both guards because `~isPublicIdentity(pool) == true` satisfies the public-side fallback, and the receipt carries Alice’s per-pair CP fragment alongside the deterministic identity hashes. Below: the re-encryption, the three-check Chaum-Pedersen verification, and a counter-example showing that re-encrypting a *different* message point causes Check 2 to fail.

### 8.1 Re-Encrypt Alice’s Identity for Bob

Alice decrypts her own ciphertext to recover  $M = m \cdot G$ , then re-encrypts under Bob’s public key with fresh randomness.

**What this proves:** Re-encryption preserves the message point  $M$ . Bob can decrypt the new ciphertext with his own secret key and recover the same  $M$  that was in Alice’s original ciphertext.

```

# Alice decrypts her own credential
M_decrypted = add(C_alice, neg(mul(R_alice, sk_alice))) # C - sk*R = M
assert eq(M_decrypted, M_alice), "Self-decryption failed"

# Re-encrypt M under Bob’s public key with fresh randomness r’
r_prime = rand()
R_for_bob = mul(G1, r_prime) # R’ = r’ * G
C_for_bob = add(M_alice, mul(pk_bob, r_prime)) # C’ = M + r’ * pk_bob
E_for_bob = (R_for_bob, C_for_bob)

print("Alice re-encrypts her identity for Bob:")
print(f" E_bob = (R’, C’) where:")
print(f" R’ = r’*G = {pt(R_for_bob)}")
print(f" C’ = M+r’*pk = {pt(C_for_bob)}")

```

```

# Verify Bob can decrypt
M_bob_decrypts = add(C_for_bob, neg(mul(R_for_bob, sk_bob)))
assert eq(M_bob_decrypts, M_alice), "Bob cannot decrypt E_bob"
print(f"\nBob decrypts: C' - sk_bob*R' = {pt(M_bob_decrypts)}")
print(f"Matches Alice's M:           {pt(M_alice)}")

```

Alice re-encrypts her identity for Bob:

```

E_bob = (R', C') where:
  R' = r'*G      = (268085..., 938691...)
  C' = M+r'*pk   = ( 96040..., 144219...)

```

```

Bob decrypts: C' - sk_bob*R' = (739132..., 937433...)
Matches Alice's M:           (739132..., 937433...)

```

**How to read the output:** The last two lines show Bob's decrypted point and Alice's original  $M$ . The coordinate fragments must be identical – same point, different ciphertext wrapping it.

## 8.2 Chaum-Pedersen Proof of Correct Re-Encryption

Alice proves that  $E_{\text{bob}}$  encrypts the same message point  $M$  as her registered  $E_{\text{alice}}$  – without revealing  $m$  or any private key.

The proof demonstrates knowledge of the re-encryption randomness  $r'$  and the difference between the two ElGamal randomness values, such that both ciphertexts decrypt to the same  $M$ .

```

# Chaum-Pedersen proof that E_alice and E_bob encrypt the same M.
#
# The key relation: C_alice - sk_alice * R_alice = M = C_bob - r' * pk_bob
# Equivalently: C_alice - sk_alice * R_alice = C_bob - r' * pk_bob
#
# Alice proves she knows (sk_alice, r') such that:
# sk_alice * R_alice = C_alice - M (she can decrypt E_alice)
# r' * pk_bob      = C_bob - M (she created E_bob correctly)
# r' * G           = R_bob (consistent randomness)

# Commit: random blinders
k1 = rand() # blinder for sk_alice
k2 = rand() # blinder for r'

T1 = mul(R_alice, k1) # k1 * R_alice
T2 = mul(pk_bob, k2) # k2 * pk_bob
T3 = mul(G1, k2)     # k2 * G

# Fiat-Shamir challenge (binds all public inputs + Ethereum context)
e_cp = keccak(
  R_alice[0], R_alice[1], C_alice[0], C_alice[1], # E_alice
  R_for_bob[0], R_for_bob[1], C_for_bob[0], C_for_bob[1], # E_bob
  pk_alice[0], pk_alice[1], # pk_alice
  pk_bob[0], pk_bob[1], # pk_bob
  T1[0], T1[1], T2[0], T2[1], T3[0], T3[1], # commitments
  42, 1337, # msg.sender, spender (stand-ins)
)

# Responses
s1_cp = (k1 + e_cp * sk_alice) % ORDER
s2_cp = (k2 + e_cp * r_prime) % ORDER

proof_cp = (e_cp, s1_cp, s2_cp)
print("Chaum-Pedersen proof:")
print(f" e = {e_cp}")
print(f" s1 = {s1_cp}")
print(f" s2 = {s2_cp}")

```

```

Chaum-Pedersen proof:
e = 1300852252227562365178083899592935362464550768388269033593317138879197357917
s1 = 1295131603057627819560401993199379362874025791642702040256233990972802442155
s2 = 21438161707089041685589888596676455100670078363009861390392130415223997637231

```

### 8.3 Verify Chaum-Pedersen Proof (What the Contract Computes)

The BUCK contract reads  $E_{\text{alice}}$  from the IdentityRegistry (never from calldata), then verifies the proof. Cost:  $\sim 29,000$  gas.

**What this proves:**  $E_{\text{alice}}$  and  $E_{\text{bob}}$  encrypt the same message point  $M$  – without revealing  $M$ ,  $\text{sk}_{\text{alice}}$ , or the re-encryption randomness  $r'$ . Three checks:

- **Check 1:**  $r'$  is consistent with  $R_{\text{bob}}$
- **Check 2:** Both ciphertexts decrypt to the same  $M$
- **Check 3:** Fiat-Shamir challenge was honestly derived

```

# === VERIFIER (BUCK contract, on-chain, ~29K gas) ===
e_v, s1_v, s2_v = proof_cp

# The verifier knows: E_alice (from registry), E_bob (from calldata),
# pk_alice (from registry), pk_bob (from registry).

# Reconstruct commitments from responses and challenge:
# T1' = s1 * R_alice - e * (C_alice - C_bob + r'*pk_bob - sk*R_alice)
# Simplified verification equations:
# s1 * R_alice == T1 + e * sk_alice * R_alice
# == T1 + e * (C_alice - M)
# But verifier doesn't know M. Use the relation C_alice - sk*R = C_bob - r'*pk_bob = M:
# s1 * R_alice - s2 * pk_bob == T1 - T2 + e*(C_alice - C_bob)
# And: s2 * G == T3 + e * R_bob

# Check 1: s2 * G == T3 + e * R_for_bob
lhs1 = mul(G1, s2_v)
rhs1 = add(T3, mul(R_for_bob, e_v))
assert eq(lhs1, rhs1), "Chaum-Pedersen check 1 FAILED"
print("Check 1 passed: s2*G == T3 + e*R_bob")
print(" (Proves r' is consistent with R_bob)")

# Check 2: s1*R_alice - s2*pk_bob == (T1 - T2) + e*(C_alice - C_bob)
lhs2 = add(mul(R_alice, s1_v), neg(mul(pk_bob, s2_v)))
rhs2 = add(add(T1, neg(T2)), mul(add(C_alice, neg(C_for_bob)), e_v))
assert eq(lhs2, rhs2), "Chaum-Pedersen check 2 FAILED"
print("Check 2 passed: s1*R_a - s2*pk_b == (T1-T2) + e*(C_a - C_b)")
print(" (Proves both ciphertexts encrypt the same M)")

# Check 3: Fiat-Shamir challenge
e_recon = keccak(
    R_alice[0], R_alice[1], C_alice[0], C_alice[1],
    R_for_bob[0], R_for_bob[1], C_for_bob[0], C_for_bob[1],
    pk_alice[0], pk_alice[1],
    pk_bob[0], pk_bob[1],
    T1[0], T1[1], T2[0], T2[1], T3[0], T3[1],
    42, 1337,
)
assert e_recon == e_v, "Fiat-Shamir challenge mismatch"
print("Check 3 passed: Fiat-Shamir challenge verified")

print("\n==> approve() PASSED. Alice's re-encrypted identity stored for Bob.")
print(" _receiptFragments[alice][bob] = keccak256(E_bob)")

```

```

Check 1 passed: s2*G == T3 + e*R_bob
  (Proves r' is consistent with R_bob)
Check 2 passed: s1*R_a - s2*pk_b == (T1-T2) + e*(C_a - C_b)
  (Proves both ciphertexts encrypt the same M)
Check 3 passed: Fiat-Shamir challenge verified

==> approve() PASSED. Alice's re-encrypted identity stored for Bob.
  _receiptFragments[alice][bob] = keccak256(E_bob)

```

**How to read the output:** All three "passed" lines must appear. If Alice re-encrypted for a *different* public key or used a different  $M$ , Check 2 would fail. A counter-example follows.

### 8.3.1 Counter-example: Re-Encryption with Wrong Message

If Alice tries to pass off a different identity point in the re-encrypted ciphertext, Check 2 catches the discrepancy:

```

# Alice tries to re-encrypt a DIFFERENT M (e.g., someone else's identity)
m_other = rand()
M_other = mul(G1, m_other)
r_bad = rand()
R_bad = mul(G1, r_bad)
C_bad = add(M_other, mul(pk_bob, r_bad)) # encrypts M_other, not M_alice

# Build Chaum-Pedersen proof (Alice tries to cheat)
k1b = rand(); k2b = rand()
T1b = mul(R_alice, k1b); T2b = mul(pk_bob, k2b); T3b = mul(G1, k2b)
e_bad = keccak(
  R_alice[0], R_alice[1], C_alice[0], C_alice[1],
  R_bad[0], R_bad[1], C_bad[0], C_bad[1],
  pk_alice[0], pk_alice[1], pk_bob[0], pk_bob[1],
  T1b[0], T1b[1], T2b[0], T2b[1], T3b[0], T3b[1], 42, 1337)
s1b = (k1b + e_bad * sk_alice) % ORDER
s2b = (k2b + e_bad * r_bad) % ORDER

# Check 2: s1*R_a - s2*pk_b == (T1-T2) + e*(C_alice - C_bad)?
lhs = add(mul(R_alice, s1b), neg(mul(pk_bob, s2b)))
rhs = add(add(T1b, neg(T2b)), mul(add(C_alice, neg(C_bad)), e_bad))
ok = eq(lhs, rhs)
print(f"Forged re-encryption -- Check 2: {ok}")
print(f" (FAILS: C_bad encrypts a different M than C_alice.)")
print(f" (The contract rejects the approve() call.)")

Forged re-encryption -- Check 2: False
(FAILS: C_bad encrypts a different M than C_alice.)
(The contract rejects the approve() call.)

```

## 8.4 Same Operation via the Wallet API

The wallet's `chaum_pedersen_prove` / `chaum_pedersen_verify` run the same three checks as the on-chain verifier, with the Fiat-Shamir transcript binding (`sender`, `spender`, `chainid`) for replay protection across addresses and chains. A wrong- $M$  ciphertext (the same forgery as the counter-example above) fails Check 2.

```

from alberta_buck.wallet import (
    ElGamalCiphertext, chaum_pedersen_prove, chaum_pedersen_verify,
)
E_alice_w = ElGamalCiphertext(R=R_alice, C=C_alice)
E_for_bob_w = ElGamalCiphertext(R=R_for_bob, C=C_for_bob)
proof_cp_w = chaum_pedersen_prove(
    E_alice_w, E_for_bob_w, pk_alice, pk_bob,

```

```

    sk_alice, r_prime, ALICE_ADDR, BOB_ADDR, 1,
)
ok_good = chaum_pedersen_verify(
    E_alice_w, E_for_bob_w, pk_alice, pk_bob,
    proof_cp_w, ALICE_ADDR, BOB_ADDR, 1,
)
# Same forgery as the counter-example: re-encrypt a different M.
E_bad_w = ElGamalCiphertext(R=R_bad, C=C_bad)
proof_bad_w = chaum_pedersen_prove(
    E_alice_w, E_bad_w, pk_alice, pk_bob,
    sk_alice, r_bad, ALICE_ADDR, BOB_ADDR, 1,
)
ok_bad = chaum_pedersen_verify(
    E_alice_w, E_bad_w, pk_alice, pk_bob,
    proof_bad_w, ALICE_ADDR, BOB_ADDR, 1,
)
print(f"Wallet API: chaum_pedersen_verify(honest E_for_bob) : {ok_good}")
print(f"          chaum_pedersen_verify(forged wrong-M)   : {ok_bad}")

Wallet API: chaum_pedersen_verify(honest E_for_bob) : True
          chaum_pedersen_verify(forged wrong-M)   : False

```

## 9 Transfer – Bilateral Identity Check

With Alice’s `approve` completed and the pool bound under a Public Identity, the `transfer` can proceed. The contract performs a bilateral identity check: both parties must be `isVerified`, and the receipt event must carry a fragment for each side. Per-pair receipt fragments stored at `approve` time are the primary source. When a side has no fragment and that side is `isPublicIdentity = true`, the bound `_identityHash` falls in for that side’s receipt slot. No cryptography runs at transfer time – only state lookups. We simulate the check for our Private-to-Public-Identity scenario.

```

# Simulating the contract's bilateral identity check
# Alice (private EOA) -> Pool (public contract)
is_verified_alice      = True
is_verified_pool       = True
is_public_alice        = False # EOA, not public-identity
is_public_pool         = True  # bindContract(pool, ..., true)
receipt_alice_to_pool  = True  # set during approve() above
receipt_pool_to_alice  = False # pool never CP-approved alice

assert is_verified_alice and is_verified_pool

# toHash = _receiptFragments[from][to] -- Alice->Pool direction
if receipt_alice_to_pool:
    toHash = "H(E_pool)" # Alice's per-pair CP fragment
elif is_public_alice or is_public_pool:
    toHash = "_identityHash(pool)" # fallback (pool is public)
else:
    raise Exception("sender must identity-approve recipient")
# fromHash = _receiptFragments[to][from] -- Pool->Alice direction
if receipt_pool_to_alice:
    fromHash = "H(E_alice)" # Pool's CP fragment (doesn't exist)
elif is_public_alice or is_public_pool:
    fromHash = "_identityHash(alice)" # fallback (pool is public)
else:
    raise Exception("recipient must identity-approve sender")

print("Bilateral identity check:")
print(f"  toHash (Alice->Pool): {toHash}")
print(f"  fromHash (Pool->Alice): {fromHash}")
print(f"\n  both hashes present => transfer SUCCEEDS")
print(f"\n=> transfer(alice, pool, 500 BUCK) SUCCEEDS")
print(f"  BuckTransferReceipt(fromHash={fromHash}, toHash={toHash})")

```

```

Bilateral identity check:
  toHash (Alice->Pool): H(E_pool)
  fromHash (Pool->Alice): _identityHash(alice)

  both hashes present => transfer SUCCEEDS

==> transfer(alice, pool, 500 BUCK) SUCCEEDS
    BuckTransferReceipt(fromHash=_identityHash(alice), toHash=H(E_pool))

```

## 10 Bob Verifies Alice's Identity (Two-Channel Design)

This is the two-channel design in action. Channel 1 (Ethereum): Bob decrypts  $E_{\text{pool}}$  with his secret key to recover the message point  $M$ . Channel 2 (Holochain): Bob receives Alice's `identity_data` and verifies the binding  $H(\text{identity\_data}) \cdot G = M$ . Neither channel alone is sufficient – the Ethereum ciphertext proves *which* identity was registered, while the Holochain data provides the *human-readable content*. A counter-example shows that changing even one character of `identity_data` in transit (e.g., "Alice" to "Mallory") produces a completely different point that doesn't match  $M$ , so Bob detects tampering immediately.

```

# === Bob's side ===

# Channel 1 (Ethereum proof chain): Bob decrypts E_bob
M_from_bob = add(C_for_bob, neg(mul(R_for_bob, sk_bob)))
print("Bob decrypts E_bob from on-chain IdentityExchange event:")
print(f" M = C' - sk_bob * R' = {pt(M_from_bob)}")

# Channel 2 (Holochain data channel): Bob receives identity_data
received_identity_data = alice_identity_data # <-- via Holochain, encrypted for Bob
print(f"\nBob receives identity_data via Holochain:")
print(f"  '{received_identity_data[:60]}...')

# Verify binding: H(identity_data) * G == M
m_check = keccak(received_identity_data)
M_check = mul(G1, m_check)
assert eq(M_check, M_from_bob), "Identity binding FAILED"

print(f"\nTwo-channel verification:")
print(f" m = H(identity_data) = {m_check}")
print(f" m * G = {pt(M_check)}")
print(f" M      = {pt(M_from_bob)}")
print(f" H(identity_data) * G == M : True")
print(f"\n==> Bob has cryptographic proof that the identity_data")
print(f"      he received is the same identity the issuer certified")
print(f"      and Alice registered on-chain.")

Bob decrypts E_bob from on-chain IdentityExchange event:
  M = C' - sk_bob * R' = (739132..., 937433...)

Bob receives identity_data via Holochain:
  '{"date_of_birth":"1992-03-15","epoch":42,"family_name":"John...'}

Two-channel verification:
  m = H(identity_data) = 6446313962173165014188361468977035479170665192834402473818305319846425860127
  m * G = (739132..., 937433...)
  M      = (739132..., 937433...)
  H(identity_data) * G == M : True

==> Bob has cryptographic proof that the identity_data
      he received is the same identity the issuer certified
      and Alice registered on-chain.

```

**How to read the output:** The coordinate fragments of  $m * G$  and  $M$  (decrypted from on-chain) must be identical. A counter-example follows showing what Bob sees if the Holochain data is tampered.

## 10.1 Counter-example: Tampered Identity Data

If a man-in-the-middle alters the identity data in transit, Bob detects it immediately – the hash no longer matches the on-chain point:

```
# Someone tampers with the identity data in transit
tampered = alice_identity_data.replace("Alice", "Mallory")
m_tampered = keccak(tampered)
M_tampered = mul(G1, m_tampered)

print(f"Tampered identity_data: ...given_name: 'Mallory'...")
print(f"  H(tampered) * G = {pt(M_tampered)}")
print(f"  M (from chain) = {pt(M_from_bob)}")
print(f"  Match: {eq(M_tampered, M_from_bob)}")
print(f"  (Bob rejects: the off-chain data doesn't match the on-chain point.)")
```

```
Tampered identity_data: ...given_name: 'Mallory'...
H(tampered) * G = (722726..., 633711...)
M (from chain) = (739132..., 937433...)
Match: False
(Bob rejects: the off-chain data doesn't match the on-chain point.)
```

## 11 Receipt Construction

After the transfer, both parties hold each other's plaintext `identity_data` (Alice has Bob's from the public IdentityRegistry; Bob has Alice's from Holochain) and the transfer metadata from on-chain events. Either party can independently construct a canonical receipt – deterministic JSON with sorted keys and no whitespace – and compute its `keccak256` hash. Alice's and Bob's independently constructed receipts produce identical hashes without coordination. This canonical form is what makes dispute resolution possible: both parties hold the same evidence, and neither can alter their copy without breaking the hash.

```
# Both parties construct the same canonical receipt
def canonical_receipt(sender_id, receiver_id, amount, tx_hash, block):
    return json.dumps({
        "sender": json.loads(sender_id),
        "receiver": json.loads(receiver_id),
        "amount": amount,
        "tx_hash": tx_hash,
        "block": block,
    }, sort_keys=True, separators=(',', ':'))

# Stand-in transaction data (would come from on-chain Transfer event)
tx_hash = "0xabcd1234567890"
block = 19400000
amount = 500

# Alice constructs her copy
receipt_alice = canonical_receipt(
    alice_identity_data, bob_identity_data, amount, tx_hash, block)

# Bob constructs his copy
receipt_bob = canonical_receipt(
    alice_identity_data, bob_identity_data, amount, tx_hash, block)
```

```

h_alice = keccak(receipt_alice)
h_bob   = keccak(receipt_bob)

print("Receipt construction (independent, canonical):")
print(f" Alice's H(receipt) = {h_alice}")
print(f" Bob's   H(receipt) = {h_bob}")
print(f" Match: {h_alice == h_bob}")
print(f"\nReceipt stored (encrypted) on each party's Holochain source chain.")
print(f"The receipt hash anchors the off-chain record to the on-chain transfer.")

Receipt construction (independent, canonical):
Alice's H(receipt) = 21535858141339728776948445993719093154923735092821067258494283187515610795617
Bob's   H(receipt) = 21535858141339728776948445993719093154923735092821067258494283187515610795617
Match: True

Receipt stored (encrypted) on each party's Holochain source chain.
The receipt hash anchors the off-chain record to the on-chain transfer.

```

## 12 Summary: What Each Party Knows

After a complete Private-to-Public transfer, the information landscape is sharply partitioned. Eve (a passive on-chain observer) learns nothing about Alice's identity from any public artifact. Bob learns Alice's identity only because Alice explicitly authorized it via `approve`. The issuer can identify Alice given  $M$ , but cannot link Alice's accounts from on-chain data alone. No party – not even the issuer – can link Alice's rerandomized signature  $\sigma'$  back to the original  $\sigma$ .

```

[
  ["Artifact", "Alice", "Bob", "Eve", "Issuer"],
  None,
  ["identity_data (Alice)", "YES (owns it)", "YES (Holochain)", "NO", "YES (issued it)"],
  ["identity_data (Bob)", "YES (public)", "YES (owns it)", "YES (public)", "YES (issued it)"],
  ["m = H(identity_data)", "YES", "YES (computes)", "NO", "YES"],
  ["M = m*G", "YES", "YES (decrypts)", "NO", "NO"],
  ["E_alice (registered)", "YES", "sees on-chain", "sees on-chain", "NO"],
  ["E_bob (re-encrypted)", "YES (created)", "YES (decrypts)", "sees on-chain", "NO"],
  ["sk_alice", "YES", "NO", "NO", "NO"],
  ["sk_bob", "NO", "YES", "NO", "NO"],
  ["PS signature sigma", "YES", "NO", "NO", "YES (created)"],
  ["sigma' (rerandomized)", "YES", "sees on-chain", "sees on-chain", "UNLINKABLE"],
  ["Chaum-Pedersen proof", "YES (created)", "verified on-chain", "sees on-chain", "N/A"],
  ["Full receipt", "YES", "YES", "NO", "NO"],
  ["Link sigma to sigma'", "NO (stat. unlinkable)", "NO", "NO", "NO"],
]

```

## 13 Appendix: The Identity Fountain – Unlinkability Demonstration

Alice derives a second credential from the same PS signature – simulating a "savings" account alongside her "checking" account. We verify that the second credential is a valid PS signature on the same  $m$ , then compare all public artifacts (rerandomized signature, public key, ElGamal ciphertext) between the two accounts. Every coordinate fragment is completely different: no shared structure, no correlation. This is *statistical* unlinkability (information-theoretic, not merely computational) – meaning no algorithm, regardless of computing power, can determine that these two accounts belong to the same person from on-chain data alone. Both credentials verify under the same issuer public key, confirming they represent certified identities, but the issuer itself cannot link them without access to a counterparty's decrypted  $M$ .

| Artifact                       | Alice                 | Bob               | Eve           | Issuer          |
|--------------------------------|-----------------------|-------------------|---------------|-----------------|
| identity_data (Alice)          | YES (owns it)         | YES (Holochain)   | NO            | YES (issued it) |
| identity_data (Bob)            | YES (public)          | YES (owns it)     | YES (public)  | YES (issued it) |
| $m = H(\text{identity\_data})$ | YES                   | YES (computes)    | NO            | YES             |
| $M = m * G$                    | YES                   | YES (decrypts)    | NO            | NO              |
| E_alice (registered)           | YES                   | sees on-chain     | sees on-chain | NO              |
| E_bob (re-encrypted)           | YES (created)         | YES (decrypts)    | sees on-chain | NO              |
| sk_alice                       | YES                   | NO                | NO            | NO              |
| sk_bob                         | NO                    | YES               | NO            | NO              |
| PS signature sigma             | YES                   | NO                | NO            | YES (created)   |
| sigma' (rerandomized)          | YES                   | sees on-chain     | sees on-chain | UNLINKABLE      |
| Chaum-Pedersen proof           | YES (created)         | verified on-chain | sees on-chain | N/A             |
| Full receipt                   | YES                   | YES               | NO            | NO              |
| Link sigma to sigma'           | NO (stat. unlinkable) | NO                | NO            | NO              |

```
# === Alice creates a SECOND account (e.g., "savings") ===
# Same m, same sigma, but fresh rerandomization and fresh keys

t_rerand_2 = rand()
sigma_p2_1 = mul(sigma[0], t_rerand_2)
sigma_p2_2 = mul(sigma[1], t_rerand_2)
sigma_p2 = (sigma_p2_1, sigma_p2_2)

sk_alice_2 = rand()
pk_alice_2 = mul(G1, sk_alice_2)

r_enc_2 = rand()
R_alice_2 = mul(G1, r_enc_2)
C_alice_2 = add(M_alice, mul(pk_alice_2, r_enc_2))
E_alice_2 = (R_alice_2, C_alice_2)

# Verify: sigma_p2 is a valid PS signature on the same m
lhs2 = pairing(add(IPK_X, mul(IPK_Y, m_alice)), sigma_p2[0])
rhs2 = pairing(G2, sigma_p2[1])
assert lhs2 == rhs2, "Second credential PS check FAILED"

print("Second credential derived (same identity, fresh account):")
print(f" pk_alice_2 = {pt(pk_alice_2)}")
print(f" E_alice_2 = (R, C) -- independent ciphertext")
print(f" sigma'_2 = {pt(sigma_p2[0])}")
print()
print("Unlinkability check:")
print(f" sigma'_1 (acct 1) = {pt(sigma_p[0])}")
print(f" sigma'_1 (acct 2) = {pt(sigma_p2[0])}")
print(f" pk (acct 1) = {pt(pk_alice)}")
print(f" pk (acct 2) = {pt(pk_alice_2)}")
print(f" E.R (acct 1) = {pt(R_alice)}")
print(f" E.R (acct 2) = {pt(R_alice_2)}")
print()
print(" All values are independent. No shared structure.")
print(" PS rerandomization is STATISTICALLY unlinkable --")
print(" not merely computationally hard, but information-")
print(" theoretically impossible to correlate.")
print()
print(" Both credentials verify under the same issuer public key,")
print(" but zero bits of mutual information exist between them.")

Second credential derived (same identity, fresh account):
pk_alice_2 = (505772..., 475564...)
E_alice_2 = (R, C) -- independent ciphertext
```

```
sigma'_2 = (568028..., 666324...)
```

Unlinkability check:

```
sigma'_1 (acct 1) = (435556..., 806524...)
sigma'_1 (acct 2) = (568028..., 666324...)
pk (acct 1)      = (587319..., 958871...)
pk (acct 2)      = (505772..., 475564...)
E.R (acct 1)     = (535490..., 86183...)
E.R (acct 2)     = (553875..., 341464...)
```

All values are independent. No shared structure.  
PS rerandomization is STATISTICALLY unlinkable --  
not merely computationally hard, but information-  
theoretically impossible to correlate.

Both credentials verify under the same issuer public key,  
but zero bits of mutual information exist between them.

**How to read the output:** Compare the coordinate fragments between "acct 1" and "acct 2" for  $\sigma'_1$ , pk, and E.R. They must all be completely different (random-looking). If any pair matched, the unlinkability property would be broken.

## 14 Appendix: Small-Group Proofs

The BN254 examples above prove the cryptography at production scale, but 77-digit numbers resist hand-verification. These demonstrations use modulus  $p = 23$  working in an order-11 subgroup where every value fits in two digits.

| EC (additive)           | Modular (multiplicative)           |
|-------------------------|------------------------------------|
| Scalar mult $m \cdot G$ | Exponentiation $g^m \bmod p$       |
| Point addition $P + Q$  | Multiplication $P \cdot Q \bmod p$ |

Schnorr proofs require prime-order groups, so we use  $g = 4$ , which generates the order-11 subgroup (the quadratic residues mod 23). All exponent arithmetic is mod  $q = 11$ .

### 14.1 Setup and ElGamal

ElGamal encryption of a group element  $M = g^m$  under public key  $pk = g^{sk}$  with randomness  $r$ :

$$E = (R, C) = (g^r, M \cdot pk^r) \pmod{p}$$
$$\text{Decrypt: } M = C \cdot (R^{sk})^{-1} \pmod{p}$$

```
import hashlib

p = 23; q = 11; g = 4      # safe prime, prime-order subgroup

def mod_exp(base, exp, mod):
    result = 1
    base = base % mod
    while exp > 0:
        if exp & 1:
            result = (result * base) % mod
        base = (base * base) % mod
        exp >>= 1
    return result

def mod_inv(a, mod):
```

```

def egcd(a, b):
    if a == 0: return b, 0, 1
    g, x, y = egcd(b % a, a)
    return g, y - (b // a) * x, x
_, x, _ = egcd(a % mod, mod)
return x % mod

def fs_hash(*args):
    """Fiat-Shamir: H(args...) mod q"""
    h = hashlib.sha256()
    for a in args:
        h.update(str(a).encode())
    return int(h.hexdigest(), 16) % q

# Identity
m = 3; M = mod_exp(g, m, p)

# Keys
sk_a = 5; pk_a = mod_exp(g, sk_a, p)
sk_b = 7; pk_b = mod_exp(g, sk_b, p)

# Encrypt for Alice (r_a=2) and Bob (r_b=9)
r_a = 2; R_a = mod_exp(g, r_a, p)
C_a = (M * mod_exp(pk_a, r_a, p)) % p

r_b = 9; R_b = mod_exp(g, r_b, p)
C_b = (M * mod_exp(pk_b, r_b, p)) % p

# Decrypt both
dec_a = (C_a * mod_inv(mod_exp(R_a, sk_a, p), p)) % p
dec_b = (C_b * mod_inv(mod_exp(R_b, sk_b, p), p)) % p

[
    ["", "m", "M=g^m", "sk", "pk", "r", "R=g^r", "C=M*pk^r", "Decrypted"],
    None,
    ["Alice", m, M, sk_a, pk_a, r_a, R_a, C_a, dec_a],
    ["Bob", m, M, sk_b, pk_b, r_b, R_b, C_b, dec_b],
]

```

|       | m | M=g <sup>m</sup> | sk | pk | r | R=g <sup>r</sup> | C=M*pk <sup>r</sup> | Decrypted |
|-------|---|------------------|----|----|---|------------------|---------------------|-----------|
| Alice | 3 | 18               | 5  | 12 | 2 | 16               | 16                  | 18        |
| Bob   | 3 | 18               | 7  | 8  | 9 | 13               | 1                   | 18        |

Same message  $M = 18$  in both ciphertexts: (16,16) for Alice, (13,1) for Bob. Different randomness makes them unlinkable.

## 14.2 NIZK: Proof of Knowledge (Registration)

At registration, Alice proves she knows the plaintext  $m$  and randomness  $r$  inside her ElGamal ciphertext – without revealing either. (In the full protocol, a Pointcheval-Sanders pairing check additionally binds  $m$  to the issuer’s signature; this demonstration isolates the sigma-protocol core.)

Statement:  $R = g^r, C = g^m \cdot pk^r$

Commit:  $T_R = g^{k_r}, T_C = g^{k_m} \cdot pk^{k_r}$

Challenge:  $e = H(g, pk, R, C, T_R, T_C) \bmod q$

Respond:  $s_m = k_m - e \cdot m \pmod{q}, s_r = k_r - e \cdot r \pmod{q}$

Verify 1:  $g^{s_r} \cdot R^e \stackrel{?}{=} T_R$

Verify 2:  $g^{s_m} \cdot pk^{s_r} \cdot C^e \stackrel{?}{=} T_C$

```
# Prove knowledge of (m=3, r=2) in E_alice = (16, 16)
k_m = 1; k_r = 1 # random nonces
T_R = mod_exp(g, k_r, p) # 4
T_C = (mod_exp(g, k_m, p) * mod_exp(pk_a, k_r, p)) % p # 2

e = fs_hash(g, pk_a, R_a, C_a, T_R, T_C) # 8

s_m = (k_m - e * m) % q # 10
s_r = (k_r - e * r_a) % q # 7

chk1 = (mod_exp(g, s_r, p) * mod_exp(R_a, e, p)) % p
chk2 = (mod_exp(g, s_m, p) * mod_exp(pk_a, s_r, p) * mod_exp(C_a, e, p)) % p

print(f"Commit: T_R = {T_R}, T_C = {T_C}")
print(f"Challenge: e = {e}")
print(f"Respond: s_m = {s_m}, s_r = {s_r}")
print(f"Verify 1: g^{s_r} * R^e = {chk1:2} == T_R={T_R:2} {'PASS' if chk1 == T_R else 'FAIL'}")
print(f"Verify 2: g^{s_m} * pk^{s_r} * C^e = {chk2:2} == T_C={T_C:2} {'PASS' if chk2 == T_C else 'FAIL'}")

# Counter-example: Alice claims m'=5 (wrong identity)
m_fake = 5
s_m_fake = (k_m - e * m_fake) % q
chk2_f = (mod_exp(g, s_m_fake, p) * mod_exp(pk_a, s_r, p) * mod_exp(C_a, e, p)) % p
print(f"\nCounter-example (m'={m_fake}):")
print(f" Verify 2: {chk2_f} == T_C={T_C} {'PASS' if chk2_f == T_C else 'FAIL'}")
print(f" The ciphertext encrypts m={m}, not m'={m_fake}.")
print(f" No response value can bridge the gap.")

Commit: T_R = 4, T_C = 2
Challenge: e = 8
Respond: s_m = 10, s_r = 7
Verify 1: g^{s_r} * R^e = 4 == T_R= 4 PASS
Verify 2: g^{s_m} * pk^{s_r} * C^e = 2 == T_C= 2 PASS

Counter-example (m'=5):
Verify 2: 4 == T_C=2 FAIL
The ciphertext encrypts m=3, not m'=5.
No response value can bridge the gap.
```

**How it works:** The prover commits to random nonces  $k_m, k_r$  before seeing the challenge  $e$ . Only someone who knows the real  $m$  and  $r$  can produce responses that satisfy both equations for the specific  $e$  the Fiat-Shamir hash generates. Check 1 confirms knowledge of  $r$ ; Check 2 confirms knowledge of  $m$  and its consistency with the ciphertext. A wrong  $m'$  produces a wrong  $s_m$ , which ripples through the multi-exponentiation in Check 2.

### 14.3 Chaum-Pedersen: Proof of Same Plaintext (Approve)

At approve time, Alice re-encrypts her identity for Bob and proves both ciphertexts contain the same  $M$  – without revealing it. Three independent checks verify her key ownership, her knowledge

of the new randomness, and the binding constraint that the decrypted values match.

$$\text{Statement: } C_a \cdot (R_a^{sk_a})^{-1} = C_b \cdot (pk_b^{r_b})^{-1} \quad (\text{same } M)$$

$$\text{Commit: } T_1 = g^{k_1}, \quad T_2 = g^{k_2}, \quad T_3 = pk_b^{k_2} \cdot R_a^{-k_1}$$

$$\text{Challenge: } e = H(g, pk_a, pk_b, R_a, C_a, R_b, C_b, T_1, T_2, T_3)$$

$$\text{Respond: } s_1 = k_1 - e \cdot sk_a, \quad s_2 = k_2 - e \cdot r_b \quad (\text{mod } q)$$

$$\text{Verify 1: } g^{s_1} \cdot pk_a^e \stackrel{?}{=} T_1 \quad (\text{owns } pk_a)$$

$$\text{Verify 2: } g^{s_2} \cdot R_b^e \stackrel{?}{=} T_2 \quad (\text{knows } r_b)$$

$$\text{Verify 3: } pk_b^{s_2} \cdot R_a^{-s_1} \cdot (C_b/C_a)^e \stackrel{?}{=} T_3 \quad (\text{same } M)$$

```
# Prove E_alice=(16,16) and E_bob=(13,1) encrypt the same M
k1 = 1; k2 = 1
T1 = mod_exp(g, k1, p) # 4
T2 = mod_exp(g, k2, p) # 4
T3 = (mod_exp(pk_b, k2, p) * mod_exp(R_a, (q - k1) % q, p)) % p # 12

e_cp = fs_hash(g, pk_a, pk_b, R_a, C_a, R_b, C_b, T1, T2, T3) # 4

s1 = (k1 - e_cp * sk_a) % q # 3
s2 = (k2 - e_cp * r_b) % q # 9

v1 = (mod_exp(g, s1, p) * mod_exp(pk_a, e_cp, p)) % p
v2 = (mod_exp(g, s2, p) * mod_exp(R_b, e_cp, p)) % p
cb_ca = (C_b * mod_inv(C_a, p)) % p
v3 = (mod_exp(pk_b, s2, p) * mod_exp(R_a, (q - s1) % q, p)
      * mod_exp(cb_ca, e_cp, p)) % p

print(f"Commit: T1={T1}, T2={T2}, T3={T3}")
print(f"Challenge: e = {e_cp}")
print(f"Respond: s1={s1}, s2={s2}")
print(f"Verify 1 (owns pk_a): {v1:2} == T1={T1:2} {'PASS' if v1 == T1 else 'FAIL'}")
print(f"Verify 2 (knows r_b): {v2:2} == T2={T2:2} {'PASS' if v2 == T2 else 'FAIL'}")
print(f"Verify 3 (same M): {v3:2} == T3={T3:2} {'PASS' if v3 == T3 else 'FAIL'}")

# Counter-example: Alice encrypts M'=g^7=8 instead of M=g^3=18
m_bad = 7; M_bad = mod_exp(g, m_bad, p)
C_b_bad = (M_bad * mod_exp(pk_b, r_b, p)) % p
e_bad = fs_hash(g, pk_a, pk_b, R_a, C_a, R_b, C_b_bad, T1, T2, T3)
s1_b = (k1 - e_bad * sk_a) % q
s2_b = (k2 - e_bad * r_b) % q
v1_b = (mod_exp(g, s1_b, p) * mod_exp(pk_a, e_bad, p)) % p
v2_b = (mod_exp(g, s2_b, p) * mod_exp(R_b, e_bad, p)) % p
cb_ca_b = (C_b_bad * mod_inv(C_a, p)) % p
v3_b = (mod_exp(pk_b, s2_b, p) * mod_exp(R_a, (q - s1_b) % q, p)
      * mod_exp(cb_ca_b, e_bad, p)) % p

print(f"\nCounter-example (M'=g^{m_bad}={M_bad}, C_b'={C_b_bad}):")
print(f" Verify 1 (owns pk_a): {v1_b:2} == T1={T1:2} {'PASS' if v1_b == T1 else 'FAIL'}")
print(f" Verify 2 (knows r_b): {v2_b:2} == T2={T2:2} {'PASS' if v2_b == T2 else 'FAIL'}")
print(f" Verify 3 (same M): {v3_b:2} == T3={T3:2} {'PASS' if v3_b == T3 else 'FAIL'}")
print(f" Alice owns her key and knows r_b -- checks 1,2 pass.")
print(f" But E_alice encrypts M={M} while E_bob encrypts M'={M_bad}.")

Commit: T1=4, T2=4, T3=12
Challenge: e = 4
Respond: s1=3, s2=9
Verify 1 (owns pk_a): 4 == T1= 4 PASS
Verify 2 (knows r_b): 4 == T2= 4 PASS
```

Verify 3 (same M):        12 == T3=12  PASS

Counter-example (M'=g<sup>7</sup>=8, C\_b'=3):

Verify 1 (owns pk\_a):    4 == T1= 4  PASS

Verify 2 (knows r\_b):    4 == T2= 4  PASS

Verify 3 (same M):        1 == T3=12  FAIL

Alice owns her key and knows r\_b -- checks 1,2 pass.

But E\_alice encrypts M=18 while E\_bob encrypts M'=8.

**How it works:** Check 3 is the binding constraint. Expanding the verification equation, it passes only when  $C_a/R_a^{sk_a} = C_b/pk_b^{r_b}$  – i.e., decrypting **E\_alice** with Alice's key yields the same  $M$  as de-randomizing **E\_bob** with  $r_b$ . A different message breaks this equality, producing a mismatch that no choice of responses can hide, because the commitments were fixed before the challenge was known.