Alberta Buck – Identity and Cryptographic Receipts (DRAFT v0.1)

Perry Kundert

2026-04-08

Alberta Buck transactions carry provably authentic identity – without exposing it. Every participant holds an ElGamal ciphertext[1] of their identity, encrypted under their own public key, backed by a Pointcheval-Sanders rerandomizable signature[2] from the issuing authority. A citizen with multiple accounts derives an independent, unlinkable credential for each – rerandomizing the signature offline – yet every credential is independently verifiable as issued by the same authority. When a transaction requires identity exchange, the participant re-encrypts under the counterparty's key and produces a Chaum-Pedersen proof[3]: a compact, unforgeable guarantee that the re-encrypted ciphertext contains the same identity as the registered credential. No one – not the contract, not the network, not the government – sees the plaintext.

Per-transaction identity verification costs ~29,000 gas via Ethereum's alt_bn128 scalar multiplication precompiles – no pairing operations at transaction time. Credential registration costs ~200,000 gas (a one-time pairing-based verification of the Pointcheval-Sanders signature). No trusted setup. Proof generation is instantaneous.

This document specifies identity requirements for every Alberta Buck operation, details the ElGamal + Pointcheval-Sanders + Chaum-Pedersen architecture, and demonstrates through adversarial scenarios that the scheme defeats forgery, substitution, and collusion while keeping on-chain costs minimal. (PDF, Text)

[1] ElGamal Encryption. ElGamal, T. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms." IEEE Transactions on Information Theory, 1985. Semantic security (IND-CPA) under the Decisional Diffie-Hellman assumption. Alberta Buck uses ElGamal over the alt_bn128 curve (BN254), which is supported by Ethereum's EIP-196/197 precompiled contracts for efficient on-chain verification.

[2] Pointcheval-Sanders Signatures. Pointcheval, D. and Sanders, O. "Short Randomizable Signatures." CT-RSA 2016, LNCS 9610, pp. 111-126. Rerandomizable signatures: given a valid signature sigma on message m, anyone can compute sigma' = (t * sigma_1, t * sigma_2) that is a valid, statistically unlinkable signature on the same m. Verification uses a bilinear pairing equation, compatible with Ethereum's alt_bn128 (BN254) ecPairing precompile.

[3] Chaum-Pedersen Protocol. Chaum, D. and Pedersen, T.P. "Wallet Databases with Observers." CRYPTO 1992, LNCS 740, pp. 89-105. The Chaum-Pedersen protocol proves equality of discrete logarithms (equivalently, that two ElGamal ciphertexts encrypt the same message) in zero knowledge. Made non-interactive via the Fiat-Shamir heuristic.

Contents

1 Identity with Anonymity

The expectation to be able to transact Alberta BUCKs in complete freedom (no one else can prevent it) and anonymity (no one else can identify the parties) is paramount. This is consistent with the expectation that one is innocent until proven guilty. Asking permission from anyone before we can buy, sell or trade is unacceptable to any free people. Anything less turns a free citizen into a serf. If the government or some regulator can lock your account and freeze you on a park bench at will, how free are you?

The expectation that I can be certain of my counterparty's identity, and that it is *at least* as reliable as the group(s) issuing it is also critical. Millions of financial fraud phone calls flood Albertans' phones daily, pillaging our neighbours and grand-parents, each carrying the government issued identity of the Canadian phone number it "originates" from – under the cover of regulators. *Everyone* involved knows where the call is coming from (the termination contract with the VoIP carrier), who is carrying it into our phone system (the VoIP carrier's contract with phone system providers), and who they are calling - except for *you*. Allowing arbitrary hostile groups to ravage a community under the cover of darkness has been used throughout history to subjugate free people. If the government can issue identity documents to whomever they wish, and you are forced to trust them, how free are you?

The Alberta Buck enforces these principles: the freedom to *transact in privacy and peace*, and the confidence that your counterparty *is who they say they are*.

1.1 Revealing Identity while Preserving Privacy

If I am suspected of a crime, an Alberta court can compel me to decrypt the participants in BUCK transactions they suspect are related to that crime, and reveal the identity of counterparties I've transacted with. I may resist, and face the consequences. But the organizations that Alberta tasks with overseeing the authenticity of my Alberta issued identity must not be able to simply authorize decryption of my counterparties' identities! They can only vouch for and reveal the identities they've directly issued, just as I can vouch for my daughter's – they can't decrypt arbitrary participants in each transaction.

This is how the cryptography enforces it: each re-encrypted identity is an ElGamal ciphertext under the specific counterparty's public key. Only that counterparty's private key can recover the plaintext. The identity issuer cannot decrypt it – they signed the identity, not the ciphertext; the encryption is mine alone. The government cannot decrypt it. No registry operator, no Holochain node, no subpoena served on anyone other than a direct participant can break this. A court must compel a *specific participant* to decrypt a *specific transaction's counterparties'* identity data. Every individual's credential must be attacked separately. The cost to surveil is O(N), not O(1).

1.1.1 The Identity Fountain

A citizen with income, savings, inheritance, and walking-around-money accounts should not have those accounts linkable to each other by any outside observer. Ethereum balances

are public; if all of Alice's accounts trace back to the same identity credential, her entire financial life is exposed to anyone who notices the link.

The identity fountain solves this. It gives every citizen the ability to derive unlimited, unlinkable identity credentials – each independently verifiable as issued by the same authority – from a single KYC ceremony. The construction uses Pointcheval-Sanders (PS) rerandomizable signatures[2] over the alt_bn128 curve.

1. Issuance: The Issuer Signs Once

   During the KYC ceremony, the issuer:

   (a) Verifies Alice's identity (documents, biometrics, whatever the jurisdiction requires).

   (b) Computes the identity scalar `m = H(identity_data)`.

   (c) Picks a random point `h` in `G_1` and computes the PS signature:
   `sigma = (h, (x + m * y) * h)`
   where `(x, y)` is the issuer's secret key.

   (d) Gives Alice the identity scalar `m` and signature `(sigma_1, sigma_2)`.

   (e) Publishes the issuance event to the issuer's Holochain source chain.

   The issuer's public key `(X, Y) = (x * g_2, y * g_2)` in `G_2` is registered in the TrustedIssuersRegistry on Ethereum. The issuer is now done – permanently, if desired.

2. Derivation: Alice Generates Credentials Offline

   For each new Ethereum account, Alice's Holochain wallet:

   (a) Picks a random scalar `t` and rerandomizes the PS signature:
   `sigma' = (t * sigma_1, t * sigma_2)`
   This `sigma'` is a valid PS signature on the same `m` under the same issuer public key – but is *statistically unlinkable* to `sigma` or to any other rerandomization. No algorithm can correlate `sigma'` back to `sigma`.

   (b) Generates a fresh identity key pair `(sk_new, pk_new)` on alt_bn128.

   (c) Encrypts the identity under the new key using ElGamal:
   `E_new = (r * G, m * G + r * pk_new)`

   (d) Computes a non-interactive zero-knowledge proof `pi` that `E_new` encrypts the same `m` that `sigma'` signs:

```
"I know (m, t, r) such that:
  a) e(sigma'_1, X + m * Y) = e(sigma'_2, g_2)   -- valid PS signature
  b) C = m * G + r * pk_new                       -- ElGamal encryption of m
  c) R = r * G                                    -- consistent randomness"
```

   (e) Packages `(pk_new, E_new, sigma', pi)` for registration.

This derivation runs entirely on Alice's device. She can pre-generate a hundred credentials in a batch and store them in her Holochain wallet, pulling one out whenever she creates a new account.

3. Registration: One-Time On-Chain Verification

   Alice registers the derived credential in the IdentityRegistry:

```
identityRegistry.registerCredential(
    alice_new_address,
    pk_new,          // identity public key
    E_new,           // ElGamal ciphertext (4 uint256)
    sigma_prime,     // rerandomized PS signature (2 G_1 points)
    pi               // NIZK proof linking sigma' to E_new
)
```

   The contract verifies the PS signature and NIZK proof using the `ecPairing` precompile (address `0x08`), `ecMul`, and `ecAdd`. Gas cost: ~200,000 gas, paid once per account.

   After registration, `isVerified(alice_new_address) == true`. The credential is stored on-chain as an ElGamal ciphertext and an identity public key. No subsequent operation consults the PS signature again; per-transaction verification is pure Chaum-Pedersen (~29,000 gas).

4. What the World Sees

   Alice creates five accounts over two years. Each registers a credential `(pk_i, E_i, sigma'_i, pi_i)`. An observer can verify:

   - Each `sigma'_i` was produced by the known issuer – the pairing check passes.
   - Each `E_i` encrypts the identity that `sigma'_i` signs – the proof `pi_i` confirms it.
   - Whether any two credentials belong to the same person – **no**.

   PS rerandomization is *statistically* unlinkable – not merely computationally hard to correlate, but information-theoretically impossible. The credentials contain zero bits of mutual information. Alice's inheritance account, her walking-around-money account, her savings account – each carries the issuer's stamp of authenticity, each is cryptographically invisible to the others.

5. Epoch-Based Credential Renewal

   If the issuer revokes Alice's identity (fraud, death, court order), all her derived credentials must be invalidated – without linking them.

   Credentials carry an expiry epoch (e.g., one year). The issuer publishes epoch keys; credential derivation requires the current epoch key. Revocation means the

issuer refuses to provide Alice's next epoch key. Expired credentials are rejected at `isVerified` time. Alice must periodically re-derive from a fresh issuance – giving the issuer a natural checkpoint for continued eligibility.

This fits Alberta Buck's model: the issuer remains the trust anchor for "is this person still in good standing," which is the real-world question regardless of the cryptography.

1.2 Accounts Belong to Someone

I may be a citizen of Alberta and have a minor child. My child's identity is vouched by me, and my identity is vouched by Alberta. I have several accounts associated with encrypted copies of my identity (income, savings, etc) and several unencrypted accounts (walking-around money). I can transfer money from my savings to my walking-around account (anonymously, via my BUCK account at ATB).

My encrypted savings account identity has been authorized by ATB to send funds. My walking-around money account has a plaintext identity, so Starbucks' BUCK account doesn't need to pre-authorize incoming transactions – it accepts BUCK transfers from any BUCK account with a public identity.

This just cuts out the middleman, and gives every citizen what government and institutions already have: knowledge of the participants in transactions with public accounts. If I prefer, I can set up a Starbucks account, and ask them to authorize me to send money to them. Again, allowing with BUCKs what is already allowed with a pre-loaded Starbucks card – private transactions with a public account. Starbucks and Visa both know who paid to fund the Visa card, and who funded their Starbucks account with that card. Now, citizens can set up similar arrangements and retain that knowledge amongst themselves.

If a court compels one of the participants (for example, as the result of an arrest in a sting operation against some members of a group performing some illicit activity), then that person's transactions with other members of the group can be discovered. Otherwise: *financial activity remains confidential, at the sole choice of the participants.*

The expectation that any government or bank functionary – or any random attacker exploiting government or bank incompetence – can know the details of a citizen's financial activities is unacceptable. The attack cost must be O(N), not O(1): every individual's credential must be attacked separately, not extracted from one compromised custodian.

1.3 Counterparties Exchange Identities

Each transaction is between parties identified to each other by provably untampered copies of their identity, encrypted by each party specifically for that counterparty.

The identity credential lifecycle:

1. **Issuance**: A trusted issuer (the Alberta government, ATB Financial, etc.) verifies Alice's identity through a KYC ceremony and produces a Pointcheval-Sanders signature on her identity scalar `m = H(identity_data)`. Alice receives `(m, sigma)`

and stores it in her Holochain wallet. (See The Identity Fountain for the full PS construction.)

2. **Derivation**: For each Ethereum account, Alice's wallet rerandomizes the PS signature, generates a fresh identity key pair, encrypts her identity under the new key using ElGamal, and produces a NIZK proof linking the rerandomized signature to the ciphertext. This runs entirely offline.

3. **Registration**: Alice registers the derived credential `(pk, E_alice, sigma', pi)` in the on-chain IdentityRegistry. The contract verifies the PS signature and NIZK proof via the `ecPairing` precompile (`200,000 gas, one-time). After registration, ~isVerified(alice) == true`.

4. **Re-encryption**: When Alice must identify herself to counterparty Bob (e.g., during an `approve` call on the BUCK contract), she decrypts her credential locally (she knows `sk_alice`) and re-encrypts the identity under Bob's public key with fresh randomness:

   `E_bob = (r' * G, r' * pk_bob + M)`

   where `M = m * G` is the identity message point. She produces a Chaum-Pedersen proof[3] `pi` that `E_bob` encrypts the same point `M` as her registered `E_alice` – without revealing `m`.

5. **Verification**: The BUCK smart contract verifies:

   - `E_alice` is read from the IdentityRegistry (not from Alice's calldata)
   - `pi` proves `E_bob` encrypts the same plaintext as `E_alice`

   Verification requires only 3-4 elliptic curve scalar multiplications (~29,000 gas via the alt_bn128 precompiles). No pairing operations at transaction time. The PS signature was verified once, at registration, and is never consulted again.

6. **Decryption**: Only Bob can decrypt `E_bob` (he has `sk_bob`). Only Alice can decrypt `E_alice` (she has `sk_alice`). The issuer cannot decrypt either – they signed Alice's identity scalar, they never encrypted under any key. No third party can decrypt any re-encrypted copy.

There is no need for Proxy Re-Encryption (PRE), because Alice is online when she calls `approve` and can perform the re-encryption directly. The Chaum-Pedersen proof is the trust anchor at transaction time: it is a mathematical guarantee – unforgeable under the discrete logarithm assumption – that Alice re-encrypted her genuine, registered identity and not a substitute. The PS signature is the trust anchor at registration time: it guarantees the credential was issued by an authorized issuer.

1.4 ElGamal and Pointcheval-Sanders Over a Mandated Curve Group

Alberta Buck mandates that all identity credentials, all participant key pairs, all re-encryption operations, and all issuer signatures use the alt_bn128 elliptic curve (BN254)[1] – the curve supported by Ethereum's precompiled contracts:

- `ecAdd` (address `0x06`): point addition, 150 gas
- `ecMul` (address `0x07`): scalar multiplication, 6,000 gas
- `ecPairing` (address `0x08`): bilinear pairing check, 34,000 gas per pair + 45,000 base

This mandate is a design strength, not a limitation:

- **Uniform proof verification**: every Chaum-Pedersen proof and every Pointcheval-Sanders signature verification uses the same curve group, so a single on-chain verifier handles all identity operations.
- **Precompile efficiency**: per-transaction Chaum-Pedersen verification uses `ecMul` / `ecAdd` (29,000 gas). One-time credential registration uses ~`ecPairing` for PS signature verification (~200,000 gas).
- **No cross-group compatibility issues**: since all participants, issuers, and credentials use the same curve, re-encryption between any two participants is always algebraically compatible.

Participant key pairs for identity purposes are distinct from their Ethereum signing keys (which use secp256k1). The identity key pair is generated during credential derivation (not during the KYC ceremony – the KYC ceremony produces only the PS signature). The public key is registered in the IdentityRegistry alongside the participant's Ethereum address.

When Alice re-encrypts for N different counterparties, each re-encryption uses independent randomness. The resulting ciphertexts are IND-CPA secure: computationally indistinguishable from random curve points. An observer who collects all N ciphertexts cannot determine that they encrypt the same plaintext (without access to a counterparty's private key).

1.5 Chaum-Pedersen Proof of Re-Encryption Correctness

The Chaum-Pedersen protocol[3] is a sigma protocol (Schnorr family) that proves two ElGamal ciphertexts encrypt the same message point, without revealing the message. Made non-interactive via the Fiat-Shamir heuristic, the proof is compact (~128 bytes: two scalars and two curve points) and verification requires only 3-4 multi-scalar multiplications.

1.5.1 Setup

```
Registered credential (verified at registration via PS signature + NIZK):
  E_alice = (R_a, C_a) = (r_a * G,  r_a * pk_alice + M)
  where M = m * G  (identity encoded as a curve point, m = H(identity_data))
  Rerandomized PS signature sigma' verified at registration (not consulted again)

Alice's re-encryption for Bob:
  E_bob = (R_b, C_b) = (r_b * G,  r_b * pk_bob + M)
  using the SAME message point M
```

1.5.2 What Alice Proves (Non-Interactively)

```
"I know (sk_alice, r_b) such that:
  1. pk_alice = sk_alice * G                    -- I own this public key
  2. R_b = r_b * G                              -- I know E_bob's randomness
  3. C_a - sk_alice * R_a = C_b - r_b * pk_bob  -- same message point M"
```

Statement 3 is the core. The left side decrypts `E_alice` (only Alice can compute this, since she knows `sk_alice`), yielding the message point `M`. The right side extracts `M` from `E_bob` (only Bob could alternatively compute this with `sk_bob`). If Alice encrypted different data `I_fake`, then `I * G != I_fake * G` and no valid proof exists – it is not merely hard to compute, it is mathematically impossible under the discrete logarithm assumption.

1.5.3 Verification (What the Contract Computes)

```
Given public inputs:
  E_alice = (R_a, C_a)     -- from IdentityRegistry (PS-verified at
                              registration, read from storage -- NOT
                              supplied by Alice)
  E_bob   = (R_b, C_b)     -- from Alice's approve() calldata
  pk_alice                 -- from IdentityRegistry
  pk_bob                   -- from IdentityRegistry
  proof   = (e, s1, s2)    -- Fiat-Shamir challenge and responses

Verify:
  Recompute challenge e from commitments
  Check consistency of s1, s2 against e, the public keys, and ciphertexts
  3-4 ecMul + 3-4 ecAdd operations
```

Critical implementation requirement: `E_alice` is **read from on-chain storage** (the IdentityRegistry), not supplied by Alice as calldata. This anchors the proof to the registered credential. If Alice could supply her own `E_alice`, she could substitute a fake credential and produce a valid proof against it.

1.5.4 Gas Cost

1. Per-Transaction (Chaum-Pedersen Verification at `approve`)

Operation	Count	Gas each	Total
`ecMul` (precompile)	4	6,000	24,000
`ecAdd` (precompile)	4	150	600
`keccak256` (Fiat-Shamir)	1	36	36
Calldata (128 bytes)			2,048
Storage read (`E_alice`)	1	2,100	2,100
Total			**~29,000 gas**

The `approve` call costs `29,000 gas for the identity proof plus the standard ERC-20 approve cost (~46,000 gas), totaling ~75,000 gas. This is paid once per counterparty pair. Subsequent ~transfer` and `transferFrom` calls emit only minimal receipt events (addresses + opaque hashes, ~2,000 gas) with no additional proof work.

2. One-Time (PS Signature + NIZK Verification at Registration)

Operation	Count	Gas each	Total
`ecPairing` (precompile)	3-4	34,000	147,000
`ecPairing` base cost	1	45,000	45,000
`ecMul` (NIZK verification)	3-4	6,000	24,000
`ecAdd` (NIZK verification)	3-4	150	600
Total			**~200,000 gas**

This one-time registration cost is comparable to deploying a small contract. For an account that will hold meaningful value and transact many times, it is negligible. Compare: Groth16 SNARK verification requires similar pairing operations (~200,000-300,000 gas) but must be performed *per transaction* if used for re-encryption proofs.

2 Attacks and Defenses

The following scenarios demonstrate that the ElGamal Chaum-Pedersen proof scheme defeats identity fraud, even when participants are actively criminal. In each scenario we show what the attacker attempts, why it fails mathematically, and what the on-chain verifier sees.

The trust chain in every scenario is:

```
Issuer PS-signs identity m              --> identity is genuine
Alice derives (E_alice, sigma', pi)     --> credential is unlinkable
PS + NIZK verified at registration      --> credential is issuer-authorized
```

```
E_alice stored in registry              --> contract reads the genuine credential
Chaum-Pedersen proof verified           --> E_bob encrypts same identity as E_alice
Only Bob can decrypt E_bob              --> identity revealed only to counterparty
```

Breaking any link requires breaking either the q-SDH assumption (PS signature unforgeability), the discrete logarithm assumption on alt_bn128 (Chaum-Pedersen soundness, ElGamal security), or ECDSA (Ethereum transaction authenticity) – all standard hardness assumptions underpinning Ethereum's own security.

2.1 Attack 1: Alice Substitutes a False Identity for Bob

2.1.1 Scenario

Alice is a sanctioned individual. She completed KYC under her real identity and received a legitimate PS-verified credential `E_alice` containing her true identity `I`. She now wants to transact with Bob but wants Bob (and any future court) to believe she is someone else – say, her clean associate Carol.

Alice constructs a fake re-encryption using Carol's identity `I_fake`:

```
E_bob_fake = (r' * G,  r' * pk_bob + I_fake * G)
```

She submits `E_bob_fake` to `approve()` and attempts to produce a Chaum-Pedersen proof tying it to her registry credential.

2.1.2 Why It Fails

The Chaum-Pedersen verification checks:

```
C_a - sk_alice * R_a  ==  C_b - r' * pk_bob
```

Expanding the left side (from the registered `E_alice` in the registry):

```
(r_a * pk_alice + I * G) - sk_alice * (r_a * G)
  = r_a * sk_alice * G + I * G - sk_alice * r_a * G
  = I * G
```

Expanding the right side (from Alice's fake `E_bob`):

```
(r' * pk_bob + I_fake * G) - r' * pk_bob
  = I_fake * G
```

The proof requires `I * G == I_fake * G`, which requires `I == I_fake`. Since `I != I_fake` (Alice is not Carol), these are distinct curve points. Under the discrete logarithm assumption, Alice **cannot produce valid proof scalars (s1, s2)** that satisfy the verification equations. The proof does not exist – not "hard to find," but mathematically non-existent.

2.1.3 What the Contract Sees

`approve()` reverts: "Re-encryption proof invalid." The transaction fails on-chain. Alice cannot complete the identity handshake with false data. The revert is indistinguishable from a malformed transaction – no information about Alice's intent leaks.

2.1.4 Cost to Alice

She wasted gas on a failed transaction. Her address is still associated with her real registered credential in the registry. Nothing has changed.

2.2 Attack 2: Alice Forges an Issuer Credential

2.2.1 Scenario

Alice never completed KYC, or completed KYC but wants a credential with different identity data. She picks a fabricated identity scalar `m_fake = H(fake_data)`, generates a random PS-like signature, creates an ElGamal ciphertext, and fabricates a NIZK proof:

```
m_fake       = H("Carol Smith, Alberta, ...")
sigma_fake   = (random_h, random_s)          -- not a valid PS signature
E_alice_fake = (r * G,  m_fake * G + r * pk_alice)
pi_fake      = ...                           -- fabricated NIZK
```

She attempts to register this credential in the IdentityRegistry.

2.2.2 Why It Fails

The IdentityRegistry verifies the Pointcheval-Sanders signature against trusted issuer public keys registered in the TrustedIssuersRegistry (part of the ERC-3643 framework[4]). Registration checks the pairing equation:

```
e(sigma'_1, X + m * Y) == e(sigma'_2, g_2)
```

where `(X, Y)` is the trusted issuer's public key in `G_2`. Alice does not know the issuer's secret key `(x, y)`, so she cannot produce a `sigma'` that satisfies this equation for any `m` – let alone for `m_fake`. The PS signature is unforgeable under the q-SDH assumption.

Even if Alice could somehow produce a valid-looking PS signature (she cannot), the NIZK proof `pi` must prove that the ElGamal ciphertext `E_alice` encrypts the same `m` that the PS signature covers. Forging the proof requires breaking the discrete logarithm assumption.

Two independent barriers must both be broken. Neither has been broken in the history of elliptic curve cryptography.

[4]ERC-3643 (T-REX): Token for Regulated EXchanges. The on-chain identity framework (IdentityRegistry, TrustedIssuersRegistry, ClaimTopicsRegistry) used by Alberta Buck for credential management. `https://eips.ethereum.org/EIPS/eip-3643`

2.2.3 What the Contract Sees

`registerCredential()` reverts: "Invalid issuer signature." Alice is not registered; `isVerified(alice) == false`. She cannot participate in any BUCK operation requiring identity.

2.3 Attack 3: Alice and Complicit Bob Launder With False Identities

2.3.1 Scenario

Alice and Bob are both criminals. Both have legitimate PS-verified credentials (they passed KYC under their real identities). They want to transact with each other, but want the encrypted identity records to show *different* people – so that if a court later compels Bob to decrypt Alice's identity blob, it points to an innocent third party.

They agree: Alice will encrypt a false identity `I_fake` for Bob and submit it. Bob will claim (if asked) that the decrypted identity is genuine.

2.3.2 Why It Fails

Alice's `approve()` call must pass Chaum-Pedersen verification. The contract reads `E_alice` from the IdentityRegistry – not from Alice's calldata.

```
// Inside approve():
E_alice = identityRegistry.getCredential(msg.sender);
require(chaumPedersenVerify(E_alice, E_bob_submitted, proof));
```


Alice cannot substitute a different `E_alice` because the contract reads it from storage. She cannot produce a valid Chaum-Pedersen proof linking her fake `E_bob` to the real `E_alice` because the message points differ (same math as Attack 1).

Bob's willingness to accept false data is irrelevant. The proof was verified against the registered credential *before* `E_bob` was stored. If `approve()` succeeded, `E_bob` necessarily contains Alice's real identity. When a court later compels Bob to decrypt, he produces Alice's true identity – regardless of any prior agreement between them.

Bob cannot produce a *different* plaintext from the same `E_bob` (ElGamal decryption is deterministic given the private key). Bob cannot claim the `E_bob` is unrelated to Alice (the on-chain `IdentityExchange` event records that Alice submitted it during `approve`, and the proof verification succeeded).

2.3.3 What the Contract Sees

If Alice submits false data: `approve()` reverts ("Re-encryption proof invalid") – same as Attack 1.

If Alice submits real data (because she has no choice): `approve()` succeeds, and `E_bob` provably contains her real identity. The laundering scheme fails silently – Alice and Bob transacted, but the identity trail is genuine and court-recoverable.

2.4 Attack 4: Alice Repudiates a Transaction She Made

2.4.1 Scenario

Alice transacted legitimately with Bob (a law-abiding merchant). Later, Alice is investigated. She claims she never transacted with Bob, or that the identity Bob received was fabricated by Bob. Alice hopes that because the re-encrypted blob is opaque to everyone except Bob, a court cannot verify its authenticity without trusting Bob's word.

2.4.2 Why It Fails

The on-chain record contains everything needed to verify, without trusting Bob:

1. The `IdentityExchange(alice, bob, E_bob)` event is on-chain, emitted during Alice's `approve()` call. It was sent from Alice's address (verified by `msg.sender`).
2. The `approve()` transaction succeeded on-chain, which means the Chaum-Pedersen proof verified against Alice's registered `E_alice` from the registry. This is an immutable execution fact.
3. Therefore `E_bob` provably encrypts the same identity as `E_alice`. This follows from the soundness of the Chaum-Pedersen protocol – it is a mathematical fact, not testimony.
4. When the court compels Bob to decrypt `E_bob`, the resulting plaintext `I` is provably Alice's issuer-certified identity. The chain of trust:


```
Issuer PS-signed Alice's identity m          (verified at registration)
E_alice stored in IdentityRegistry           (on-chain, immutable)
Alice called approve() with E_bob + proof    (on-chain tx, msg.sender = alice)
Chaum-Pedersen proof verified on-chain        (execution succeeded)
==> E_bob encrypts Alice's real identity      (mathematical fact)
Bob decrypts E_bob --> I                      (compelled, deterministic)
I is Alice's issuer-certified identity        (proven)
```

2.4.3 What the Court Sees

An unbroken, cryptographically verified chain from issuer to decrypted identity. Alice's repudiation is contradicted by on-chain proof verification records that she cannot alter or dispute.

2.5 Attack 5: Criminal Bob Denies Receiving Alice's Identity

2.5.1 Scenario

Bob is a criminal merchant. Alice (law-abiding) transacted with Bob. Authorities investigate Bob. Bob claims he never received Alice's identity and cannot produce it – hoping to deny knowledge of his counterparties, or to shield Alice from being identified as his customer.

2.5.2 Why It Fails

The `IdentityExchange(alice, bob, E_bob)` event is on-chain and immutable. The court retrieves `E_bob` directly from the blockchain – Bob's cooperation is not needed to find the ciphertext.

Bob is compelled to decrypt:

```
M = C_b - sk_bob * R_b
```

If Bob claims he "lost" his identity private key (the alt_bn128 key, which is distinct from his Ethereum signing key):

- The court holds Bob in contempt, as it would for any refusal to produce subpoenaed records. "I lost the key" is no more credible for a cryptographic key than for a filing cabinet – particularly when the key was generated during a KYC ceremony and Bob accepted transactions requiring its use.
- Even without Bob's cooperation, the court can establish from the on-chain proof verification that `E_bob` *does* contain a genuine identity. Bob's refusal to decrypt is itself evidence of obstruction.

Bob cannot claim `E_bob` is meaningless or fabricated: the successful Chaum-Pedersen proof verification is an on-chain fact.

2.5.3 What the Court Sees

`E_bob` is on-chain. The proof that it contains Alice's real identity is on-chain. Bob decrypts or faces contempt. The cryptographic record is self-authenticating.

2.6 Attack 6: Eve Links Alice's Identities Across Counterparties

2.6.1 Scenario

Eve is a surveillance actor (competitor, stalker, rogue state). She observes that Alice's address called `approve` for multiple counterparties: Bob, Carol, and Dave. Eve collects the `IdentityExchange` events:

```
IdentityExchange(alice_addr, bob_addr,   E_bob)
IdentityExchange(alice_addr, carol_addr, E_carol)
IdentityExchange(alice_addr, dave_addr,  E_dave)
```

Eve wants to learn Alice's real identity, or to determine whether these transactions reveal a pattern (e.g., Alice's spending habits).

2.6.2 Why It Fails (Content Privacy Holds)

Each re-encryption uses independent randomness:

```
E_bob   = (r_1 * G,  r_1 * pk_bob   + I * G)
E_carol = (r_2 * G,  r_2 * pk_carol + I * G)
E_dave  = (r_3 * G,  r_3 * pk_dave  + I * G)
```

The ciphertexts are IND-CPA secure (semantic security of ElGamal): each is computationally indistinguishable from a random pair of curve points. Eve cannot decrypt any of them without the counterparty's private key. She cannot determine from the ciphertexts alone what identity they encode, nor can she compare them to determine if they encode the same identity.

What Eve does know: all three events originate from `alice_addr`. But this is exactly the information already visible in standard ERC-20 `Approval` events. The encrypted identity blobs add **zero additional linkability** beyond Ethereum's inherent pseudonymous address model.

What Eve does not know: Alice's name, jurisdiction, ID number, or any attribute of her identity. She sees an address; she does not see a person.

2.6.3 What Eve Sees

Pseudonymous Ethereum addresses and opaque ciphertexts that are computationally indistinguishable from random. The same view she has of any ERC-20 token, plus meaningless (to her) encrypted blobs.

2.6.4 Cross-Account Unlinkability

A stronger version of this attack: Eve suspects that addresses `addr_1`, `addr_2`, and `addr_3` all belong to Alice. She examines each address's registered credential in the IdentityRegistry.

Each credential was derived independently through the Identity Fountain: different rerandomized PS signatures, different identity key pairs, different ElGamal ciphertexts with independent randomness. PS rerandomization is *statistically* unlinkable – not merely computationally hard to correlate, but information-theoretically impossible. The credentials contain zero bits of mutual information.

Eve's only path to linking accounts is off-chain analysis: timing, amounts, behavioral patterns. The cryptographic identity layer gives her nothing.

2.7 Attack 7: Alice Uses a Revoked or Expired Credential

2.7.1 Scenario

Alice's identity credential was revoked by the issuer (e.g., her Alberta residency expired, fraud was detected in her original KYC documents, or she is subject to new sanctions). Alice still holds her old `E_alice` and `sigma` and attempts to continue transacting.

2.7.2 Why It Fails

The IdentityRegistry tracks credential validity. When the issuer revokes Alice's credential:

```
identityRegistry.revokeIdentity(alice_address)
```

Subsequent calls to `isVerified(alice)` return false. Alice's `approve` call – and any `transfer` or `transferFrom` – reverts at the identity check, before the Chaum-Pedersen proof is even evaluated.

Alice's old credential and proofs remain mathematically valid (the cryptography is eternal), but the contract refuses to accept them because her registration is no longer active. The revocation is a separate on-chain state, controlled by the trusted issuer.

2.7.3 What the Contract Sees

`isVerified(alice) == false`. Transaction reverts immediately. The still-valid-but-revoked credential is never consulted.

2.8 Attack 8: Unwitting Bob Receives a Transaction From Criminal Alice

2.8.1 Scenario

Bob is a law-abiding merchant running a BUCK-accepting storefront. Alice is a criminal who passed KYC (perhaps before she was sanctioned, or under a jurisdiction that has not yet flagged her). Alice sends BUCK to Bob through a normal transaction. Bob has no way to know Alice is a criminal at the time of the transaction.

Later, authorities investigate Alice. Bob is concerned: did he unknowingly participate in money laundering? Can Alice's identity in the receipt exonerate him?

2.8.2 Why the Scheme Protects Bob

Bob received Alice's re-encrypted identity `E_bob` during the `approve` handshake (or Bob pre-approved Alice via `approve(alice, 0, ...)` to receive from her). The Chaum-Pedersen proof verified on-chain, so `E_bob` provably contains Alice's real, issuer-certified identity.

When authorities investigate:

1. Bob decrypts `E_bob` and produces Alice's identity `I`.
2. The on-chain proof record establishes that `I` is genuine (not something Bob fabricated to frame Alice, and not something Alice fabricated to deceive Bob).
3. Bob can demonstrate he had no way to know Alice was a criminal: the identity he received was a valid, PS-verified credential at the time of the transaction. The revocation (if any) came later.

4. Bob's good faith is supported by the cryptographic record: he accepted a transaction from a verified participant and holds a court-presentable receipt.

The system protects honest participants by creating a tamper-proof record of who they transacted with. Bob is not liable for Alice's criminality; he is protected by the same cryptographic chain that convicts Alice.

2.8.3 What the Court Sees

Bob holds a provably genuine receipt. Alice's identity was valid at transaction time. Bob acted in good faith. The receipt is evidence of Bob's compliance, not his complicity.

2.9 Summary: Why the Scheme Defeats All Attacks

Every attack that involves substituting false identity data fails for the same fundamental reason: the Chaum-Pedersen proof is **sound**. A valid proof for a false statement does not exist.

```
The proof asserts:  E_bob encrypts the same M as E_alice
The contract reads: E_alice from the registry (PS-verified, immutable)
Therefore:          E_bob must encrypt Alice's real identity
Alice cannot:       supply a different E_alice (contract reads from storage)
Alice cannot:       produce a valid proof for different M (soundness)
Alice cannot:       forge the issuer's PS signature (q-SDH unforgeability)
Alice cannot:       link her accounts to each other (PS rerandomization)
Bob cannot:         produce a different plaintext (ElGamal decryption is deterministic)
Eve cannot:         decrypt E_bob (she lacks sk_bob)
Eve cannot:         link Alice's accounts (statistical unlinkability)
```


Three independent hardness assumptions protect the system:

1. **PS signature unforgeability** (q-SDH assumption): prevents forged credentials
2. **Discrete logarithm hardness** (alt_bn128): prevents false Chaum-Pedersen proofs and unauthorized ElGamal decryption
3. **ECDSA unforgeability** (secp256k1): authenticates Ethereum transactions

All three assumptions also underpin Ethereum itself. If any breaks, Ethereum's own security fails first – the Alberta Buck identity system does not introduce any novel cryptographic assumptions.

3 Identity Visibility Modes

Mode	What on-chain reveals	What off-chain reveals
Anonymous-valid	Address is verified (bit) isVerified(addr) == true	Encrypted receipt: identities of both parties, decryptable only by participants + optional regulatory escrow key
Public	Address is verified + public identity claim (plaintext)	Full name, jurisdiction, credentials, track record – world-readable
Contract	Contract address is verified; linked to deployer's identity	Deployer's identity is the contract's identity; receipts encrypted to deployer
Permissionless	No identity check	N/A

4 Operations Table

4.1 Core BUCK (ERC-20) Operations

Operation	Caller identity	Counterparty identity	Receipt	Notes
`mint`	Anonymous-valid KYC (topic 1)	N/A (self-mint)	Event log	Caller must hold BuckCredits; ide verified but not disclosed on-chain
`burn`	Anonymous-valid KYC (topic 1)	N/A (self-burn)	Event log	Reduces outstanding balance
`transfer`	Anonymous-valid KYC (topic 1)	Anonymous-valid KYC (topic 1)	Encrypted 3-party	Both parties verified; receipt encr to sender + receiver + escrow
`transferFrom`	Anonymous-valid KYC (topic 1)	Anonymous-valid KYC (topic 1)	Encrypted 3-party	Spender verified; owner and recipi identities in encrypted receipt
`approve`	Anonymous-valid KYC (topic 1)	Anonymous-valid KYC (topic 1)	Event log	Approving a spender for transferF

4.2 BuckCredit (ERC-721) Operations

Operation	Caller identity	Counterparty identity	Receipt	Notes
`createCredit`	Public KYC + insurer (42)	Anonymous-valid KYC (topic 1)	Encrypted to parties	Insurer MUST be publicly id (their reputation backs the c Client may be anonymous-va
`activate`	Anonymous-valid KYC (topic 1)	N/A (self)	Event log	Credit owner activates porti
`updateCredit`	Public KYC + insurer (42)	N/A	Event log	Insurer updates params; mu (reappraisal is a public comm
`transfer` (NFT)	Anonymous-valid KYC (topic 1)	Anonymous-valid KYC (topic 1)	Encrypted 3-party	Transferring insurance NFT inherits the credit

4.3 BuckKController (PID Stabilization)

Operation	Caller identity	Receipt	Notes
`compute`	Permissionless	Event log	Anyone can trigger PID update; minter pays gas. No identity needed.
`setGains`	Governance	Event log	Governance multisig or vote result. Governance members publicly identified.
`addBasketComponent`	Governance	Event log	Adding a new commodity to the basket.
`setDT`	Governance	Event log	Changing the PID update interval.

4.4 Oracle Operations

Operation	Caller identity	Receipt	Notes
Submit claim	Public KYC + reporter (43)	Public log	"I, Perry Kundert, claim Gold was $2905 at block 19400000." Public identity is the entire point: reputation accrues.
Commit value (L1)	Public KYC + reporter (43)	Event log	Authorized reporters sign the commitment. Signatures are publicly verifiable.
Challenge/dispute	Anonymous-valid KYC (topic 1)	Evidence on-chain	Challenger need not be public; the evidence speaks for itself.
Governance vote (oracle selection)	Anonymous-valid KYC (topic 1)	Vote record	Stake-weighted vote on oracle selection. Vote weight from Buck holdings is public; voter identity may be private.

4.5 DeFi Integration (Uniswap, Lending, etc.)

Operation	Caller identity	Contract identity	Receipt	Notes
Create pool (BUCK/USDT)	Public KYC (topic 1)	Contract registered to deployer's identity	Event log	Pool deployer is publicly Contract address linked t ElGamal credential in Ide
Swap BUCK -> USDT	Anonymous-valid KYC (topic 1)	Contract (deployer)	Encrypted 3-party	User sends BUCK to poo Receipt: user + deployer
Swap USDT -> BUCK	Anonymous-valid KYC (topic 1)	Contract (deployer)	Encrypted 3-party	Pool sends BUCK to user
Add liquidity	Anonymous-valid KYC (topic 1)	Contract (deployer)	Encrypted 3-party	User transfers BUCK to p
Remove liquidity	Anonymous-valid KYC (topic 1)	Contract (deployer)	Encrypted 3-party	Pool transfers BUCK to u

4.6 Identity Management

Operation	Caller identity	Issuer identity	Receipt	Notes
KYC ceremony	Self (no claims yet)	Trusted issuer (publicly identified)	Off-chain PS sig	Participant submits docs to issu issuer PS-signs identity scalar.
Register in registry	Self	N/A	Event log	Derives credential offline; registe with PS signature + NIZK proo
Add public identity claim	Self KYC (topic 1)	Trusted issuer (publicly identified)	On-chain claim	Opts in to public identification f oracle/governance participation.
Revoke claim	Issuer or self	N/A	Event log	Removes a claim (e.g., expired li
N/M recovery	N-of-M parties (all identified)	N/A	On-chain + off-chain	Transfers identity to new addres N of M designated parties agree

5 Contract Identity: How DeFi Pools Become BUCK Counterparties

A Uniswap pool contract is just an address. For BUCK transfers to/from that address to pass `isVerified`, the contract must have an identity in the IdentityRegistry. But a contract can't do KYC. The solution:

The pool deployer's identity becomes the contract's identity.

When someone creates a BUCK/USDT Uniswap pool:

1. Deployer (a verified BUCK participant) calls the Uniswap factory to create the pool. The factory deploys a new contract at a deterministic address.
2. Deployer derives a separate credential for the pool contract (from the same PS-signed identity, via the Identity Fountain) and registers it: `identityRegistry.registerCredential(poo pk_deployer, E_deployer, sigma', pi)`

   This links the pool contract's address to the deployer's credential. The pool inherits the deployer's verified identity.
3. Now `isVerified(poolAddress)` returns true. BUCK transfers to/from the pool pass the identity check.
4. When a user calls `approve` for the pool, the Chaum-Pedersen proof is verified against the pool's credential (which is the deployer's credential). The deployer's identity becomes the counterparty identity in all encrypted receipts.

This means: if you deploy a BUCK DeFi pool, you are the identified legal operator. Your identity is on the line for every transaction through that pool. If a court subpoenas you, you can be compelled to decrypt the ElGamal ciphertexts (because your identity private key was used for the pool's identity exchange handshakes).

6 On-Chain Integration: Receipts and Transfer Flow

The cryptography is sound. The question is where it meets the EVM – a machine that hides nothing.

6.1 The Receipt Constraint

Events are public. Every `emit` on Ethereum is readable by every node. A naive design that emits identity addresses or public keys in receipt events creates a permanent, public, linkable identity graph of every BUCK transfer – defeating the entire privacy model.

The constraints that drive the on-chain design:

1. **The EVM is transparent**: every contract execution, every storage slot, every emitted event is publicly reconstructible. The BUCK contract cannot hold decrypted identity data, even transiently.

2. **The contract must verify, not learn**: the BUCK contract enforces `isVerified` and verifies Chaum-Pedersen proofs. It confirms that a re-encrypted identity ciphertext is genuine. It never learns the identity itself.

3. **DeFi timing**: in a `transferFrom` call, the Uniswap router cannot pass receipt data. Identity exchange must happen earlier, at `approve` time.

The ElGamal Chaum-Pedersen approach resolves all three. The contract reads `E_alice` from the IdentityRegistry (not from calldata), verifies a compact Chaum-Pedersen proof (~29,000 gas), stores a hash of the re-encrypted ciphertext, and emits only opaque curve point coordinates. No identity content touches the EVM in any form.

6.2 The `approve` Function as Identity Handshake

The ERC-20 `approve` function is the natural point for identity exchange. Alice is directly calling the BUCK contract – she can provide the re-encrypted ElGamal ciphertext and the Chaum-Pedersen proof:

```
function approve(
    address spender,
    uint256 amount,
    uint256[4] calldata E_spender,       // ElGamal ciphertext (R.x, R.y, C.x, C.y)
    uint256[3] calldata chaumPedersen     // Fiat-Shamir proof (e, s1, s2)
) public returns (bool) {
    _approve(msg.sender, spender, amount);

    // Read the caller's registered credential from the registry
    // CRITICAL: read from storage, never from calldata
    uint256[4] memory E_caller = identityRegistry.getCredential(msg.sender);
    uint256[2] memory pk_caller = identityRegistry.getIdentityKey(msg.sender);
    uint256[2] memory pk_spender = identityRegistry.getIdentityKey(spender);

    // Verify Chaum-Pedersen proof: E_spender encrypts same M as E_caller
    require(
        _verifyChaumPedersen(
```

```
            E_caller, E_spender, pk_caller, pk_spender, chaumPedersen
        ),
        "Re-encryption proof invalid"
    );

    // Store hash of the verified re-encrypted ciphertext for this pair
    _receiptFragments[msg.sender][spender] = keccak256(
        abi.encodePacked(E_spender)
    );

    // Emit ONLY the opaque ElGamal ciphertext (4 uint256 curve coordinates)
    emit IdentityExchange(msg.sender, spender, E_spender);
    return true;
}
```

What the public sees in the `IdentityExchange` event:

- Alice's address approved a spender (same as standard ERC-20 Approval)
- Four `uint256` values encoding an ElGamal ciphertext – opaque curve point coordinates that only the spender can decrypt
- **No** identity address, **no** public key, **no** linkable identifier

What the contract verified (in ~29,000 gas):

- The ciphertext contains Alice's real identity (Chaum-Pedersen proof passed against the registered credential read from the registry)

What the spender (Bob / pool operator) can do:

- Decrypt `E_bob` using `sk_bob`: `M = C_b - sk_bob * R_b`
- Decode the identity from the message point `M = I * G`
- Store it for receipt construction on Holochain

6.2.1 Receiving Transfers: `approve(sender, 0)` Handshake

For an account that wants to **receive** from a specific counterparty:

```
// Alice wants to receive BUCK from Bob (Alice has private identity):
buck.approve(bob, 0, E_alice_for_bob, chaumPedersenProof)
```

Amount = 0 means no spending authorization. The side-effect: Alice's re-encrypted identity is now stored and verified for Bob. When Bob later calls `transfer(alice, amount)`, the BUCK contract knows Alice's identity fragment exists and is verified.

6.3 Receipt Construction: What the Contract Emits

The BUCK contract emits a **minimal** event that reveals only pseudonymous addresses (which the ERC-20 `Transfer` event already reveals) plus a reference to the pre-stored identity fragments:

```
event BuckTransferReceipt(
    address indexed from,
    address indexed to,
    uint256 amount,
    bytes32 fromIdentityHash,    // keccak256 of from's ElGamal ciphertext for to
    bytes32 toIdentityHash       // keccak256 of to's ElGamal ciphertext for from
);
```

The `fromIdentityHash` and `toIdentityHash` are keccak256 hashes of the opaque ElGamal ciphertexts stored during `approve`. They add **no linkability** beyond what the `Transfer` event already provides (the addresses are the same in both events).

The actual re-encrypted identity ciphertexts are retrievable:

- From the `IdentityExchange` events (emitted during `approve`)
- From Holochain (stored on the participants' source chains)

Only the counterparty (who has the identity private key to decrypt the ElGamal ciphertext) can recover the plaintext identity.

6.4 The Complete `transfer` / `transferFrom` Flow

```
function transfer(address to, uint256 amount) public override returns (bool) {
    require(identityRegistry.isVerified(msg.sender), "Sender not verified");
    require(identityRegistry.isVerified(to),         "Receiver not verified");
    require(compliance.canTransfer(msg.sender, to, amount), "Compliance");

    _transfer(msg.sender, to, amount);
    compliance.transferred(msg.sender, to, amount);

    // Emit minimal receipt -- no identity data, just hashes of
    // pre-stored re-encrypted ElGamal ciphertexts
    emit BuckTransferReceipt(
        msg.sender, to, amount,
        _receiptFragments[msg.sender][to],
        _receiptFragments[to][msg.sender]
    );

    return true;
}
```

The `transferFrom` path is identical: the receipt fragment hashes were stored during `approve`, so no additional identity data passes through the DeFi router. The Chaum-Pedersen proof was verified once (at `approve` time); subsequent transfers emit only the minimal receipt event (~2,000 gas).

6.5 Holochain: Off-Chain Computation and Storage

Alice's wallet (a Holochain hApp) performs all credential derivation and transaction preparation:

- **Credential derivation**: Alice's PS-signed identity `(m, sigma)` lives on her Holochain source chain. For each new account, the hApp rerandomizes the PS signature, generates a fresh key pair, creates an ElGamal encryption, and computes the NIZK proof. This runs entirely offline and can be batched – Alice can pre-generate credentials for future accounts.
- **Credential storage**: Each derived credential `(pk, E_alice, sigma', pi)` is stored on Alice's Holochain source chain. The full identity data (name, jurisdiction, ID number, KYC documents) is encoded in the identity scalar `m` and encrypted in the ElGamal ciphertext, not stored separately on Ethereum.
- **Re-encryption**: The hApp decrypts `E_alice` locally (Alice has `sk_alice`), extracts the message point `M`, and re-encrypts for Bob: `E_bob = (r' * G, r' * pk_bob + M)`. This is 2 scalar multiplications and 1 point addition – milliseconds on any device.
- **Proof generation**: The Chaum-Pedersen proof is a Schnorr-family sigma protocol. Generation requires 2 scalar multiplications and 1 hash – instantaneous on any modern device. Unlike Groth16 SNARK proofs (which require minutes of circuit evaluation and a trusted setup ceremony), Chaum-Pedersen proof generation is negligible.
- **Receipt storage**: Full encrypted receipts (with decrypted identity details, transaction metadata, purpose) are stored on both parties' Holochain source chains. The on-chain `BuckTransferReceipt` event provides only an anchor hash.

Holochain provides agent-centric, cryptographically auditable storage: the hApp DNA is identified by hash, so anyone can verify the exact code that runs on each participant's node. However, Holochain is **not** the trust anchor: Alice runs her own node and could run modified code. The on-chain verifications are the trust anchors – PS signature verification at registration and Chaum-Pedersen proof verification at transaction time are mathematically verifiable on-chain regardless of what software generated them. Holochain provides the execution environment; the proofs provide the guarantees.

6.6 Summary: What Each Layer Sees

Layer	What it sees
Ethereum (everyone)	`Transfer(from, to, amount)` – pseudonymous addresses
	`BuckTransferReceipt` – same addresses + opaque hashes
	`IdentityExchange` – address + opaque ElGamal ctxt
	Chaum-Pedersen proof verified (pass/fail) – no content
BUCK contract	`isVerified(from) == true`, `isVerified(to) == true`
	Chaum-Pedersen proof of re-encryption correctness
	keccak256 of re-encrypted ElGamal ciphertext (stored)
	Never sees plaintext identity
Counterparty (Bob)	Decrypts ElGamal ciphertext -> learns Alice's identity
	Constructs full receipt on Holochain
Sender (Alice)	Knows her own identity, knows Bob's (from his approve)
	Constructs full receipt on Holochain
Holochain hApp	Derives credentials (PS rerandomization + NIZK)
	Re-encrypts credential + generates proof locally
	Stores encrypted receipts on source chains
	Code identified by DNA hash (auditable)
Court (with subpoena)	Compels Bob to decrypt his ElGamal ciphertext
	On-chain proof record authenticates the decrypted result

7 Cryptographic Receipts: 3-Party ECDH

The BUCK contract emits minimal on-chain receipt events (previous section). The wallet layer (Holochain hApp) constructs full receipts with decrypted identity details, then encrypts them via 3-party ECDH so that only the participants can decrypt.

7.1 The Three Keys

For a transfer from Alice to a DeFi pool operated by Bob:

Key	Held by	Purpose
`k_s` (sender)	Alice	Sender's identity private key (alt_bn128)
`k_e` (ephemeral)	Deterministic	Derived from hash of on-chain receipt event; reproducible by anyone who knows the event
`K_r` (receiver)	Bob (deployer)	Receiver's identity public key (from registry)

The ephemeral key `k_e` is derived deterministically from the on-chain receipt event data (which is immutable and publicly verifiable):

```
// All fields are from the on-chain BuckTransferReceipt event:
receipt_anchor = abi.encode(
    from, to, amount,
    fromIdentityHash, toIdentityHash,
    block.number
```

```
)
k_e = keccak256(receipt_anchor)    // deterministic "private key"
```

Note: `receipt_anchor` contains only pseudonymous addresses and opaque hashes – no identity content. The identity public keys `K_s` and `K_r` are looked up from the IdentityRegistry (on-chain, public), not emitted in events.

7.2 Encryption Scheme

The wallet (Holochain hApp) computes the encryption key via ECDH:

```
// Alice's hApp computes (knows k_s, k_e; looks up K_r from registry):
K_r = identityRegistry.getIdentityKey(to)     // Bob's identity public key
shared_1 = ECDH(k_s, K_r)          // sender-receiver shared secret
shared_2 = ECDH(k_e, K_r)          // ephemeral-receiver shared secret
encryption_key = keccak256(shared_1 || shared_2)

// Construct the full receipt with decrypted identity data:
full_receipt = {
    from, to, amount, block.number, tx_hash,   // from on-chain event
    sender_identity:   decrypt(E_alice),       // from Alice's source chain
    receiver_identity: decrypt(E_bob_for_alice), // from Bob's approve
    purpose: "Uniswap V3 BUCK->USDT swap"
}

// Encrypt:
ciphertext = AES-GCM(encryption_key, encode(full_receipt))
```


The encrypted receipt is stored on Holochain (both parties' source chains). A commitment hash can optionally be emitted on-chain to anchor the receipt:

```
event ReceiptCommitment(
    bytes32 indexed transferHash,    // hash of the BuckTransferReceipt event
    bytes32 receiptHash              // hash of the encrypted full receipt
);
```

7.3 Who Can Decrypt

7.3.1 Sender (Alice)

Alice has `k_s` (her identity private key) and can recompute `k_e` from the on-chain receipt event:

```
k_e = keccak256(receipt_anchor)       // deterministic from on-chain data
K_r = identityRegistry.getIdentityKey(to)   // look up receiver's key
shared_1 = ECDH(k_s, K_r)            // she has k_s
shared_2 = ECDH(k_e, K_r)            // she has k_e
encryption_key = keccak256(shared_1 || shared_2)
```

7.3.2 Receiver / Pool Operator (Bob)

Bob has `k_r` (his identity private key) and looks up Alice's identity public key from the registry:

```
K_e = G * k_e                          // computable: k_e from receipt anchor
K_s = identityRegistry.getIdentityKey(from)  // Alice's identity public key
shared_1 = ECDH(k_r, K_s)             // he has k_r
shared_2 = ECDH(k_r, K_e)             // he has k_r
encryption_key = keccak256(shared_1 || shared_2)
```

7.3.3 Court-Compelled Decryption (Subpoena of Pool Operator)

A court subpoenas Bob to decrypt a specific transaction receipt:

```
// Court provides: the transaction hash
// Bob looks up: the BuckTransferReceipt event from that transaction
// Bob computes (using his identity private key k_r):
K_s = identityRegistry.getIdentityKey(from)
K_e = G * keccak256(receipt_anchor)
shared_1 = ECDH(k_r, K_s)
shared_2 = ECDH(k_r, K_e)
encryption_key = keccak256(shared_1 || shared_2)
receipt = AES-GCM-decrypt(encryption_key, ciphertext_from_holochain)
```


Bob reveals only the decrypted receipt. He never reveals `k_r`. He cannot decrypt receipts where he is not a party.

7.3.4 Regulatory Escrow (Optional)

A regulatory authority's public key `K_reg` adds a third shared secret:

```
shared_3 = ECDH(k_e, K_reg)
encryption_key = keccak256(shared_1 || shared_2 || shared_3)
```

The regulator can compute `shared_3` (they have `k_reg`, and `K_e` is deterministic from the on-chain event). But they still need `shared_1` or `shared_2` – requiring cooperation from a transaction participant. No unilateral surveillance.

8 Play-by-Play: BUCK/USDT Swap on Uniswap

Alice holds 1000 BUCK and wants to swap 500 BUCK for USDT. Bob has deployed a BUCK/USDT Uniswap V3 pool.

8.1 Prerequisites (One-Time Setup)

```
Alice:
  1. Completed KYC with trusted issuer (off-chain ceremony)
  2. Issuer PS-signs Alice's identity: sigma = PS.Sign(sk_issuer, m)
     where m = H(identity_data)
  3. Alice receives (m, sigma) and stores in Holochain wallet
  4. Alice's wallet derives a credential for this account (offline):
     - Rerandomizes PS signature: sigma' = (t * sigma_1, t * sigma_2)
     - Generates fresh identity key pair (sk_alice, pk_alice)
     - Encrypts: E_alice = (r * G,  m * G + r * pk_alice)
     - Computes NIZK proof pi linking sigma' to E_alice
  5. Registers on-chain (~200K gas):
     identityRegistry.registerCredential(
         alice_wallet, pk_alice, E_alice, sigma', pi)
  6. identityRegistry.isVerified(alice_wallet) == true

Bob (pool deployer):
  1. Completed KYC (same ceremony as Alice)
  2. Derived credential for his account (same offline process)
  3. Credential registered in IdentityRegistry (~200K gas)
  4. Created BUCK/USDT pool via Uniswap factory
     -> pool deployed at address 0xPOOL
  5. Derived and registered a separate credential for the pool
     (from same PS-signed identity, via the Identity Fountain):
     identityRegistry.registerCredential(
         0xPOOL, pk_bob_pool, E_bob_pool, sigma'_pool, pi_pool)
  6. identityRegistry.isVerified(0xPOOL) == true
```


8.2 The Swap Transaction

8.2.1 Step 1: Alice Approves the Router (Identity Handshake)

Alice's wallet (Holochain hApp) prepares the identity exchange locally, then Alice calls `approve` on the BUCK contract:

```
// 1. Off-chain (Holochain hApp on Alice's node -- milliseconds):
//    - Read E_alice from her source chain
//    - Decrypt: M = C_a - sk_alice * R_a  (extract message point)
//    - Look up the pool operator's identity public key pk_bob
//    - Re-encrypt: E_bob = (r' * G,  r' * pk_bob + M)
//    - Generate Chaum-Pedersen proof:
//        "E_bob encrypts the same M as E_alice"
//        (2 scalar muls + 1 hash -- instantaneous)

// 2. On-chain:
```

```
buck.approve(
    uniswapRouter,                      // spender
    500e18,                             // amount
    [R_b.x, R_b.y, C_b.x, C_b.y],    // E_bob (ElGamal ciphertext)
    [e, s1, s2]                         // Chaum-Pedersen proof
)
```

The BUCK contract:

```
1. Sets allowance[alice][uniswapRouter] = 500e18
2. Reads E_alice from IdentityRegistry (PS-verified, immutable)
3. Verifies Chaum-Pedersen proof: E_bob encrypts same M as E_alice
   (4 ecMul + 4 ecAdd + 1 keccak256 = ~29,000 gas)
4. Stores keccak256(E_bob) in _receiptFragments[alice][0xPOOL]
5. Emits IdentityExchange(alice, 0xPOOL, E_bob)
   -- the event contains ONLY the opaque ElGamal ciphertext
   -- no identity address, no public key, no linkable identifier
```

This is the critical moment. Alice is directly calling the BUCK contract, so she can provide the re-encrypted ciphertext and proof. By the time `transferFrom` is called later (by the router), the verified identity fragment is already stored.

8.2.2 Step 2: Alice Calls the Router

```
// Alice initiates the swap (standard Uniswap call -- no BUCK-specific data)
uniswapRouter.exactInputSingle(
    tokenIn:  BUCK,
    tokenOut: USDT,
    fee:      3000,        // 0.3% fee tier
    recipient: alice,
    amountIn: 500e18,
    amountOutMinimum: 490e6,  // slippage protection
    sqrtPriceLimitX96: 0
)
```

No identity data passes through this call. The router is a standard Uniswap contract that knows nothing about BUCK identity.

8.2.3 Step 3: Router Calls BUCK.transferFrom (Minimal Receipt Emission)

Inside the router, the swap triggers:

```
buck.transferFrom(alice, 0xPOOL, 500e18)
```

The BUCK contract executes (no identity data passes through the router):

```
function transferFrom(address from, address to, uint256 amount) ...
{
    _spendAllowance(from, msg.sender, amount);
    require(identityRegistry.isVerified(from), "Sender not verified");
    require(identityRegistry.isVerified(to),   "Receiver not verified");
    require(compliance.canTransfer(from, to, amount), "Compliance");

    _transfer(from, to, amount);
    compliance.transferred(from, to, amount);

    // Emit minimal receipt -- ONLY hashes of pre-stored ElGamal ciphertexts
    emit BuckTransferReceipt(
        from, to, amount,
        _receiptFragments[from][to],     // keccak256 of Alice's E_bob
        _receiptFragments[to][from]       // keccak256 of Bob's E_alice
    );

    return true;
}
```

What the public sees: the same from/to/amount as the standard ERC-20 Transfer event, plus two opaque hashes. No identity addresses. No public keys. No linkable identity information beyond the pseudonymous Ethereum addresses.

What the contract verified (during `approve`, earlier): the Chaum-Pedersen proof that Alice's re-encrypted ElGamal ciphertext encrypts the same identity as her registered credential.

8.2.4 Step 4: Wallet Constructs the Full Receipt (Off-Chain)

Alice's wallet (Holochain hApp) observes the `BuckTransferReceipt` event and the `IdentityExchange` events to construct the full receipt:

```
// 1. From on-chain events (trusted, immutable):
transfer_data = {from, to, amount, block.number, tx_hash}
receipt_hashes = {fromIdentityHash, toIdentityHash}

// 2. From IdentityExchange event (emitted during approve):
E_bob = Alice's re-encrypted identity for Bob (ElGamal ciphertext)

// 3. Alice decrypts her own credential locally:
M_alice = C_a - sk_alice * R_a     // extract message point from E_alice
alice_identity = decode(M_alice)    // recover identity data

// 4. From Bob's IdentityExchange (if Bob also did approve for Alice):
E_alice_from_bob = Bob's re-encrypted identity for Alice
```

```
M_bob = C_b' - sk_alice * R_b'    // Alice decrypts with her key
bob_identity = decode(M_bob)

// 5. Construct full receipt with decrypted identities
full_receipt = {
    transfer:      transfer_data,
    sender_name:   alice_identity.name,      // "Alice Johnson"
    receiver_name: bob_identity.name,         // "Bob Smith, DeFi Ops Inc."
    operation:     "Uniswap V3 swap BUCK -> USDT",
    pool:          "0xPOOL",
    timestamp:     block.timestamp
}

// 6. Encrypt the full receipt with 3-party ECDH
//    (see Cryptographic Receipts section)
receipt_anchor = abi.encode(from, to, amount, receipt_hashes, block.number)
k_e = keccak256(receipt_anchor)
K_bob = identityRegistry.getIdentityKey(to)  // look up from registry
shared_1 = ECDH(sk_alice, K_bob)
shared_2 = ECDH(k_e, K_bob)
encryption_key = keccak256(shared_1 || shared_2)
ciphertext = AES-GCM(encryption_key, encode(full_receipt))

// 7. Store on Holochain (both parties' source chains)
holochain.store(alice_chain, ciphertext)
holochain.store(bob_chain,   ciphertext)
```


8.2.5 Step 5: Pool Executes the Swap

The Uniswap pool sends USDT to Alice. Since USDT is not an Alberta Buck token, the USDT transfer has no BUCK identity requirements (it follows whatever rules USDT imposes, which is typically none).

8.2.6 Step 6: Completed State

```
On-chain (public):
  - ERC-20 Transfer event: alice_addr -> 0xPOOL, 500 BUCK
  - BuckTransferReceipt: same addresses + two opaque bytes32 hashes
  - IdentityExchange (from approve): alice_addr + opaque ElGamal ciphertext
  - Swap event: BUCK/USDT pool, 500 BUCK in, ~495 USDT out
  - Chaum-Pedersen proof verified (29K gas, no identity content leaked)

Off-chain (Holochain, encrypted):
  - Full encrypted receipt on Alice's and Bob's source chains
  - Alice's re-encrypted identity (ElGamal ciphertext, only Bob can decrypt)
```

```
  - Bob's re-encrypted identity (ElGamal ciphertext, only Alice can decrypt)
  - PS-signed identity on issuer's source chain (audit trail)

What anyone can see:
  - Two pseudonymous addresses transacted 500 BUCK
  - Both addresses pass isVerified (they are KYC'd participants)
  - Opaque ElGamal ciphertexts and hashes (indistinguishable from random)
  - A Chaum-Pedersen proof was verified (the ciphertexts are genuine)

What Alice can see:
  - Her own identity (trivially)
  - Bob's identity (decrypt his ElGamal ciphertext with her key)
  - Full receipt with names, amounts, purpose

What Bob can see:
  - His own identity (trivially)
  - Alice's identity (decrypt her ElGamal ciphertext with his key)
  - Full receipt with names, amounts, purpose

What a court can compel (via subpoena of Bob):
  - Bob decrypts Alice's ElGamal ciphertext for the specific transaction
  - On-chain Chaum-Pedersen proof record authenticates the result
  - Reveals Alice's identity for that one receipt
  - Bob never reveals his private key

What the public CANNOT see:
  - Alice's real name or identity details
  - Bob's real name (as pool operator)
  - Any linkable identity across transactions
  - The purpose or terms of the transaction
```


9 Edge Cases and Design Considerations

9.1 Flash Loans and Atomic Transactions

A flash loan borrows and repays in a single transaction. The borrower's address must be verified, and the lending contract must be registered. The receipt covers the full borrow-repay cycle as a single atomic event.

9.2 Multi-Hop Swaps

A swap routed through BUCK -> WETH -> USDT involves two BUCK transfers (BUCK -> pool1, pool1 -> pool2 if intermediate). Each BUCK transfer generates its own encrypted receipt. The router contract must also be registered if it holds BUCK transiently.

9.3 Contract-to-Contract Transfers

When one DeFi contract sends BUCK to another (e.g., a yield aggregator moving BUCK between pools), both contracts must be registered. The receipt identifies both contracts' deployers as the parties. This creates a clear chain of custody: every BUCK movement has an identified legal operator at each end.

9.4 Wrapping and Bridging

If BUCK is wrapped (WBUCK) for use on another chain, the wrapping contract must be registered. The wrap/unwrap generates receipts. On the destination chain, the bridged BUCK may operate under different identity rules (or no identity rules if the destination chain doesn't enforce ERC-3643). This is a known limitation – the Alberta Buck's identity guarantees only hold on chains where the IdentityRegistry is deployed.

9.5 Gas Costs

The design has three gas-consuming identity operations:

Operation	Cost (~gas)	When
`registerCredential` with PS verify (PS pairing: ~147K + NIZK: ~53K)	~200,000 gas	Once per account
`approve` with Chaum-Pedersen verify (proof verify: ~29K + std approve: ~46K)	~75,000 gas	Once per pair
`BuckTransferReceipt` event (2 indexed addresses + 3 uint256)	~2,000 gas	Per transfer

Credential registration uses the `ecPairing` precompile to verify the Pointcheval-Sanders signature and NIZK proof (`200,000 gas, once per account). The Chaum-Pedersen proof verification uses ~ecMul` / `ecAdd` (~29,000 gas per counterparty pair). Subsequent transfers emit only the minimal receipt event (~2,000 gas) with no additional proof work.

All encrypted identity data and full receipts are stored on Holochain, not on Ethereum. The only identity-related data emitted on-chain is the compact ElGamal ciphertext (4 curve coordinates = 128 bytes of calldata) during `approve`, and two `bytes32` hashes during each transfer.

9.6 Privacy of Pool Deployer Identity

The pool deployer's ElGamal credential is stored in the IdentityRegistry alongside the pool contract's address. The credential `E_deployer` is an opaque ciphertext – anyone can see it on-chain, but no one can decrypt it without the deployer's identity private key.

What is publicly visible: "0xPOOL is operated by a verified entity" (because `isVerified(0xPOOL) == true`). What is **not** visible: who that entity is. The deployer's real identity is revealed only in the ElGamal ciphertexts re-encrypted for specific counterparties during `approve` handshakes.

If a deployer wants public identification (e.g., "DeFi Ops Inc., Alberta"), they can register a separate plaintext identity claim alongside their encrypted credential. This is the expected case for commercial DeFi operators – public identity builds trust and attracts liquidity.

10 Alternative Approaches Considered

The ElGamal Chaum-Pedersen scheme was chosen after evaluating several cryptographic alternatives. Each has merits for other use cases, but none matches the combination of low gas cost, zero setup, instant proof generation, and strong security guarantees needed for an ERC-20 identity layer.

10.1 Pedersen Commitments + Groth16 ZK Proofs

In this approach, the issuer stores a Pedersen commitment `C = v*G + r*H` on-chain. Alice proves in zero knowledge (via a Groth16 SNARK) that her re-encrypted ciphertext `E_bob` encrypts the same value `v` committed in `C`.

Advantages:

- Extremely flexible: the ZK circuit can prove arbitrary predicates alongside re-encryption correctness (age > 18, jurisdiction membership, credit score in range, etc.)
- Information-theoretically hiding (Pedersen commitments hide `v` perfectly, vs. ElGamal's computational hiding)
- Well-supported toolchain (circom, snarkjs, Hardhat plugins)

Why Chaum-Pedersen is superior for Alberta Buck:

- **Gas cost**: Groth16 verification requires the `ecPairing` precompile (~113,000 gas base + per-pair costs), totaling ~200,000-300,000 gas. Chaum-Pedersen costs ~29,000 gas – **7-10x cheaper**.
- **Trusted setup**: Groth16 requires a multi-party computation ceremony to generate proving/verification keys. A compromised ceremony allows forged proofs – the one thing the system cannot tolerate. Chaum-Pedersen has no setup of any kind.
- **Proof generation time**: Groth16 proof generation takes seconds to minutes (circuit evaluation, witness computation, FFTs over large fields). Chaum-Pedersen proof generation is 2 scalar multiplications and 1 hash – milliseconds on a phone.
- **Complexity**: Groth16 requires circuit definition (R1CS/circom), a prover library, and careful constraint engineering. A bug in the circuit silently compromises soundness. Chaum-Pedersen is ~50 lines of elliptic curve arithmetic with a well-understood security proof.

- **Unnecessary generality**: Alberta Buck needs exactly one thing: proof that two ElGamal ciphertexts encrypt the same message point. Groth16 can prove anything in NP; that generality is wasted here and only adds attack surface.

10.2 Proxy Re-Encryption (NuCypher Umbral)[5]

In PRE, Alice generates a re-encryption key `rk_{A->B}` that allows an untrusted proxy to transform `E_alice` into `E_bob` without decrypting. Umbral includes NIZK proofs of correct re-encryption.

Advantages:

- Alice does not need to be online at re-encryption time (the proxy acts on her behalf)
- Threshold PRE distributes trust across multiple proxies (no single point of compromise)
- Algebraic NIZK proofs are efficient (comparable gas cost to Chaum-Pedersen)

Why Chaum-Pedersen is superior for Alberta Buck:

- **Alice is always online**: in the Alberta Buck model, Alice calls `approve` directly – she is necessarily online. The proxy delegation feature of PRE is unnecessary complexity.
- **No proxy metadata leakage**: even with threshold PRE, the proxies learn metadata (who is re-encrypting for whom, how often). Direct re-encryption eliminates this metadata channel entirely.
- **Simpler key management**: PRE requires generating and distributing re-encryption keys per counterparty pair. Direct ElGamal re-encryption uses only the existing identity key pairs already registered on-chain.
- **Protocol complexity**: Umbral manages capsules, key fragments, and proxy coordination. Direct re-encryption with Chaum-Pedersen is two scalar multiplications and a sigma protocol – no capsules, no fragments, no proxies.
- PRE remains valuable in other contexts (e.g., encrypted email where the sender may be offline), but Alberta Buck's on-chain interaction model makes it unnecessary.

10.3 BBS+ Signatures with Selective Disclosure

BBS+ (pairing-based multi-message signatures) allows an issuer to sign a vector of attributes. Alice can selectively disclose specific attributes while hiding others, with ZK proofs of signature validity. The W3C Verifiable Credentials standard is converging on BBS+ for exactly this use case.

Advantages:

[5] Umbral: A Threshold Proxy Re-Encryption Scheme. NuCypher. Reference implementation: `https://github.com/nucypher/pyUmbral` Whitepaper: `https://raw.githubusercontent.com/nucypher/umbral-doc/master/umbral-doc.pdf`

- Attribute-level granularity: prove "jurisdiction = Alberta" without revealing name or ID number
- Unlinkable presentations (different proofs from the same credential cannot be correlated)
- Ecosystem alignment with W3C VC / DID standards

Why Chaum-Pedersen is superior for Alberta Buck:

- **Gas cost**: BBS+ verification requires pairing operations (~113,000+ gas). Chaum-Pedersen uses only scalar multiplications (~29,000 gas).
- **Selective disclosure is the wrong primitive**: Alberta Buck transfers require the *full* identity to be re-encrypted for the counterparty (for court-recoverable receipts). Selective attribute hiding would defeat the purpose: the counterparty needs the whole identity, and a court needs the whole identity.
- **Different trust model**: BBS+ proves "I hold a valid credential with these properties." Alberta Buck needs "this ciphertext contains my specific issuer-certified identity." The latter is a re-encryption correctness proof, not a selective disclosure proof.
- **Cross-account unlinkability is already solved**: Pointcheval-Sanders rerandomizable signatures provide unlinkable credential derivation for different accounts. Within a single account, re-encryptions share a common Ethereum address regardless of the credential scheme, so BBS+ presentation unlinkability adds nothing.
- BBS+ could complement Chaum-Pedersen in the future – e.g., for compliance checks that verify jurisdiction eligibility without full identity disclosure. But it does not replace the core re-encryption mechanism.

10.4 Camenisch-Shoup Verifiable Encryption

A purpose-built primitive: encrypt a value and simultaneously prove that the plaintext satisfies a public relation, without revealing it.

Advantages:

- Theoretically clean: directly solves "encrypt for Bob, prove to everyone it's the right value"
- Can prove arbitrary relations on the encrypted plaintext

Why Chaum-Pedersen is superior for Alberta Buck:

- Camenisch-Shoup is the general framework; Chaum-Pedersen is its optimal specialization for the "same plaintext" relation. When the relation is "same plaintext as another ElGamal ciphertext in the same group," the Chaum-Pedersen protocol is the canonical, minimal-cost instantiation.
- General Camenisch-Shoup constructions are more complex and less efficient.

10.5 Pedersen Equality Proofs (Homomorphic Comparison)[6]

Pedersen commitments support homomorphic equality checking: given `C_1 = v*G + r_1*H` and `C_2 = v*G + r_2*H`, revealing `delta_r = r_1 - r_2` lets the verifier check `C_1 - C_2 == delta_r * H` without learning `v`.

Advantages:

- Extremely cheap (1 point addition, 1 scalar multiplication)
- Information-theoretically hiding

Why this is insufficient for Alberta Buck:

- Pedersen commitments are *commitments*, not *encryptions*. They hide the value but do not allow a specific party to decrypt. Alberta Buck needs Bob to recover Alice's identity, not just to verify that two opaque commitments match. ElGamal provides both hiding and targeted decryption.
- Revealing `delta_r` to prove equality leaks the randomness difference, which is acceptable for on-chain verification but means the commitment scheme cannot support multiple independent equality proofs without additional blinding (each proof leaks one linear relation on the randomness).

10.6 Comparison Summary

Approach	On-chain gas	Proof gen	Setup	Targeted decryption	Sufficient alone?
ElGamal Chaum-Pedersen	**~29K**	**ms**	**none**	**yes**	**yes**
Groth16 SNARK	~200-300K	sec-min	trusted	yes	yes
Umbral PRE	~30-50K	ms	none	yes	yes
BBS+	~150-200K	ms	pairing params	no (proves properties)	no (complement)
Camenisch-Shoup	~50-100K	ms	none	yes	yes
Pedersen equality	~7K	ms	none	no	no

ElGamal Chaum-Pedersen wins on per-transaction gas cost, has zero setup requirements, generates proofs instantly, supports targeted decryption, and requires minimal implementation. Combined with Pointcheval-Sanders rerandomizable signatures for credential derivation (one-time ~200K gas at registration), the system achieves both cross-account unlinkability and per-transaction re-encryption correctness – without the gas burden or trusted setup of SNARK-based alternatives.

[6]Pedersen Commitments and Bulletproofs. Bulletproofs: Short Proofs for Confidential Transactions and More. Bunz, Bootle, Boneh, Poelstra, Wuille, Maxwell. Overview: `https://tlu.tarilabs.com/cryptography/the-bulletproof-protocols` Pedersen commitments in confidential transactions: `https://www.nccgroup.com/research/on-the-use-of-pedersen-commitments-for-confidential-payments/`

11 Summary: Identity Requirement by Risk Level

Risk level	Operations	Identity mode	Receipt
Highest	BuckCredit creation/update, oracle claims, governance	Public	On-chain event + public record
High	Pool deployment, compliance changes	Public	On-chain event
Medium	BUCK transfer, DeFi swap, add/remove liquidity	Anonymous-valid	Encrypted 3-party ECDH receipt
Low	BUCK burn, credit activation	Anonymous-valid	On-chain event
None	BuckK compute, read oracle, view balances	Permissionless	On-chain event