

The Alberta Buck - Identity and Receipts (v1.0)

Perry Kundert

2026-04-08



Alberta Buck transactions carry provably authentic identity – without exposing it. Every participant holds an ElGamal ciphertext¹ of their identity, backed by a Pointcheval-Sanders rerandomizable signature². A citizen with multiple accounts derives an independent, unlinkable credential for each – rerandomizing the signature offline – yet every credential is verifiable as issued by the same authority. At transaction time, the participant re-encrypts under the counterparty’s key and produces a Chaum-Pedersen proof³ that the re-encrypted ciphertext contains the same identity as the registered credential. No one sees the plaintext.

Per-transaction verification costs ~29,000 gas (alt_bn128 scalar multiplications, no pairings). One-time credential registration costs ~235,000 gas (pairing-based PS signature check). No trusted setup.

Identity data resides in three layers: Ethereum contract storage (proof chain), the local Holochain wallet (plaintext identity, secret keys), and the Holochain DHT (encrypted identity exchange, Warrants for malfeasance detection). All proofs are permanently on-chain, so a public AMM pool operator can satisfy a regulatory subpoena months later using only their secret key and an archival node – without any counterparty’s cooperation. (PDF, Text, Cryptography Example Code)

¹ElGamal 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.

²Pointcheval-Sanders Signatures. Pointcheval, D. and Sanders, O. "Short Randomizable Signatures." CT-RSA 2016, LNCS 9610, pp. 111-126. Rerandomizable signatures: given a valid signature σ 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.

³Chaum-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	4
1.1	Revealing Identity while Preserving Privacy	4
1.2	Accounts Belong to Someone	7
1.3	Counterparties Exchange Identities	8
1.4	ElGamal and Pointcheval-Sanders Over a Mandated Curve Group	9
1.5	Chaum-Pedersen Proof of Re-Encryption Correctness	10
2	Attacks and Defenses	13
2.1	Attack 1: Alice Substitutes a False Identity for Bob	13
2.2	Attack 2: Alice Forges an Issuer Credential	14
2.3	Attack 3: Alice and Complicit Bob Launder With False Identities	15
2.4	Attack 4: Alice Repudiates a Transaction She Made	16
2.5	Attack 5: Criminal Bob Denies Receiving Alice’s Identity	16
2.6	Attack 6: Eve Links Alice’s Identities Across Counterparties	17
2.7	Attack 7: Alice Uses a Revoked or Expired Credential	19
2.8	Attack 8: Unwitting Bob Receives a Transaction From Criminal Alice	19
2.9	Attack 9: Systematic Forgery – Alice Attacks the Trust Chain	20
2.10	Summary: Why the Scheme Defeats All Attacks	22
3	Identity Visibility Modes	23
4	Operations Table	23
4.1	Core BUCK (ERC-20) Operations	24
4.2	BuckCredit (ERC-721) Operations	24
4.3	BuckKController (PID Stabilization)	24
4.4	Oracle Operations	24
4.5	DeFi Integration (Uniswap, Lending, etc.)	25
4.6	Identity Management	25
5	Contract Identity: How DeFi Pools Become BUCK Counterparties	25
5.1	What’s actually wired up	26
5.2	The interaction surface	26
5.3	Step-by-step: bringing up a BUCK/USDC AMM	27
5.4	What this means for accountability	27
5.5	Encrypted-Identity contracts	28
6	Why approve Is Always CP-Bound	28
6.1	Approve documents consent; transfer documents execution	28
6.2	Engagement with the BN254 identity key, not just the ETH key	29
6.3	Per-pair freshness vs. constant identity hash	29
6.4	Uniform receipt semantics; no fast-path / slow-path bifurcation	29
6.5	Defense against silent rebinding / identity rotation	29
6.6	Cost	30

7	On-Chain Integration: Receipts and Transfer Flow	30
7.1	The Receipt Constraint	30
7.2	The <code>approve</code> Function as Identity Handshake	30
7.3	Receipt Construction: What the Contract Emits	32
7.4	The Complete <code>transfer</code> / <code>transferFrom</code> Flow	33
7.5	Holochain: Off-Chain Computation and Storage	34
7.6	Summary: What Each Layer Sees	36
8	Cryptographic Receipts: 3-Party ECDH	36
8.1	The Three Keys	37
8.2	Encryption Scheme	37
8.3	Who Can Decrypt	38
9	Play-by-Play: BUCK/USDT Swap on Uniswap	39
9.1	Prerequisites (One-Time Setup)	39
9.2	The Swap Transaction	40
10	Edge Cases and Design Considerations	44
10.1	Flash Loans and Atomic Transactions	44
10.2	Multi-Hop Swaps	44
10.3	Contract-to-Contract Transfers	45
10.4	Wrapping and Bridging	45
10.5	Gas Costs	45
10.6	Pool Operator Identity Binding	45
10.7	<code>alt_bn128</code> Security Horizon	46
11	Alternative Approaches Considered	46
11.1	Pedersen Commitments + Groth16 ZK Proofs	47
11.2	Proxy Re-Encryption (NuCypher Umbral)	47
11.3	BBS+ Signatures with Selective Disclosure	48
11.4	Camenisch-Shoup Verifiable Encryption	49
11.5	Pedersen Equality Proofs (Homomorphic Comparison)	49
11.6	Comparison Summary	50
12	Summary: Identity Requirement by Risk Level	50

1 Identity with Anonymity

Every Alberta Buck account holds a cryptographically certified identity that is invisible to observers but provably genuine to authorized counterparties. The core design rests on three pillars: the Identity Fountain (offline credential derivation with statistical unlinkability), the two-channel architecture (Ethereum proof chain plus Holochain data channel), and the agent-per-account isolation model. Together with the bilateral **approve** / **transfer** flow, these give citizens free transactions with verified counterparties, let regulators recover identity under subpoena from on-chain proof chains, and ensure that no party – including the government – can surveil transactions at $O(1)$ cost.

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 government-approved hostile groups to ravage citizens 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 and use them to identify yourself as a participant in every transaction, how free are you?

The Alberta Buck enforces *these* principles:

- The freedom to *transact in privacy and peace* by keeping your identity private between only you and your counterparties (no third party is able to correlate activity between any of your accounts), and
- The confidence that someone *really is who they say they are* by proving their identity has not been forged (you have the ability to vet every counterparty's identity before transacting with them).

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 signatures² 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 $m = H(\text{identity_data})$.
- (c) Picks a random point h in G_1 and computes the PS signature:
 $\text{sigma} = (h, (x + m * y) * h)$
where (x, y) is the issuer's secret key.
- (d) Gives Alice the identity scalar m and signature $(\text{sigma}_1, \text{sigma}_2)$.
- (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. What `identity_data` Contains

`identity_data` is a canonical (deterministically serialized) structure containing Alice's verified identity claims:

```
identity_data = canonical_encode({
  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
})
```

The identity scalar is $m = H(\text{identity_data})$ where H is keccak256 over the canonical encoding. The canonical encoding is deterministic: field order, encoding format, and whitespace

are fixed by the hApp DNA, so any party who holds the same `identity_data` computes the same `m`.

The issuer verifies these claims during the KYC ceremony and computes `m`. Alice receives both `m` (the scalar) and `identity_data` (the plaintext). The plaintext never touches Ethereum – it is the *content* that the on-chain proof chain guarantees. Alice stores `identity_data` as a Private entry on her Holochain source chain, encrypted with her agent key and never published to the DHT.

Different issuers may define different schemas (a province vs. a bank), but all schemas produce a single scalar `m` via the same hash function. The `epoch` field binds the credential to a renewal period (see Epoch-Based Credential Renewal below).

3. 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:

$$\text{sigma}' = (t * \text{sigma}_1, t * \text{sigma}_2)$$

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:

$$E_{\text{new}} = (r * G, m * G + r * \text{pk}_{\text{new}})$$

- (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) $e(\text{sigma}'_1, X + m * Y) = e(\text{sigma}'_2, g_2)$ -- valid PS signature
- b) $C = m * G + r * \text{pk}_{\text{new}}$ -- 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.

Each credential derivation corresponds to a **separate Holochain agent**. Alice’s wallet spawns a new Holochain agent – a fresh Ed25519 key pair and an independent source chain – for each Ethereum account. From the perspective of both Ethereum and Holochain, these agents are entirely distinct identities: separate addresses, separate agent IDs, separate source chains. The one artifact shared across agents is Alice’s `identity_data` (and therefore `m = H(identity_data)`), which is copied into each new agent’s Private entry store. Since `m` is committed only inside ElGamal ciphertexts (IND-CPA secure) and PS signatures (statistically rerandomized), no observer on either network can detect that two agents share the same `m`.

4. 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: $\sim 235,000$ 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 ($\sim 29,000$ gas).

5. 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.

6. 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 $m = H(\text{identity_data})$. 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 $(pk, E_alice, \text{sigma}', pi)$ in the on-chain IdentityRegistry. The contract verifies the PS signature and NIZK proof via the `ecPairing` precompile (235,000 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 proof³ `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 ($\sim 29,000$ 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.

This is a **two-channel design**. The on-chain ElGamal ciphertexts and Chaum-Pedersen proofs form the *proof chain*: they bind Alice’s registered identity to the re-encrypted ciphertext Bob receives, without revealing content. The actual `identity_data` (name, jurisdiction, ID number) travels via the *data channel*: Alice’s Holochain hApp sends it to Bob’s hApp, encrypted end-to-end. Bob verifies the binding:

```
M = C_b - sk_bob * R_b          // decrypt ElGamal ciphertext
assert H(identity_data) * G == M // bind off-chain data to proof chain
```

EC-ElGamal decryption yields a curve point `M`, not the scalar `m`. Recovering `m` from `M = m * G` is the discrete logarithm problem. The proof chain never carries plaintext – it proves *which* identity was sent. The data channel delivers *what* that identity contains. The hash commitment `H(identity_data) * G == M` binds them cryptographically: Bob can verify the `identity_data` he received via Holochain is exactly the identity the issuer certified and Alice registered on-chain.

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 ($\sim 235,000$ 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 protocol³ 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):

$$E_{\text{alice}} = (R_a, C_a) = (r_a * G, r_a * pk_{\text{alice}} + M)$$

where $M = m * G$ (identity encoded as a curve point, $m = H(\text{identity_data})$)

Rerandomized PS signature sigma’ verified at registration (not consulted again)

Alice’s re-encryption for Bob:

$$E_{\text{bob}} = (R_b, C_b) = (r_b * G, r_b * pk_{\text{bob}} + M)$$

using the SAME message point M

1.5.2 What Alice Proves (Non-Interactively)

"I know (sk_{alice}, r_b) such that:

1. $pk_{\text{alice}} = sk_{\text{alice}} * G$ -- I own this public key
2. $R_b = r_b * G$ -- I know E_{bob} ’s randomness
3. $C_a - sk_{\text{alice}} * R_a = C_b - r_b * pk_{\text{bob}}$ -- 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 $I * G \neq I_{\text{fake}} * G$ 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:

$E_{\text{alice}} = (R_a, C_a)$ -- 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 ciphertxts
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.

The Fiat-Shamir challenge must bind all public inputs and context to prevent replay and malleability:

```

e = H(G, pk_alice, pk_bob, R_a, C_a, R_b, C_b, T1, T2,
      msg.sender, spender, block.chainid)

```

Including `msg.sender` and `spender` (Ethereum addresses) binds the proof to the specific transaction context. Including `block.chainid` prevents cross-chain replay. On-chain, `msg.sender` is authenticated by ECDSA and `spender` is a calldata argument to `approve()`, but including them in the hash explicitly prevents a proof generated for one (`sender`, `spender`) pair from being replayed in another context. This is standard Fiat-Shamir domain separation, not an additional cryptographic assumption.

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 (<code>E_alice</code>)	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
Calldata + SSTORE overhead			~10,000
Total			~235,000 gas

The NIZK proof linking the PS signature to the ElGamal ciphertext is a standard Schnorr-family sigma protocol – *not* a Groth-Sahai proof. The PS verification relation $e(\text{sigma}'_1, X + m * Y) = e(\text{sigma}'_2, g_2)$ is *linear* in m : the public inputs sigma'_1 , sigma'_2 , X , Y , g_2 are fixed curve points, and the secret m enters linearly via $m * Y$. A single challenge-response pair binds m to both the pairing equation and the ElGamal ciphertext. The pairing is computed by the *verifier* on public inputs; the prover demonstrates knowledge of m via scalar responses. No bilinear-map proof system is needed.

The concrete verification protocol:

Public inputs:

```

sigma' = (sigma'_1, sigma'_2) -- rerandomized PS signature (G_1)
E_new  = (R, C)              -- ElGamal ciphertext (G_1)
pk_new -- identity public key (G_1)
(X, Y) -- issuer public key (G_2)
g_2    -- generator (G_2)

```

Proof: $\text{pi} = (e, s_m, s_r)$

Verifier computes:

```

// (a) PS signature check via ecPairing precompile:
//   Reconstruct the commitment  $A = s_m * \text{sigma}'_1 - e * \text{sigma}'_2$ 
//   in  $G_1$ , then verify the pairing product equation:
A_ps = ecMul(sigma'_1, s_m) + ecMul(sigma'_2, -e) // 2 ecMul + 1 ecAdd
ecPairing([A_ps, Y], [ecMul(sigma'_1, e), X],
          [ecMul(sigma'_2, -e), g_2]) == 1 // 3 pairings

// (b) ElGamal consistency check (pure G_1 arithmetic):
//    $s_m * G + s_r * \text{pk\_new} == e * C + T_b$ 
//    $s_r * G == e * R + T_r$ 
//   where  $T_b, T_r$  are the prover's commitments (included in proof)

// (c) Fiat-Shamir challenge reconstruction:
e' = H(sigma', E_new, pk_new, A_ps, T_b, T_r, registrant_address)
require(e' == e)

```

Part (a) uses the ecPairing precompile (3 pairs: 147K gas). Parts (b) and (c) use ~ecMul / ecAdd (~30K gas). Total: ~235K gas. This is a standard Schnorr-family sigma protocol applied to a linear pairing relation – the same class of proof used in BLS signature verification and PS credential presentation protocols.

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

Nine adversarial scenarios follow – from a sanctioned individual substituting a false identity, through issuer credential forgery, to systematic attacks on the trust chain – each defeated by a specific mathematical checkpoint. The Chaum-Pedersen proof catches identity substitution; the PS pairing equation catches credential forgery; the Fiat-Shamir binding prevents replay; and the on-chain proof chain provides non-repudiation. A summary table maps each attack to its failure point and the hardness assumption that protects it.

The trust chain in every scenario is:

Issuer PS-signs identity m	--> identity is genuine
Alice derives $(E_{\text{alice}}, \sigma', \pi)$	--> 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} :

$$E_{\text{bob_fake}} = (r' * G, \quad r' * \text{pk}_{\text{bob}} + I_{\text{fake}} * G)$$

She submits $E_{\text{bob_fake}}$ 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:

$$C_a - \text{sk}_{\text{alice}} * R_a == C_b - r' * \text{pk}_{\text{bob}}$$

Expanding the left side (from the registered E_{alice} in the registry):

$$\begin{aligned}
& (r_a * pk_{alice} + I * G) - sk_{alice} * (r_a * G) \\
&= r_a * sk_{alice} * G + I * G - sk_{alice} * r_a * G \\
&= I * G
\end{aligned}$$

Expanding the right side (from Alice's fake E_{bob}):

$$\begin{aligned}
& (r' * pk_{bob} + I_{fake} * G) - r' * pk_{bob} \\
&= I_{fake} * G
\end{aligned}$$

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 framework⁴). Registration checks the pairing equation:

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

⁴ERC-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>

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 σ' 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 π 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.

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 <code>m</code>	(verified at registration)
<code>E_alice</code> stored in <code>IdentityRegistry</code>	(on-chain, immutable)
Alice called <code>approve()</code> with <code>E_bob</code> + proof	(on-chain tx, <code>msg.sender = alice</code>)
Chaum-Pedersen proof verified on-chain	(execution succeeded)
<code>==></code> <code>E_bob</code> encrypts Alice's real identity	(mathematical fact)
Bob decrypts <code>E_bob</code> <code>--></code> <code>I</code>	(compelled, deterministic)
<code>I</code> 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:

$$\begin{aligned} E_{bob} &= (r_1 * G, r_1 * pk_{bob} + I * G) \\ E_{carol} &= (r_2 * G, r_2 * pk_{carol} + I * G) \\ E_{dave} &= (r_3 * G, r_3 * pk_{dave} + I * G) \end{aligned}$$

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.

2.6.5 Scope of Unlinkability

Alberta Buck provides *identity privacy* (no observer can determine who owns an address from the credential) and *credential unlinkability* (no observer can cryptographically link two credentials to the same person). It does *not* provide *transaction graph privacy*: Ethereum balances and transfer events are public, and standard chain-analysis techniques – timing correlation, amount matching, funding-source tracing – apply to BUCK as they do to any ERC-20 token.

Transaction graph privacy is a distinct problem solved by different primitives (confidential transactions, mixing protocols, or fully shielded transfers as in Zcash). These are orthogonal to the identity layer and could complement it in the future. The identity system’s guarantee is narrower but sufficient: even if Eve links Alice’s addresses through behavioral analysis, she cannot learn Alice’s *name* or *identity data* from any on-chain artifact. The cryptographic identity layer ensures that linking addresses reveals spending patterns, not people.

2.6.6 Why M Is the Same Across All Accounts

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 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* (σ', pk, E), not the identity scalar itself.

The privacy boundary is:

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

Before `approve`, Alice’s accounts are unlinkable to everyone. After `approve`, only that specific counterparty learns Alice’s identity for that account. Counterparties who already know Alice’s name gain nothing new from comparing M values.

See the worked example for the full argument and the AMM pool subpoena scenario.

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 Attack 9: Systematic Forgery – Alice Attacks the Trust Chain

The previous attacks target individual operations. This attack assumes Alice is willing to lie about *every* artifact she produces – hashes, proofs, signatures, ciphertexts – and asks whether she can deceive anyone about the validity or content of her identity.

The trust chain has three links. We attack each one.

2.9.1 Link 1: PS Signature (Issuer Authority)

Alice has $(m, \text{sigma}_1, \text{sigma}_2)$ from the issuer. She produces $\text{sigma}' = (t * \text{sigma}_1, t * \text{sigma}_2)$.

Can she produce a sigma' that signs a different m_{fake} ?

PS verification checks:

$$e(\text{sigma}'_1, X + m * Y) = e(\text{sigma}'_2, g_2)$$

Rearranging:

$$e(\text{sigma}'_1, Y)^m = e(\text{sigma}'_2, g_2) * e(\text{sigma}'_1, X)^{-1}$$

The right side is fixed by σ' (which Alice chose). The left side is determined by m . For σ' derived from the issuer's signature on m , this equation holds only for the real m . Alice cannot produce a valid PS signature on any m_{fake} without the issuer's secret key (x, y) . **Blocked by q-SDH unforgeability.**

2.9.2 Link 2: NIZK Proof (Binding PS Signature to ElGamal Ciphertext)

Alice provides $(\sigma', E_{\text{new}}, pk_{\text{new}}, \pi)$. The NIZK proof proves knowledge of (m, r) satisfying:

- a) $e(\sigma'_1, X + m * Y) = e(\sigma'_2, g_2)$ -- σ' signs m
- b) $C = m * G + r * pk_{\text{new}}$ -- E_{new} encrypts m
- c) $R = r * G$ -- consistent randomness

Can she use a valid σ' (signing m) but create E_{new} encrypting m_{fake} ?

The sigma protocol uses a single response scalar s_m that must simultaneously satisfy conditions (a) and (b):

- For condition (a): s_m encodes the real m (the value σ' signs)
- For condition (b): s_m encodes whatever is in E_{new}

If E_{new} encrypts $m_{\text{fake}} \neq m$, no single s_m satisfies both checks. **Blocked by sigma protocol soundness.**

Can she forge the Fiat-Shamir challenge? The challenge $e = H(\sigma', E_{\text{new}}, pk_{\text{new}}, \text{commitments})$ depends on all public inputs. Finding inputs that yield a favorable challenge requires a keccak256 preimage attack. **Blocked by random oracle assumption.**

2.9.3 Link 3: Chaum-Pedersen Proof (Binding Re-encryption to Registered Credential)

At `approve()` time, Alice provides E_{bob} and a Chaum-Pedersen proof. The contract reads E_{alice} from storage.

Can she re-encrypt a different identity? The proof requires:

$$C_a - sk_{\text{alice}} * R_a = C_b - r_b * pk_{\text{bob}}$$

Both sides must equal the same message point $M = m * G$. If Alice encrypts $M_{\text{fake}} \neq M$, the equation is unsatisfiable. **Blocked by DLog hardness.** (Same mathematics as Attack 1.)

Can she supply a different E_{alice} ? No – read from storage, not from calldata.

2.9.4 Critical Edge Case: The Trivial Signature

If Alice sets $\sigma' = (0, 0)$ where 0 is the point at infinity:

$$\begin{aligned} e(0, X + m * Y) &= 1 && \text{(pairing with identity is trivial)} \\ e(0, g_2) &= 1 && \text{(same)} \end{aligned}$$

The pairing equation $1 = 1$ holds for any m . Alice could register a credential encrypting any m_{fake} she wants, produce a valid-looking NIZK proof (because condition (a) is trivially satisfied), and the verification would pass.

She achieves this by picking $t = 0$ in the rerandomization: $\text{sigma}' = (0 * \text{sigma}_1, 0 * \text{sigma}_2) = (0, 0)$.

The fix is standard: the PS signature specification requires $\text{sigma}'_1 \neq 0$. The registration contract must check this before evaluating the pairing:

```
require(sigma_prime[0] != 0 || sigma_prime[1] != 0, "Degenerate signature");
```

On BN254, G_1 has prime order (cofactor 1), so there are no small-order points. The identity check $\text{sigma}'_1 \neq 0$ is the *only* degenerate case.

Without this check, the entire fountain is broken. With it, the system is sound.

2.9.5 Additional Attack Surfaces

Off-curve points: post-Istanbul, the `ecPairing` precompile validates that inputs lie on `alt_bn128` and reverts otherwise. Alice cannot use off-curve points to confuse the pairing computation.

Subgroup attacks on G_2 : the `ecPairing` precompile validates G_2 subgroup membership (EIP-197). Alice cannot exploit small subgroups in the twist curve.

Eve copies Alice's registered credential: Eve can read $(pk, E, \text{sigma}', pi)$ from the chain and re-register it for her own address. But Eve cannot *use* it – the Chaum-Pedersen proof requires knowledge of sk_{alice} (condition 1: $pk_{\text{alice}} = sk_{\text{alice}} * G$). A stolen credential is useless without the corresponding secret key.

Alice shares raw credential material: Alice can give (m, sigma) to an accomplice, who can then derive valid credentials containing Alice's identity. But the decrypted identity is always Alice's – the consequences fall on her. This is inherent in any bearer credential system; it is no different from handing someone your driver's license.

2.9.6 Verdict

The system is sound against an untrustworthy Alice who is willing to lie about every artifact she produces. She cannot forge any artifact that would cause a verifier to accept a false identity. The three layers – PS unforgeability, NIZK soundness, Chaum-Pedersen soundness – chain together so that breaking any link requires breaking a standard hardness assumption.

One critical implementation requirement: $\text{sigma}'_1 \neq 0$ must be checked at credential registration. This is a standard requirement of the Pointcheval-Sanders scheme and costs negligible gas.

2.10 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

BUCK accounts operate in one of three identity modes, which determine the data flow at **approve** and **transfer** time. Private accounts require bilateral **approve** calls (each party re-encrypts their identity for the counterparty); public accounts (AMM pools, exchanges) are exempt from **approve** because their `identity_data` is already world-readable; contract accounts inherit their deployer’s identity. The four transfer combinations (Private-Private, Private-Public, Public-Private, Public-Public) range from two **approve** calls with full Holochain exchange down to a standard ERC-20 transfer with no identity work at all.

Mode	What on-chain reveals	What off-chain reveals
Anonymous-valid (private)	Address is verified (bit) <code>isVerified(addr) == true</code>	Encrypted receipt: identities of both parties, decryptable only by participants + optional regulatory escrow key. Requires bilateral approve for transfer.
Public-Identity (contract)	Contract is verified + bound with public identity claim <code>isPublicIdentity(c) == true</code>	Full name, jurisdiction, credentials, track record – world-readable. Counterparty side falls back to bound <code>_identityHash</code> when no per-pair receipt.
Contract	Contract address is verified <code>(bindContract)</code>	Deployer publishes (pk, E); receipts use the bound <code>_identityHash</code> on the contract’s side when no per-pair fragment exists.
Permissionless	No identity check	N/A

4 Operations Table

Every Alberta Buck operation has a defined identity requirement and receipt model. High-risk operations (BuckCredit creation, oracle claims, governance) require public identity; medium-risk operations (transfers, DeFi swaps) require anonymous-valid identity with encrypted 3-party ECDH receipts; low-risk operations (burns, credit activation) require only self-verification; and permissionless operations (price reads, balance queries) need no identity at all. The tables below specify the exact identity mode, counterparty requirements, and receipt format for each operation across the BUCK ERC-20, BuckCredit ERC-721, and governance contracts.

4.1 Core BUCK (ERC-20) Operations

Operation	Caller identity	Counterparty identity	Receipt	Notes
<code>mint</code>	Anonymous-valid KYC (topic 1)	N/A (self-mint)	Event log	Caller must hold BuckCredits; identity verified but not disclosed on-chain
<code>burn</code>	Anonymous-valid KYC (topic 1)	N/A (self-burn)	Event log	Reduces outstanding balance
<code>transfer</code>	Anonymous-valid KYC (topic 1)	Anonymous-valid KYC (topic 1)	Encrypted 3-party	Both parties verified; receipt encrypted 3-party ECDH (no unilateral decryption)
<code>transferFrom</code>	Anonymous-valid KYC (topic 1)	Anonymous-valid KYC (topic 1)	Encrypted 3-party	Spender verified; owner and recipient identities in encrypted receipt
<code>approve</code>	Anonymous-valid KYC (topic 1)	Anonymous-valid KYC (topic 1)	Event log	Approving a spender for transferFrom

4.2 BuckCredit (ERC-721) Operations

Operation	Caller identity	Counterparty identity	Receipt	Notes
<code>createCredit</code>	Public KYC + insurer (42)	Anonymous-valid KYC (topic 1)	Encrypted to parties	Insurer MUST be publicly identified (their reputation backs the credit). Client may be anonymous-valid.
<code>activate</code>	Anonymous-valid KYC (topic 1)	N/A (self)	Event log	Credit owner activates portion of
<code>updateCredit</code>	Public KYC + insurer (42)	N/A	Event log	Insurer updates params; must be (reappraisal is a public commitment)
<code>transfer (NFT)</code>	Anonymous-valid KYC (topic 1)	Anonymous-valid KYC (topic 1)	Encrypted 3-party	Transferring insurance NFT; new inherits the credit

4.3 BuckKController (PID Stabilization)

Operation	Caller identity	Receipt	Notes
<code>compute</code>	Permissionless	Event log	Anyone can trigger PID update; minter pays gas. No identity needed.
<code>setGains</code>	Governance	Event log	Governance multisig or vote result. Governance members publicly identified.
<code>addBasketComponent</code>	Governance	Event log	Adding a new commodity to the basket.
<code>setDT</code>	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 identified. Contract address linked to deployer's ElGamal credential in IdentityRegistry.
Swap BUCK -> USDT	Anonymous-valid KYC (topic 1)	Contract (deployer)	Encrypted 3-party	User sends BUCK to pool via contract. Receipt: user + deployer + escrow.
Swap USDT -> BUCK	Anonymous-valid KYC (topic 1)	Contract (deployer)	Encrypted 3-party	Pool sends BUCK to user. Receipt: pool + user + escrow.
Add liquidity	Anonymous-valid KYC (topic 1)	Contract (deployer)	Encrypted 3-party	User transfers BUCK to pool via contract.
Remove liquidity	Anonymous-valid KYC (topic 1)	Contract (deployer)	Encrypted 3-party	Pool transfers BUCK to user via contract.

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 issuer; issuer PS-signs identity scalar.
Register in registry	Self	N/A	Event log	Derives credential offline; registers with PS signature + NIZK proof.
Add public identity claim	Self KYC (topic 1)	Trusted issuer (publicly identified)	On-chain claim	Opts in to public identification. Exempts counterparties from approval.
Revoke claim	Issuer or self	N/A	Event log	Removes a claim (e.g., expired license).
N/M recovery	N-of-M parties (all identified)	N/A	On-chain + off-chain	Transfers identity to new address after N of M designated parties agree.

5 Contract Identity: How DeFi Pools Become BUCK Counterparties

Smart contracts can't do KYC, yet every BUCK transfer requires both parties to be *identifiable*. The solution is **bindContract**: at deployment time the operator (or an atomic `BuckAwareDeployer.deployAndBind` helper) calls `IdentityRegistry.bindContract(target, pk, E, isPublicIdentity_)`, publish-

ing the operator’s BN254 public key `pk` and ElGamal ciphertext `E` against the contract’s address. `bindContract` is **first-binder-wins** (anyone may bind any address whose code is deployed; the deployer protects against front-running by binding atomically with the deploy), the bound contract is marked `isVerified[target] = true`, and the boolean `isPublicIdentity[target]` is set per the binding. When the spender or a transfer endpoint is bound under a Public Identity, the BUCK contract may fall back to the bound `_identityHash(target)` in receipts when no per-pair receipt fragment exists – but Chaum-Pedersen on the sender’s side is **always** required at **approve** time, even against a Public- Identity spender, so a subpoena of the sender always recovers a full encrypted-to-counterparty receipt.

The Uniswap V2 integration (`test/UniswapV2Integration.t.sol`) exercises this end-to-end: the BUCK/USDC pair is created by the Uniswap factory, then bound under a Public Identity with the operator’s (`pk`, `E`). Verified Alice adds liquidity; verified Bob swaps in either direction; unverified Carol is rejected.

5.1 What’s actually wired up

The relevant code in `src/IdentityRegistry.sol`:

```

// Bind a deployed contract address to an operator-controlled identity.
// First-binder-wins; deployers protect against front-running by binding
// atomically alongside the deploy (BuckAwareDeployer.deployAndBind).
// 'isPublicIdentity_' selects the receipt-fragment fallback policy: if
// true, transfers may use the bound _identityHash on this side when no
// per-pair CP fragment exists; if false (Encrypted-Identity contract)
// the contract participates exactly like an EOA (no fallback).
function bindContract(
    address target,
    Point    calldata pk,
    Cipher   calldata E,
    bool          isPublicIdentity_
) external {
    require(target.code.length > 0, "not a contract");
    require(!isVerified[target],   "already bound");
    _pks[target]                    = pk;
    _ciphers[target]                = E;
    isVerified[target]              = true;
    isPublicIdentity[target]        = isPublicIdentity_;
    emit ContractBound(target, msg.sender, isPublicIdentity_);
}

```

No `register` for the contract address (no PS signature is needed because the contract has no underlying KYC credential of its own); the operator’s already-registered (`pk`, `E`) is republished against the contract’s address. Whoever holds the operator’s secret key `sk` can decrypt every receipt that uses the bound ciphertext.

5.2 The interaction surface

Public-Identity contracts compose with BUCK in two places. Both relax the per-pair receipt-fragment requirement, but neither bypasses the Chaum-Pedersen handshake on the sender’s side.

1. **Approve against any spender, including Public-Identity** (`Buck.approve`). Approve always requires CP – there is no `isPublicIdentity` shortcut. The sender’s wallet constructs `E_spender` by re-encrypting against the spender’s published `pk` (read from `IdentityRegistry`), produces the CP proof, and the contract verifies it. This guarantees that a subpoena of the sender’s records always recovers a full receipt for the counterparty’s identity. See **Why approve Is Always CP-Bound** for the full rationale (consent vs. execution, BN254-key engagement, per-pair freshness, uniform audit semantics, and defence against future identity rotation).
2. **Transfer with a Public-Identity counterparty** (`Buck._identityCheckedTransfer`). Both parties must be `isVerified`. If a per-pair receipt fragment exists for a side, it is used as that side’s identity material in the receipt. If a side has no fragment and `isPublicIdentity[side] == true`, the bound `_identityHash(side)` is used as that side’s fragment. EOA \leftrightarrow EOA Encrypted-Identity transfers MUST have a prior CP receipt – there is no fallback for EOAs.

5.3 Step-by-step: bringing up a BUCK/USDC AMM

Concrete deployment sequence as exercised by the integration test:

1. Operator calls `UniswapV2Factory.createPair(BUCK, USDC)`. The factory deploys a new `UniswapV2Pair` at a CREATE2-deterministic address.
2. Operator calls `reg.bindContract(pair, pk_op, E_op, true)` and `reg.bindContract(router, pk_op, E_op, true)` (or atomically via `BuckAwareDeployer.deployAndBind`, which avoids any first-binder-wins front-running between deploy and bind). `pk_op` and `E_op` are the operator’s already-registered identity material.
That’s the entire administrative step. The pair and the router are now valid Public-Identity BUCK counterparties; their `_identityHash` is the operator’s bound hash.
3. A verified user (Alice) approves the router for BUCK using the identity-bound `approve` overload. Alice’s wallet reads `pk_router` from `IdentityRegistry` (which is `pk_op`), decrypts her own stored ciphertext to recover `M`, re-encrypts under `pk_router` to produce `E_router`, generates the CP proof, and submits. The contract verifies CP and stores `_receiptFragments[alice][router] = H(E_router)`. She approves USDC using plain ERC-20 `approve`. Then `router.addLiquidity(BUCK, USDC, ...)`. The router’s internal `safeTransferFrom` calls `land` Alice’s BUCK and USDC into the pair. The pair issues LP tokens to Alice.
4. Verified swappers (Bob, ...) call `router.swapExactTokensForTokens`. BUCK \rightarrow USDC: pair receives Bob’s BUCK (verified from, Public-Identity to – allowed); pair sends USDC out (no BUCK identity check on the USDC side). USDC \rightarrow BUCK: pair sends BUCK to Bob (Public-Identity from, verified to – the pair side falls back to `_identityHash(pair)` because no per-pair fragment exists from the pair to Bob).
5. An unverified user (Carol) trying to receive BUCK from a swap is rejected: the pair’s `_safeTransfer` on BUCK reverts with "BUCK: recipient not verified", which the pair surfaces as "UniswapV2: TRANSFER_FAILED".

5.4 What this means for accountability

The pool’s identity is the operator’s identity. The bound (`pk`, `E`) is the operator’s already-registered material; whoever holds the operator’s `sk` can decrypt every receipt that uses the pool’s

`_identityHash`. Every BUCK transfer touching the pool emits a `BuckTransferReceipt` whose pool-side fragment is either a per-pair CP-bound ciphertext (if one exists) or the bound `_identityHash(pool)` (if none does). Auditors and subpoena responders can:

- **On the sender's side:** produce the CP-bound ciphertext from the sender's own records (always present – approve always requires CP).
- **On the pool's side:** ask the operator (whose key `sk` binds to `pk`) for the corresponding identity, just as for an EOA counterparty.

The operator is the legal counterparty. Setting up a BUCK/X AMM is permissionless at the factory level (anyone can call `createPair`). What gates the pool's usability for BUCK is `bindContract` – and **whoever calls `bindContract` first wins**. That is operationally significant: the operator must bind atomically with the deploy (via `BuckAwareDeployer.deployAndBind`) to prevent an attacker from front-running and binding the pool to *their* key. Once bound, the operator's identity is the contract's identity for audit purposes.

`bindContract` does not require governance. Permissioning is instead a property of *how the contract is deployed*: a contract meant to be operator-controlled deploys via a deployer/binder helper that performs both steps atomically. Governance plays no role.

5.5 Encrypted-Identity contracts

Setting `isPublicIdentity_=false` in `bindContract` produces an **Encrypted-Identity contract**: the contract is bound and verified exactly like a Public-Identity contract, but it participates in transfers identically to an EOA – no per-side receipt-fragment fallback to `_identityHash`. Use this for treasury or escrow contracts whose interactions deserve full per-pair CP receipts on both sides without exposing a permanent `_identityHash` in receipts that lack a fragment.

6 Why approve Is Always CP-Bound

The receipt-fragment fallback to `_identityHash` described above applies only to the *transfer* side of the protocol. At `approve` time the rule is simpler and stricter: **every** call to `Buck.approve` requires a Chaum-Pedersen proof, regardless of whether the spender is a private EOA, a Public-Identity contract, or an Encrypted-Identity contract. The parameterless ERC-20 `approve(address, uint256)` reverts with `BUCK: use identity-bound approve`.

This is a deliberate departure from a tempting optimization. When the spender is a Public-Identity contract, both parties' identities are already on-chain (the sender via `register`, the spender via `bindContract`), so a *transfer* receipt could in principle be reconstructed at execution time using `_identityHash` on both sides without any CP work. The transfer event would document *who paid whom* correctly.

What that optimization would lose is the on-chain record of *who consented to whom*. Five concrete things break if `approve` becomes CP-optional:

6.1 Approve documents consent; transfer documents execution

In every `transferFrom` flow – DeFi routers, swap aggregators, multi-hop trades, allowance-based subscriptions – the spender is not the ultimate recipient and the `approve` precedes the transfer by an unbounded amount of time. `Approve` is the moment Alice says "I am authorising this address to draw against my balance"; the `transferFrom` is when the spender does the drawing,

often weeks later. With CP-bound approve, that consent leaves a permanent per-pair commitment (`_receiptFragments[alice][spender] = keccak256(E_spender)`) plus an `IdentityExchange` event carrying the ciphertext. With plain ERC-20 approve, the only artifact is a standard `Approval(alice, spender, amount)` log – indistinguishable from a unit-test approval, a phishing approval, or any other context-free allowance write. The transfer event happens after the consent moment has been forgotten.

6.2 Engagement with the BN254 identity key, not just the ETH key

CP requires the sender's wallet to decrypt its own `E_alice` with `sk_alice{BN254}`, re-encrypt under `pk_spender`, and produce a Schnorr-style transcript. That proves the holder of the *identity* key personally signed off on this counterparty – not merely the holder of the Ethereum tx-signing key. The two keys live in the same wallet today, but a custodian, a hardware wallet without BN254 support, a session key, or a compromised tx-signer can authorise plain ERC-20 approves without ever touching the identity layer. CP forecloses that gap. An EOA whose Ethereum key has leaked to an attacker can still be drained on-chain, but the leaked approves will not produce CP fragments and will be visibly non-conforming to a downstream audit.

6.3 Per-pair freshness vs. constant identity hash

`_identityHash(spender)` is the same value forever – it appears identically in every receipt that spender ever participates in. The CP fragment is per (`sender, spender`) pair and is committed at the moment of consent. Without it, a regulator looking at receipts for "Bob's pool" sees `_identityHash(pool)` against every counterparty and cannot, from chain data alone, separate "Alice deliberately authorised this pool" from "Alice was somehow drawn against by this pool." The existence of `_receiptFragments[alice][pool] != 0` is itself the cryptographic proof of consent for that specific pair.

6.4 Uniform receipt semantics; no fast-path / slow-path bifurcation

If approve were CP-optional for Public-Identity counterparties, every audit tool, every operator, and every regulator would have to handle two receipt flavours: "fragment present, sender-signed off-on" and "fragment absent, fall back to identity hash." Operators would route flow through the bifurcation to obscure consent trails (e.g., always approve the public router, never the EOA). Mandatory CP collapses both cases into one: the fragment is *always* present after a successful approve, and `_identityHash` fallback exists only on the *transfer* side, only between parties who never approved each other, and only for the side that is `isPublicIdentity = true`.

6.5 Defense against silent rebinding / identity rotation

Today `bindContract` is one-shot, so a contract's identity is fixed at first bind. But any future revocation, rotation, or compliance-driven rebind would mean `_identityHash(spender)` changes. The CP fragment captures *the spender identity that the sender actually authorised at this block height*. Plain ERC-20 approve would capture only the address – useless if the operator changes hands later, since the address can outlive the binding.

6.6 Cost

The premium for the always-CP rule is 29K gas per `approve` ($4 \text{ ecMul} + 4 \text{ ecAdd} + 1 \text{ keccak256}$) plus the storage write of one 32-byte `_receiptFragments` slot. `Approve` is amortised across many `transferFrom` calls – the per-transfer cost remains $\sim 2\text{K}$ gas (one event + two SLOADs). In exchange, every BUCK authorisation leaves a sender-key-bound, per-pair, on-chain consent record that is openable on subpoena without the counterparty’s cooperation and without trust in any operator.

The shortest version: the receipt event is reconstructable from registry data, but the consent event is not. CP at `approve` is what makes the BUCK identity layer’s *consent* visible on-chain, not just its *execution*.

7 On-Chain Integration: Receipts and Transfer Flow

The BUCK ERC-20 contract stores and emits the bilateral `approve / transfer` protocol, the `BuckTransferReceipt` event format, and the `_receiptFragments` storage layout. The central tension is that the EVM hides nothing – every storage slot and event is publicly reconstructible – so the contract carries only opaque ciphertexts and verification flags, never plaintext identity. All proof verification happens during `approve` ($\sim 29\text{K}$ gas); the `transfer` itself performs only boolean checks on pre-computed flags, yielding a standard ERC-20 transfer flow with identity guarantees where no on-chain artifact reveals who is transacting with whom.

7.1 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 ($\sim 29,000$ 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.

7.2 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: $M = C_b - sk_bob * R_b$
- Receive identity_data from Alice via Holochain (encrypted for Bob)

- Verify authenticity: $H(\text{identity_data}) * G == M$ (binds off-chain data to the on-chain proof chain)
- Store verified identity for receipt construction on Holochain

7.2.1 Bilateral Identity Exchange

The `transfer` and `transferFrom` functions require that *both* parties' identities are available to each other. For private accounts, this means both must call `approve`:

```
// Alice (private) wants to transact with Bob (private):
// Step 1: Alice approves Bob -- re-encrypts her identity for Bob
buck.approve(bob, amount, E_alice_for_bob, chaumPedersenProof)

// Step 2: Bob approves Alice -- re-encrypts his identity for Alice
buck.approve(alice, 0, E_bob_for_alice, chaumPedersenProof)
// Amount = 0: no spending authorization, just identity exchange
```

Both identity fragments must exist before BUCK can move. If Bob's counterparty is a **Public-Identity contract** (e.g., a Uniswap pool bound via `bindContract(pool, pk_op, E_op, true)`), step 2 is not required from the pool side: `isPublicIdentity(pool) == true` lets the bilateral check fall back to the bound `_identityHash(pool)` on the pool side. Alice still must complete step 1 (her own CP- bound `approve`) – `approve` always requires CP, even against Public-Identity spenders, so a subpoena of Alice always recovers her per-pair ciphertext for the pool.

This means a Public-Identity pool receives BUCK from any verified private account that has done a single `approve` – no per-counterparty setup on the pool's side. EOA<->EOA private transfers always require both `approve` steps; there is no fallback for EOAs.

7.3 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.

7.3.1 Canonical Receipts and Public Accounts

Both parties in a transfer hold each other's plaintext `identity_data` (either from `approve` + Holochain, or from the public IdentityRegistry). Both hold the transfer data (from the on-chain event). If the receipt format is canonical (deterministic field order, encoding), either party can independently construct the identical receipt and compute $H(\text{receipt})$. The receipts match without coordination.

The receipt model varies by visibility mode:

Sender	Receiver	Receipt construction
Public	Public	Fully on-chain. Reconstructable by anyone from Transfer event + public IdentityRegistry entries. No off-chain component.
Public	Private	Private identity delivered via Holochain. Both parties construct canonical receipt off-chain.
Private	Private	Both identities delivered via Holochain. Both parties construct canonical receipt off-chain.

For transfers involving private accounts, the contract cannot compute $H(\text{receipt})$ because it never sees plaintext identities. The on-chain `BuckTransferReceipt` event provides anchoring data (the ElGamal ciphertext hashes from `approve`) that binds the off-chain receipt to the on-chain transfer.

For two public accounts, no receipt hash is needed: every input to the receipt is already public on-chain. The transfer is a standard ERC-20 `transfer` with identity verification – no Holochain wallet, no off-chain coordination required.

7.4 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");

    // Bilateral identity check. The primary identity material is the
    // per-pair receipt fragment stored at approve() time. If a side
    // has no fragment AND that side is bound under a Public Identity,
    // fall back to the bound _identityHash for that side. EOA<->EOA
    // Encrypted transfers MUST have prior CP receipts (no fallback).
    bytes32 fromHash = _receiptFragments[msg.sender][to];
    bytes32 toHash = _receiptFragments[to][msg.sender];

    if (toHash == 0) {
        require(
            identityRegistry.isPublicIdentity(msg.sender)
            || identityRegistry.isPublicIdentity(to),
            "no receipt fragment and no public side"
        );
    }
}
```

```

    );
    toHash = identityRegistry.identityHash(to);
}
// fromHash always falls back best-effort to identityHash(from).
if (fromHash == 0) {
    fromHash = identityRegistry.identityHash(msg.sender);
}

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

emit BuckTransferReceipt(
    msg.sender, to, amount,
    fromHash, toHash
);

return true;
}

```

The bilateral identity check enforces that both parties are `isVerified` and that the receipt event carries identity material for both sides. A **Public-Identity contract** (pool, router, treasury bound via `bindContract(target, pk, E, true)`) lets the pool side fall back to the bound `_identityHash(pool)` when no per-pair receipt fragment exists. Approvals are still always CP-bound; there is no `isPublicIdentity` shortcut at approve time.

Sender	Receiver	Approvals needed
EOA	EOA	Both must call CP-bound <code>approve</code> before move
EOA	Pub-Id	Only sender approves; pool side falls back
Pub-Id	EOA	Sender approves; receiver side falls back
Pub-Id	Pub-Id	One side may approve; both can fall back

The `transferFrom` path applies the same bilateral check. Receipt fragments were stored during `approve`, so no additional identity data passes through a DeFi router. Chaum-Pedersen verification ran once at `approve` time; subsequent transfers emit only the minimal receipt event (~2,000 gas plus two SLOADs for the fragments).

7.5 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.
- **Identity data storage:** Alice’s plaintext `identity_data` is stored as a **Private entry** on her Holochain source chain, encrypted with her agent key and never published to the DHT. Only Alice’s own agent can read it. This is the master copy from which `m = H(identity_data)` can always be recomputed.

- **Credential storage:** Each derived credential (pk , E_{alice} , σ' , pi , sk) is also stored as a Private entry on the agent’s source chain. The identity private key sk (an alt_bn128 scalar) is stored alongside the credential. The ElGamal ciphertext binds m to the credential; the plaintext $identity_data$ that gives m meaning lives only in the Private entry, never on Ethereum.
- **Identity data delivery** (the data channel): When Alice re-encrypts her identity for Bob, her hApp also sends $identity_data$ to Bob’s hApp via Holochain, encrypted under Bob’s Holochain agent public key. Bob’s hApp decrypts it, computes $H(identity_data) * G$, and checks equality with M recovered from the on-chain ElGamal ciphertext. This verification binds the off-chain plaintext to the on-chain proof chain. The Holochain data channel carries the content; the Ethereum proof chain guarantees its authenticity.
- **Non-publication detection:** Bob’s wallet watches for `IdentityExchange` events on Ethereum and checks for the corresponding DHT entry within a grace period. If Alice completes the on-chain `approve` but never publishes her $identity_data$ to the DHT, Bob’s wallet can flag the account as non-compliant, issue a Holochain **Warrant** citing the on-chain tx hash, and optionally revoke the approval. The warrant is independently verifiable: any agent can confirm the `IdentityExchange` event exists on Ethereum but the corresponding DHT entry does not. Alice’s non-publication does not protect her regardless – Bob can still decrypt M from the on-chain ciphertext, and the regulator can obtain $identity_data$ from the issuer.
- **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 meta-data, purpose) are stored on both parties’ Holochain source chains. The on-chain `BuckTransferReceipt` event provides only an anchor hash.

7.5.1 Agent Isolation Model

Alice does not reuse a single Holochain agent across multiple BUCK accounts. For each Ethereum address she creates, her wallet application spawns a new Holochain agent:

- A fresh Holochain key pair (Ed25519) is generated.
- A new source chain is initialized.
- The hApp DNA is the same (same code, same hash), but the agent identity is distinct.
- Alice copies her $identity_data$ into the new agent’s Private store and derives a credential from it (rerandomizing her PS signature, generating a fresh alt_bn128 key pair, encrypting via ElGamal, computing the NIZK proof).

Why agent isolation matters: Holochain’s DHT uses agent public keys as identifiers. If Alice used a single Holochain agent for all her BUCK accounts, a DHT observer could correlate them: "agent X stored credentials for addresses addr_1, addr_2, addr_3." With separate agents, each agent’s DHT presence is independent. No Holochain observer can link Alice_1 to Alice_2, just as no Ethereum observer can link addr_1 to addr_2.

Practical operation: Alice’s wallet application (a Holochain Launcher or custom conductor) manages multiple agent instances internally. From Alice’s perspective, she sees a unified wallet with multiple accounts. Under the hood, each account is a separate Holochain agent with its own source chain. The wallet stores a master seed (protected by Alice’s passphrase or biometrics) from which all agent keys are deterministically derived, allowing backup and recovery without storing raw keys.

What is NOT shared across agents: Holochain key pairs, source chain entries, Ethereum addresses, alt_bn128 identity key pairs, ElGamal ciphertexts, PS signature rerandomizations, NIZK proofs. **What IS shared** (locally, within Alice’s wallet, never on any network): Alice’s plaintext `identity_data` and the master PS signature (`m`, `sigma`) from the issuer.

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.

7.6 Summary: What Each Layer Sees

Layer	What it sees
Ethereum (everyone)	<code>Transfer(from, to, amount)</code> – pseudonymous addresses <code>BuckTransferReceipt</code> – same addresses + opaque hashes <code>IdentityExchange</code> – address + opaque ElGamal ctxt Chaum-Pedersen proof verified (pass/fail) – no content
BUCK contract	<code>isVerified(from) == true, isVerified(to) == true</code> 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

8 Cryptographic Receipts: 3-Party ECDH

While the BUCK contract emits only opaque ciphertext hashes on-chain, the Holochain wallet layer constructs full receipts containing both parties’ decrypted `identity_data` and the transfer

metadata. These receipts are encrypted using 3-party ECDH (sender + receiver + ephemeral transaction key) so that only the direct participants can decrypt. A regulator cannot decrypt unilaterally – they hold two of three ECDH components but lack the participants’ shared secret, making the surveillance cost $O(N)$ per participant rather than $O(1)$. The canonical receipt format (deterministic JSON with sorted keys) ensures both parties independently produce byte-identical receipts with matching keccak256 hashes, enabling dispute resolution without coordination.

8.1 The Three Keys

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

Key	Held by	Purpose
<code>k_s</code> (sender)	Alice	Sender’s identity private key (alt_bn128)
<code>k_e</code> (ephemeral)	Deterministic	Derived from hash of on-chain receipt event; reproducible by anyone who knows the event
<code>K_r</code> (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.

8.2 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
);
```

8.3 Who Can Decrypt

8.3.1 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)
```

8.3.2 Receiver (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)
```

8.3.3 Court-Compelled Decryption (Subpoena of a Counterparty)

A court subpoenas Bob (a verified counterparty to the transfer) to decrypt a specific transaction receipt. The pool itself cannot be subpoenaed – it has no decryption key. The only cooperators who can produce plaintext are the verified humans who exchanged the BUCK:

```
// 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.

8.3.4 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). They can also compute $\text{shared}_2 = \text{ECDH}(k_e, K_r)$ since the ephemeral key k_e is deterministic from public on-chain data. But they cannot compute $\text{shared}_1 = \text{ECDH}(sk_{\text{sender}}, K_{\text{receiver}})$ – this requires the sender's or receiver's private key. Without a participant's cooperation, the regulator holds two of three ECDH components and cannot reconstruct the encryption key. No unilateral surveillance.

This is the deliberate design: a court compels a *participant* to decrypt; the escrow key merely lets the regulator *verify* the decryption (by independently recomputing the same key once a participant cooperates). Contrast with custodial key escrow, where the authority holds a master decryption key and can surveil at $O(1)$ cost. Here the cost is $O(N)$ – one subpoena per participant, per transaction.

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

A complete BUCK/USDT swap, step by step: from Alice's initial KYC ceremony through the final on-chain transfer, showing every piece of data, where it resides, and what each party does. Alice is a private EOA; Bob's pool is a **Public-Identity contract** (`isPublicIdentity(pool) = true`, bound at deploy time via `bindContract(pool, pk_bob, E_bob, true)`). The Private-to-Public-Identity mode means Alice still calls `approve` (always CP-bound, $\sim 29\text{K}$ gas) – there is no `isPublicIdentity` shortcut at approve time – but no `approve` is needed from the pool's side because its receipt fragment falls back to the bound `_identityHash(pool)`. `transferFrom` performs only state lookups, no cryptography. Each step shows the on-chain gas cost, the Holochain wallet actions, and what an observer (Eve) can and cannot see.

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

9.1 Prerequisites (One-Time Setup)

Alice:

1. Completed KYC with trusted issuer (off-chain ceremony)
2. Issuer PS-signs Alice's identity: $\text{sigma} = \text{PS.Sign}(sk_{\text{issuer}}, m)$
where $m = H(\text{identity_data})$
3. Alice receives (m, sigma) and stores in Holochain wallet
4. Alice's wallet derives a credential for this account (offline):
 - Rerandomizes PS signature: $\text{sigma}' = (t * \text{sigma}_1, t * \text{sigma}_2)$
 - Generates fresh identity key pair $(sk_{\text{alice}}, pk_{\text{alice}})$
 - Encrypts: $E_{\text{alice}} = (r * G, m * G + r * pk_{\text{alice}})$
 - Computes NIZK proof π linking sigma' to E_{alice}
5. Registers on-chain ($\sim 235\text{K}$ gas):
`identityRegistry.registerCredential(
 alice_wallet, pk_alice, E_alice, sigma', pi)`
6. `identityRegistry.isVerified(alice_wallet) == true`

Pool operator Bob:

1. Completed KYC (same ceremony as Alice). His EOA is registered in IdentityRegistry with credential (pk_bob, E_bob).
2. Deploys the BUCK/USDT pool atomically with bindContract:
BuckAwareDeployer.deployAndBind(
 UniswapV2Factory.createPair.selector(BUCK, USDT),
 pk_bob, E_bob,
 isPublicIdentity_=true
)
-> pair at address 0xPOOL bound under Bob's published (pk, E),
 with isPublicIdentity[0xPOOL] = true and
 isVerified[0xPOOL] = true.
Atomic deploy+bind defends against first-binder-wins front-running.
3. Same atomic pattern for the router 0xROUTER.

After binding:

```
isVerified(0xPOOL) == true
isPublicIdentity(0xPOOL) == true
identityHash(0xPOOL) == keccak(pk_bob, E_bob)
-- Bob's EOA secret key sk_bob decrypts every ciphertext that
    uses the pool's bound identity.
```

9.2 The Swap Transaction

9.2.1 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.

Note: the pool (`0xPOOL`) is a **Public-Identity contract** – its bound (`pk`, `E`) is the operator’s published material in the `IdentityRegistry`. Alice’s wallet reads `pk_pool == pk_bob` from the registry, decrypts her own ciphertext to recover `M`, then re-encrypts under `pk_bob` to produce `E_bob` for the CP proof. The pool itself does not call `approve` back – the bilateral identity check in `transferFrom` is satisfied because `isPublicIdentity(0xPOOL) == true` lets the pool-side fragment fall back to `_identityHash(0xPOOL)`. `Approve`, however, was still fully CP-bound on Alice’s side.

9.2.2 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.

9.2.3 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");
}
```

```

bytes32 fromHash = _receiptFragments[from][to]; // keccak256(E_pool from Alice)
bytes32 toHash   = _receiptFragments[to][from]; // 0 -- pool never approved Alice
if (toHash == 0) {
    require(
        identityRegistry.isPublicIdentity(from)
        || identityRegistry.isPublicIdentity(to),
        "no receipt fragment and no public side"
    );
    toHash = identityRegistry.identityHash(to); // _identityHash(0xPOOL)
}
if (fromHash == 0) {
    fromHash = identityRegistry.identityHash(from);
}

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

emit BuckTransferReceipt(from, to, amount, fromHash, toHash);
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.

9.2.4 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 already knows her own identity (she submitted it at KYC):
alice_identity = alice_identity_data // local, never on-chain

// 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_data = receive_via_holochain(bob) // off-chain data channel
assert H(bob_identity_data) * G == M_bob // bind to on-chain proof

// 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)

```

9.2.5 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).

9.2.6 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

10 Edge Cases and Design Considerations

Practical deployment scenarios beyond the simple Alice-to-Bob model: flash loans (atomic borrow-repay cycles), multi-hop swaps (BUCK through intermediate pools), contract-to-contract transfers (yield aggregators), epoch-based credential expiry, Holochain integration details (private entry storage, agent isolation, DHT identity delivery and non-publication detection), the 3-party ECDH regulatory escrow model ($O(N)$ surveillance cost, no unilateral decryption), and the `alt_bn128` security horizon (~ 100 -bit pairing security, BLS12-381 migration path via EIP-2537).

10.1 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.

10.2 Multi-Hop Swaps

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

10.3 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.

10.4 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.

10.5 Gas Costs

The design has three gas-consuming identity operations:

Operation	Cost (~gas)	When
<code>registerCredential</code> with PS verify (PS pairing: ~181K + NIZK: ~44K)	~235,000 gas	Once per account
<code>approve</code> with Chaum-Pedersen verify (proof verify: ~29K + std approve: ~46K)	~75,000 gas	Once per pair
<code>BuckTransferReceipt</code> 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 (235,000 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.

10.6 Pool Operator Identity Binding

Under `bindContract`, the pool's identity is the operator's identity. `isVerified(OxPOOL) == true` and `isPublicIdentity(OxPOOL) == true` once the operator (or `BuckAwareDeployer.deployAndBind` atomically) has bound the pool to the operator's published (`pk`, `E`). The pool itself holds no decryption key; the operator does, and the operator's `sk` decrypts every receipt that uses `_identityHash(pool)`.

First-binder-wins. Because `bindContract` requires only that `target.code.length > 0` and that the address has not yet been bound, an attacker can race a passive deployment and bind a freshly deployed contract to *their* key before the deployer does. The defense is atomic `deploy+bind` via `BuckAwareDeployer`; passive `createPair` followed by a separate `bindContract` from the operator's wallet is unsafe in adversarial settings.

The factory transaction is on-chain and pseudonymous; the IdentityRegistry has no record of who deployed the contract. Once bound, the operator's identity (the EOA who registered (`pk_op`, `E_op`)) is the pool's accountability surface.

For commercial DeFi operators who want public branding ("DeFi Ops Inc., Alberta"), publication happens off-chain (their own website, the IdentityRegistry's optional `plaintextIdentity` claim on the *operator's EOA*, social channels) – `bindContract` supplies the on-chain link between contract and operator credential.

10.7 alt_bn128 Security Horizon

The `alt_bn128` curve (BN254) offers ~ 128 -bit security for discrete logarithm in G_1 but only ~ 100 -bit security against pairing-based attacks, due to Kim-Barbulescu tower-NFS improvements (2016) to the Number Field Sieve in extension fields. This affects the PS signature verification (which uses pairings at registration) but *not* the Chaum-Pedersen proof (which uses only G_1 scalar multiplications at transaction time and remains at full ~ 128 -bit security).

This is not specific to Alberta Buck. *Every* Ethereum application using the `ecPairing` precompile – Groth16 SNARK verifiers (zkSync, Polygon zkEVM), BLS signature schemes, any pairing-based construction – faces the same constraint. The `alt_bn128` precompiles are the only pairing-capable curve available on-chain until EIP-2537 (BLS12-381 precompiles) is deployed.

Three mitigations apply:

1. **Epoch-based credential expiry:** PS signatures are verified once at registration and carry an expiry epoch. An attacker who invests computational resources to break a pairing equation recovers a single credential that has likely already expired. The Identity Fountain's renewal model limits the value of any single break.
2. **DLog security is unaffected:** Per-transaction Chaum-Pedersen proofs rely on discrete logarithm hardness in G_1 , which remains at ~ 128 -bit security. An attacker who breaks pairings cannot forge re-encryption proofs or decrypt ElGamal ciphertexts.
3. **BLS12-381 migration path:** EIP-2537 introduces precompiles for BLS12-381 (128-bit pairing security). When deployed, credential registration migrates to BLS12-381 for PS verification while the Chaum-Pedersen layer continues on `alt_bn128` (or also migrates). The architecture's separation of registration-time pairings from transaction-time scalar multiplications makes this migration straightforward -- only the `registerCredential` function changes.

The ~ 100 -bit pairing security is adequate for the current threat model. The practical cost of a tower-NFS attack on BN254 pairings remains far beyond the value of any single Alberta Buck credential, particularly given epoch-based expiry. The system shares this constraint with the entire Ethereum L2 scaling ecosystem.

11 Alternative Approaches Considered

Five cryptographic alternatives were evaluated before settling on the ElGamal + Pointcheval-Sanders + Chaum-Pedersen architecture: Pedersen Commitments with Groth16 SNARKs, Proxy Re-Encryption (NuCypher Umbral), BBS+ Signatures with Selective Disclosure, Camenisch-Shoup Verifiable Encryption, and Pedersen Equality Proofs. Each is assessed on per-transaction gas cost, setup requirements, proof generation time, targeted decryption support, and implementation complexity. The comparison table shows ElGamal Chaum-Pedersen wins on per-transaction gas ($\sim 29K$ vs. $\sim 200K+$ for SNARK-based alternatives), requires no trusted setup, and generates proofs instantaneously. Combined with PS rerandomizable signatures for one-time credential registration ($\sim 235K$

gas), the system achieves both cross-account unlinkability and per-transaction re-encryption correctness without the gas burden or trusted setup of alternatives.

11.1 Pedersen Commitments + Groth16 ZK Proofs

In this approach, the issuer stores a Pedersen commitment $C = v \cdot G + r \cdot H$ 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.

11.2 Proxy Re-Encryption (NuCypher Umbral)⁵

In PRE, Alice generates a re-encryption key $rk_{\{A \rightarrow B\}}$ 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)

⁵Umbral: 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>

- 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.

11.3 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:

- 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 ($\sim 113,000+$ gas). Chaum-Pedersen uses only scalar multiplications ($\sim 29,000$ 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.

11.4 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.

11.5 Pedersen Equality Proofs (Homomorphic Comparison)⁶

Pedersen commitments support homomorphic equality checking: given $C_1 = v \cdot G + r_1 \cdot H$ and $C_2 = v \cdot G + r_2 \cdot H$, revealing $\delta_r = r_1 - r_2$ lets the verifier check $C_1 - C_2 == \delta_r \cdot H$ 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 δ_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).

⁶Pedersen 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/>

11.6 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)
Camensisch-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 ~235K 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.

12 Summary: Identity Requirement by Risk Level

The table below maps every Alberta Buck operation to its risk level, identity mode, and receipt format. The design principle: identity requirements scale with the damage a malicious actor could cause. Governance and BuckCredit operations require full public identity; transfers and DeFi swaps require anonymous-valid identity with encrypted receipts; read-only operations are permissionless. In all cases, the on-chain proof chain ensures that identity can be recovered under subpoena without requiring ongoing cooperation from any party.

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