Alberta Buck - Wormhole (DRAFT v0.1)

Identity-Preserving Transaction Graph Privacy

Perry Kundert

2026-04-10

Alberta Buck's identity system provides credential unlinkability and identity privacy, but the Ethereum transaction graph remains public: balances, transfers, and counterparty relationships are visible to all. This document proposes a *wormhole* mechanism – adapted from EIP-7503[1] – that provides transaction graph privacy while preserving the identity guarantees that distinguish BUCK from anonymous cryptocurrencies.

The wormhole operates in two phases. Phase 1 uses fixed-denomination burn-and-mint transfers: Alice sends BUCKs to a provably unspendable address (which holds a valid, identity-registered credential with the same M as her source account), then proves ownership via a SNARK that internally verifies a Chaum-Pedersen identity binding, and mints the same amount at a fresh address. The mint uses `_mint_noincrease` – no change to `totalSupply`, no Jubilee distortion, no PID controller interference. Phase 2 extends this with a shared note pool using Pedersen commitments, enabling amount splitting across time and addresses to defeat correlation analysis.

The wormhole exposes a deeper primitive: *Identity Guardians* – identity-based authorization that decouples ECDSA key possession from the right to act on an account. Burn addresses (no ECDSA private key) are handled via `approveFor` with Schnorr proof of the alt_bn128 identity key. The same mechanism enables cold wallet management, account recovery after key loss (a designated identity activates after a period of inactivity), and cosigned transfers (a spouse or partner must co-approve amounts above a threshold).

The key contribution over EIP-7503 and existing privacy pools (Tornado Cash, Zcash, Aztec) is *identity continuity*: the SNARK proves that a legitimately identified person deposited and withdrew, without revealing who. Regulators get identity recoverability at the destination address via the standard ElGamal proof chain. Users get transaction graph privacy. (PDF, Text)

[1] EIP-7503: Zero-Knowledge Wormholes. Kambakhsh, K. et al., 2023. A protocol-level mechanism for private ETH transfers via proof-of-burn, using ZK-SNARKs to prove that ETH was sent to a provably unspendable address and authorizing re-minting at a new address. `https://eips.ethereum.org/EIPS/eip-7503`

Contents

1 The Problem: Transaction Graph Leakage

Alberta Buck's identity architecture (Identity, Examples) solves identity privacy: no observer can determine who owns an address, and no observer can link two credentials to the same person. But all transactions occur on a public Ethereum ledger. Balances, transfer amounts, counterparty addresses, and timing are visible to everyone.

Standard chain analysis can reconstruct the flow of funds between addresses. Even without knowing *who* owns each address, the *pattern* of transactions reveals economic relationships: which addresses transact together, how funds flow through the system, and which addresses likely belong to the same entity (clustering heuristics).

This is a separate problem from identity privacy, requiring separate cryptographic machinery. The identity layer (Chaum-Pedersen, PS signatures, ElGamal) ensures no one learns your name from an address. A privacy layer must ensure no one learns your address from your transactions.

2 Economic Model: Demurrage-Compatible Privacy

BUCK has economic properties that both complicate and enable a wormhole mechanism.

2.1 Demurrage and Jubilee

Every BUCK balance is subject to demurrage: 2% per year, linear (not compounding), computed lazily at transfer-out time. The contract stores two values per account: the balance, and the *BUCK-age* – a cumulative balance-time accumulator (in BUCK-years) updated at every balance-changing event. Demurrage owed at transfer-out is:

$$\text{demurrage} = \text{BUCK-age} \times 0.02$$

For a constant balance B held for T years: BUCK-age $= B \times T$, so demurrage $= 0.02BT$. After 50 years the entire balance is consumed. The *net available* at any moment is:

$$\text{net\_available} = B - 0.02 \times \text{BUCK-age}$$

The Jubilee account grows as a function of `totalSupply` and the demurrage rate. It represents the accumulated demurrage across all BUCK holders, redistributed to the BUCK issuers who pledged assets to create the BUCK liquidity in circulation.

2.2 Why `_mint_noincrease` Works

A standard burn-and-mint wormhole (as in EIP-7503) burns ETH at one address and mints it at another. For ETH, this is supply-neutral because the burn reduces supply and the mint restores it.

If we used standard `_mint` (which increments `totalSupply`), the total nominal supply would increase – the burn address keeps its balance AND new BUCKs are created.

The solution: `_mint_noincrease`.

```
Standard _mint:
  _balances[to] += amount;
  _totalSupply  += amount;       // supply increases

_mint_noincrease(to, amount, buckAge):
```

```
_balances[to] += amount;
_buckAge[to]   = buckAge;      // set arbitrary BUCK-age
// _totalSupply unchanged      // supply stays constant
```

The critical detail: `_mint_noincrease` sets *both* the balance and the BUCK-age at the destination. The wormhole caller is free to choose any BUCK-age for the destination account, provided the SNARK proves the conservation constraint (see Phase 1 below).

After a wormhole transfer:

- `sum(balances) = totalSupply + dead_burn_sum`
- The difference (`dead_burn_sum`) is auditable from `WormholeTransfer` event logs
- Jubilee growth is computed from `totalSupply` – unaffected
- The BUCK_K PID controller uses `totalSupply` – unaffected
- The burn address's balance is phantom: it exists in state but is economically inert

The wormhole user creates a "ghost balance" – the burned balance is permanently lost, but must continue to appear to all outside observers to be a valid, potentially usable sum of BUCKs. It sits in contract storage, accruing phantom demurrage that is never collected. Over 50 years, it would have reached zero anyway; instead, it simply never participates in the economy again.

2.3 Supply Accounting

Metric	Before wormhole	After wormhole	Change
`totalSupply`	S	S	none
`sum(balances)`	S	S + burn_amount	+burn_amount
Live circulating	S	S	none
Dead (burn addresses)	0	burn_amount	+burn_amount
Jubilee growth rate	f(S)	f(S)	none
PID controller input	S	S	none

The only observable on-chain effect: `sum(balances)` exceeds `totalSupply` by exactly the cumulative wormholed amount. This is intentional and auditable.

3 Protocol Design

3.1 Phase 1: Fixed-Denomination Wormholes

Fixed denominations eliminate amount correlation. All wormhole transfers use standard amounts (e.g., 100, 1,000, 10,000 BUCK), so observers cannot match a specific burn to a specific mint by amount alone.

But fixed denominations create a *change problem*: Alice's net available balance rarely decomposes exactly into standard denominations. BUCK-age solves this.

Since `_mint_noincrease` controls both the balance and the BUCK-age at the destination, Alice can choose *any* (amount, BUCK-age) pair for each minted denomination, provided the net-available values sum correctly:

$$\sum_i (\text{amount}_i - 0.02 \times \text{buckAge}_i) = \text{net\_available}_{\text{source}}$$

The BUCK-age is a continuous parameter that absorbs any remainder. Like withdrawing cash from an ATM, the burner chooses a mix of BUCK SNARK "denominations" (10,000, 1,000, 100, 10, and 1) – but *unlike* cash, each denomination carries a freely chosen BUCK-age that shifts how much net value it represents.

Source: 2,500 BUCK held 10 years	Net available: 2,000 BUCK	
Denomination	BUCK-age (equiv years)	Net after demurrage
1,000	0 (fresh)	1,000
1,000	5,000 (5 yrs)	900
100	0 (fresh)	100
Total		2,000

This has three privacy benefits:

1. **Amount correlation defeated**: every wormhole transfer is a standard denomination (1,000 BUCK looks like any other 1,000 BUCK).
2. **Age correlation defeated**: the destination BUCK-age bears no relation to the source BUCK-age. Eve cannot match burns to mints by observing account ages.
3. **Exact decomposition**: the continuous BUCK-age parameter eliminates the need for non-standard "change" amounts that would fingerprint the source.

It is recommended that these SNARK wormhole certificates be deposited into several independent Ethereum accounts associated with the same identity, to further shroud their relation to any existing potential burn account values.

3.1.1 Step 1: Burn

Alice holds BUCKs at `addr_source` with identity credential `(pk_s, E_s, sigma'_s)`. She derives a *second* credential from her Identity Fountain – same $M = m \cdot G$, fresh keys, fresh rerandomized signature – and registers it at a burn address.

```
Burn address derivation (EIP-7503 style):
  secret   = random 256-bit value
  addr_burn = sha256(MAGIC_BURN || secret)[12:]   // last 20 bytes

SHA-256 produces the address, not Keccak-256.
Since Ethereum derives addresses from Keccak-256(pubkey),
no ECDSA private key maps to this address.
addr_burn is provably unspendable.
```

Alice registers an identity credential at `addr_burn` via `registerCredentialFor(addr_burn, pk_b, E_b, sigma'_b, pi_b)`. The NIZK proof π_b proves the credential is issuer-signed, so registration succeeds even without a transaction from `addr_burn`.

Alice then executes a normal BUCK transfer:

1. `buck.approve(addr_burn, amount, E_for_burn, chaumPedersenProof)`
2. `buck.transfer(addr_burn, amount)`

To every observer, this looks like Alice paying someone. The burn address has a registered identity, participates in a Chaum-Pedersen identity exchange, and receives a standard transfer. Nothing distinguishes it from any other private BUCK account – except that no one ever transacts from it (indistinguishable from an idle wallet).

3.1.2 Step 2: ZK Proof

Alice constructs a SNARK whose *private witness* contains:

```
1. secret                        -- preimage of addr_burn
2. merkle_proof                  -- proof that addr_burn has (balance, buckAge)
                                    in Ethereum's state trie
3. source_balance, source_age    -- balance and BUCK-age at burn address
4. E_source                      -- ElGamal ciphertext at source address
5. E_dest                        -- ElGamal ciphertext at destination address
6. (k1, k2, s1, s2, e)           -- Chaum-Pedersen proof that E_source and
                                    E_dest encrypt the same M
7. sigma'_source, sigma'_dest    -- PS signatures (issuer-verified)
```

The SNARK circuit internally verifies:

- `sha256(MAGIC_BURN || secret)[12:] = addr_burn` (burn address is valid)
- Merkle proof against a recent state root (burn address holds `source_balance` with `source_age`)
- **Conservation**: `dest_amount - 0.02 * dest_buckAge <` source_balance - 0.02 * source_age= (net available is conserved or reduced; any shortfall is demurrage legitimately owed)
- Chaum-Pedersen equations (same identity on both sides)
- PS pairing check (both credentials are issuer-signed)
- Nullifier = `sha256(MAGIC_NULL || secret)` (computed correctly)

Public inputs (what the EVM sees):


```
nullifier    = sha256(MAGIC_NULL || secret)
amount       = one of {1, 10, 100, 1000, 10000}  // fixed denomination
buck_age     = chosen BUCK-age for destination    // continuous parameter
destination  = addr_dest
state_root   = recent block's state root
snark_proof  = pi
```

Note: a single burn address may generate *multiple* SNARKs (one per denomination), each with its own nullifier derived from =sha256(MAGIC_NULL

secret index)=. The SNARK proves that the cumulative net-available

across all certificates does not exceed the source's net-available.

3.1.3 Step 3: Mint

```
function wormholeTransfer(
    uint256       amount,       // must be a valid denomination
    uint256       buckAge,      // chosen BUCK-age (in BUCK-years, scaled)
    address       destination,
    bytes32       nullifier,
    bytes32       stateRoot,
```

```
    bytes calldata proof
) external {
    require(validDenomination(amount));
    require(!nullifierUsed[nullifier]);
    require(isRecentStateRoot(stateRoot));
    require(identityRegistry.isVerified(destination));
    require(verifySNARK(proof, nullifier, amount, buckAge,
                        destination, stateRoot));

    nullifierUsed[nullifier] = true;
    _mint_noincrease(destination, amount, buckAge);

    emit WormholeTransfer(nullifier, destination, amount, buckAge);
}
```

The `buckAge` parameter is the freely chosen BUCK-age for the destination account. The SNARK proves that the net-available (`amount - 0.02 * buckAge`) does not exceed the source's net-available, and that the amount is a valid denomination. The BUCK-age is *not* required to be a denomination – it is a continuous value that absorbs the remainder when decomposing an arbitrary source balance into fixed-denomination certificates.

The destination address must have a valid identity registration. The SNARK proves (internally) that this identity shares the same M as the source. But the verifier (the EVM) sees only: "a valid proof exists, the nullifier is fresh, and the destination has a registered identity."

3.2 Phase 2: Note-Based Privacy Pool

Fixed denominations solve amount correlation but force awkward splitting (moving 1,337 BUCK requires one 1,000 + three 100 + manual handling of the 37 BUCK residual). Phase 2 introduces a shared pool of encrypted note commitments, enabling arbitrary amounts and splitting across time.

3.2.1 Architecture

A shared Merkle tree stores Pedersen commitments to note values. Each note commits to a value and a secret:

$$C = v \cdot G + r \cdot H$$

where G is the standard BN254 generator and H is a second generator with unknown discrete log relative to G (derived from a hash: $H = \text{HashToPoint}(\texttt{"BUCK\_PEDERSEN\_H"})$).

3.2.2 Deposit

Alice sends X BUCK to the WormholePool contract. The contract records a note commitment $C = X \cdot G + r \cdot H$ in a Merkle tree. Alice keeps (X, r) private.

The deposit requires identity verification: Alice's `approve` to the pool address includes a Chaum-Pedersen proof. The pool is a public account, so only Alice's side of the identity handshake is needed.

3.2.3 Split

Alice can split her note into two (or more) notes without interacting with the contract:

```
Original note:  C   = 1000*G + r*H
Split into:     C_1 = 400*G  + r_1*H
                C_2 = 600*G  + r_2*H
Constraint:     r_1 + r_2 = r  (mod ORDER)
```

Pedersen commitments are additively homomorphic: $C_1 + C_2 = (400+600) \cdot G + (r_1+r_2) \cdot H = C$. A SNARK proves:

- The original note exists in the Merkle tree (membership proof)
- The two new notes sum to the original (homomorphic balance)
- Both new values are positive (range proofs)
- A nullifier for the spent note is correctly derived

The two new notes are inserted into the Merkle tree. The old note's nullifier prevents double-spending. No observer can link the new notes to the old one – the Merkle tree contains notes from all users, and the SNARK hides which note was spent.

3.2.4 Withdraw

Alice proves she owns a note of value V in the Merkle tree and withdraws V BUCK to a fresh address. The SNARK additionally proves Chaum-Pedersen identity binding: the withdrawal address has the same M as the deposit address.

```
function withdrawFromPool(
    uint256       amount,
    address       destination,
    bytes32       nullifier,
    bytes32       merkleRoot,
    bytes calldata proof
) external {
    require(!nullifierUsed[nullifier]);
    require(validMerkleRoot(merkleRoot));
    require(identityRegistry.isVerified(destination));
    require(verifySNARK(proof, nullifier, amount, destination, merkleRoot));

    nullifierUsed[nullifier] = true;
    _mint_noincrease(destination, amount);

    emit PoolWithdraw(nullifier, destination, amount);
}
```

With many users depositing and withdrawing, Alice's 400 BUCK withdrawal cannot be linked to her 1,000 BUCK deposit – other users' operations interleave in the shared tree, and the SNARK hides which note was consumed.

3.2.5 Demurrage Inside the Pool

Without countermeasures, the pool would become a demurrage-avoidance mechanism (deposit BUCKs, wait, withdraw without paying demurrage). Each note records its deposit timestamp. At withdrawal, the SNARK computes demurrage owed:

$$\text{claimable} = \text{value} - \text{value} \times 0.02 \times \text{years\_idle}$$

The range proof ensures claimable > 0 (i.e., the note hasn't expired). After 50 years, a note is worthless.

4 Identity-Authorized Operations

Burn addresses expose a fundamental issue: the BUCK bilateral `approve` requires both parties to sign Ethereum transactions, but burn addresses have no ECDSA private key. The solution – Schnorr-authorized identity delegation – turns out to solve a much broader class of problems.

4.1 The Problem: Two Key Systems, One Authorization Model

BUCK accounts live at the intersection of two independent key systems:

Key system	Curve	Controls	Who holds it
ECDSA	secp256k1	Ethereum transactions	Address owner
Identity (BN254)	alt_bn128	Identity credentials, CP proofs	Identity holder

Normally, the same person holds both keys. But they can diverge:

- **Burn addresses**: Identity key exists (Alice generated it), ECDSA key does not (SHA-256 derivation)
- **Cold wallets**: ECDSA key exists but is offline/inaccessible
- **Lost keys**: ECDSA key is permanently lost, identity persists via issuer re-issuance
- **Smart contract wallets**: No ECDSA key at all (contracts can't sign as EOAs)

The current BUCK contract checks only `msg.sender` – the ECDSA key. If that key is unavailable, no operation is possible, even if the caller can prove they hold the *identity* that the address represents.

4.2 `approveFor`: Schnorr-Authorized Identity Exchange

A Schnorr proof of knowledge of sk such that $sk \cdot G = pk$ authorizes the caller to act on behalf of any address whose registered identity public key is pk. The proof demonstrates that the caller controls the identity credential – the alt_bn128 secret key – regardless of whether they hold the ECDSA key.

$$\begin{aligned}
\text{Commit:} &\quad T = k \cdot G \\
\text{Challenge:} &\quad e = H(pk,\, T,\, \texttt{msg.sender},\, \texttt{principal}) \\
\text{Respond:} &\quad s = k - e \cdot sk \pmod{q} \\
\text{Verify:} &\quad s \cdot G + e \cdot pk \stackrel{?}{=} T
\end{aligned}$$

Cost: ~12,000 gas (2 `ecMul` + 1 `ecAdd` + 1 `keccak256`).

4.2.1 Why `approveFor` Cannot Be Used Against Normal Addresses

`approveFor` looks up the *pk already registered* at the principal address, then requires a Schnorr proof for that specific key. The security boundary is upstream, at credential registration:

- **Normal addresses**: The owner calls `registerCredential(pk, E, \sigma', \pi)`, which binds the credential to `msg.sender`. Only the ECDSA key holder can register. Since the owner chose sk privately and published only $pk = sk \cdot G$, no one else can produce a valid Schnorr proof (discrete log hardness on alt_bn128, ~128 bits).
- **Burn addresses**: No ECDSA key exists, so `registerCredential` can never be called from them. Alice uses `registerCredentialFor(addr\_burn, pk, E, \sigma', \pi)` instead. Since she generated sk_{burn} herself, she can pass the Schnorr check.

The critical invariant: `registerCredentialFor` **must reject if a credential already exists at the target address.** This prevents front-running: Eve cannot race to register her own credential at Bob's address before Bob does, because Bob registers (via `msg.sender`-gated `registerCredential`) as part of onboarding, before he can receive any transfers.

```
function registerCredentialFor(address target, ...) external {
    require(identityRegistry.getIdentityKey(target) == ZERO,
            "credential already exists");  // ← the guard
    // ... PS signature + NIZK verification, then store
}
```


The chain of custody:

```
Normal address:  registerCredential (msg.sender gate)
                 → only owner knows sk
                 → approveFor rejects outsiders

Burn address:    registerCredentialFor (no prior credential)
                 → creator knows sk
                 → approveFor works for creator
```

```
function approveFor(
    address principal,              // the address being represented
    address counterparty,           // who principal is approving
    uint256 amount,
    uint256[4] calldata E_counterparty,  // re-encrypted identity
    uint256[3] calldata chaumPedersen,   // same-M proof
    uint256[2] calldata schnorrAuth     // proof of knowledge of sk
) external {
    // 1. Schnorr: caller knows sk such that sk * G == pk_principal
    uint256[2] memory pk = identityRegistry.getIdentityKey(principal);
    require(verifySchnorr(schnorrAuth, pk, msg.sender, principal));

    // 2. Read E_principal from storage (not calldata)
    uint256[4] memory E_principal = identityRegistry.getCredential(principal);
```

```
    // 3. Chaum-Pedersen: same M
    require(verifyChaumPedersen(chaumPedersen, E_principal, E_counterparty,
                                pk, identityRegistry.getIdentityKey(counterparty)));

    // 4. Record the identity exchange
    _receiptFragments[principal][counterparty] = keccak256(abi.encode(E_counterparty));

    emit IdentityExchange(principal, counterparty, E_counterparty);
}
```

For the wormhole burn, Alice calls three functions:

```
1. registerCredentialFor(addr_burn, pk_b, E_b, '_b, _b)
   -- registers identity at burn address (NIZK proves issuer-signed)

2. approveFor(addr_burn, alice, 0, E_burn_for_alice, CP_proof, schnorr)
   -- addr_burn's half of bilateral exchange (Alice knows sk_burn)

3. approve(addr_burn, amount, E_alice_for_burn, CP_proof)
   -- Alice's half of bilateral exchange (normal approve)

4. transfer(addr_burn, amount)
   -- both _receiptFragments populated → bilateral check passes
```


To an observer, this is a normal private-to-private transfer. The `approveFor` call is indistinguishable from someone managing a cold wallet or a contract-held account.

4.3 The General Pattern: Identity Guardians

`approveFor` solves the burn address problem, but it is a special case of a more powerful primitive: *identity-based authorization*. Instead of asking "does `msg.sender` match the target address?" the contract can ask "does the caller hold an identity that is *authorized* to act on this account?"

An account can designate **guardians** – identified persons authorized to perform specific operations under specific conditions:

```
enum GuardianType { SELF, RECOVERY, COSIGNER }

struct Guardian {
    address          guardianAddress;  // must have registered identity
    GuardianType     guardianType;
    uint256          activationParam;  // delay (RECOVERY) or threshold (COSIGNER)
}

// mapping: principal → guardian list
mapping(address => Guardian[]) public guardians;
```

Type	Who	When active	What they can do
`SELF`	Same identity (same M)	Always	Full identity operations
`RECOVERY`	Designated person	After N days of inactivity	Redirect funds to new address
`COSIGNER`	Designated person	Transfers above threshold	Must co-approve large sends

4.3.1 Use Case: Wormhole Burns (SELF)

Alice holds the identity credential for `addr_burn` – same M, different keys. She proves knowledge of `sk_burn` via Schnorr and completes the bilateral identity exchange. No guardian designation needed: the contract can verify same-identity by accepting a Chaum-Pedersen proof that the caller's credential and the principal's credential share the same M.

4.3.2 Use Case: Cold Wallet Management (SELF)

Alice keeps her savings in a cold wallet (`addr_cold`). She derives a hot-wallet credential from the same identity (same M, fresh keys) and registers it at `addr_hot`. From `addr_hot`, she can call `approveFor(addr_cold, ...)` to manage identity exchanges for the cold wallet without ever exposing the cold wallet's ECDSA key.

The cold wallet's ECDSA key stays offline. Identity operations happen from the hot wallet. Funds remain in cold storage until Alice signs a transfer with the cold key.

4.3.3 Use Case: Account Recovery (RECOVERY)

Alice designates her spouse as a recovery guardian with a 90-day activation delay:

```
buck.designateGuardian(
    spouse_address,         // must have registered identity
    GuardianType.RECOVERY,
    90 days                 // activation delay
)
```


If Alice loses her ECDSA key:

1. She obtains a fresh PS signature from the issuer (re-issuance, same m, new credential)
2. She registers the new credential at `addr_new`
3. After 90 days of inactivity at `addr_old`, the recovery guardian activates
4. Spouse calls `recoverAccount(addr_old, addr_new, schnorr_proof)`

```
function recoverAccount(
    address oldAccount,
    address newAccount,
    uint256[2] calldata schnorrAuth
) external {
    Guardian memory g = getGuardian(oldAccount, msg.sender);
    require(g.guardianType == GuardianType.RECOVERY);
    require(block.timestamp > lastActive[oldAccount] + g.activationParam);

    // Verify caller is the designated guardian
    uint256[2] memory guardianPk = identityRegistry.getIdentityKey(msg.sender);
    require(verifySchnorr(schnorrAuth, guardianPk, msg.sender, oldAccount));

    // Transfer entire balance to new account
    require(identityRegistry.isVerified(newAccount));
```

```
    uint256 balance = balanceOf(oldAccount);
    _transfer(oldAccount, newAccount, balance);

    emit AccountRecovered(oldAccount, newAccount, msg.sender, balance);
}
```

Security properties:

- The 90-day delay prevents premature activation (Alice can cancel by transacting from `addr_old`)
- The guardian is identified – if the spouse acts maliciously, their identity is recoverable via the standard proof chain
- The new account must have a registered identity, maintaining BUCK's identity invariant
- The guardian cannot redirect to their own unregistered address (`isVerified` check)

4.3.4 Use Case: Cosigned Transfers (`COSIGNER`)

Alice designates her spouse as a cosigner for transfers above 10,000 BUCK:

```
buck.designateGuardian(
    spouse_address,
    GuardianType.COSIGNER,
    10_000 * 10**18        // threshold in wei
)
```


For a 50,000 BUCK transfer to Bob:

```
1. Alice:  approve(bob, 50000, ...)          // normal identity exchange
2. Spouse: cosign(alice, bob, 50000, proof)   // cosigner approval
3. Alice:  transfer(bob, 50000)               // proceeds with both approvals

function cosign(
    address principal,
    address recipient,
    uint256 amount,
    uint256[2] calldata schnorrAuth
) external {
    Guardian memory g = getGuardian(principal, msg.sender);
    require(g.guardianType == GuardianType.COSIGNER);
    require(amount >= g.activationParam);  // above threshold

    uint256[2] memory pk = identityRegistry.getIdentityKey(msg.sender);
    require(verifySchnorr(schnorrAuth, pk, msg.sender, principal));

    cosignatures[principal][recipient] = amount;
    emit Cosigned(principal, recipient, amount, msg.sender);
}
```

The `transfer` function checks: if the principal has a cosigner guardian and the amount exceeds the threshold, require a matching cosignature. Below the threshold, transfers proceed normally.

Applications:

- Spousal co-authorization for large purchases
- Corporate expense policies (CFO must approve above a limit)
- Parental controls on youth accounts
- Multisig-like security without multisig wallet complexity

4.4 The Authorization Check (General Form)

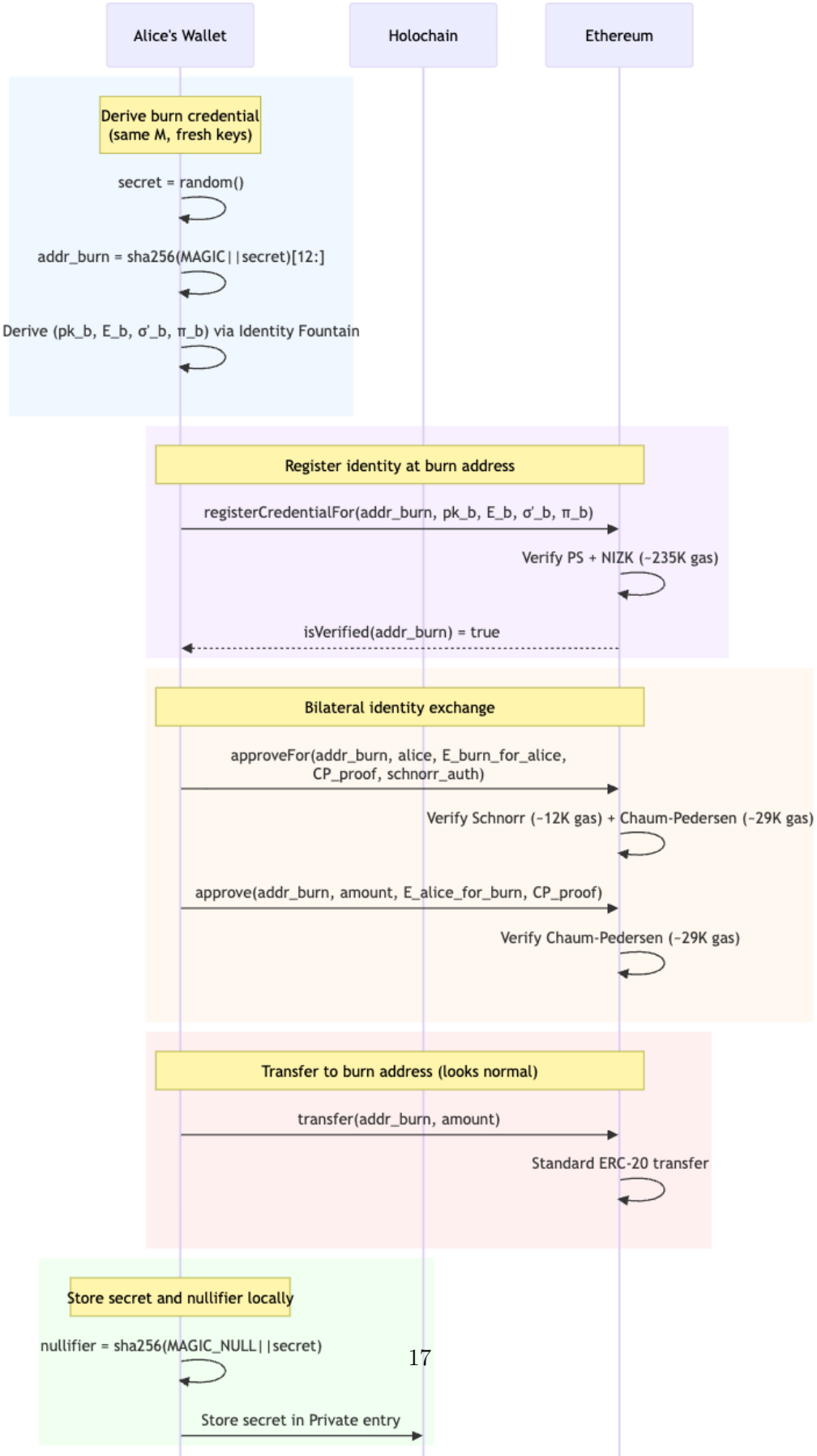
The BUCK contract's authorization logic generalizes from a simple `msg.sender` check to an identity-aware policy:

```
function isAuthorizedFor(
    address actor,
    address principal,
    ActionType action,
    uint256 amount
) internal view returns (bool) {
    // 1. Direct: actor IS the principal (standard case)
    if (actor == principal) return true;

    // 2. Self-identity: actor proved same M via approveFor
    //    (verified at approveFor call time, recorded in state)
    if (selfIdentityAuthorized[principal][actor]) return true;

    // 3. Guardian policies
    Guardian memory g = getGuardian(principal, actor);
    if (g.guardianType == RECOVERY
            && block.timestamp > lastActive[principal] + g.activationParam)
        return true;
    if (g.guardianType == COSIGNER
            && action == ActionType.COSIGN
            && amount >= g.activationParam)
        return true;

    return false;
}
```

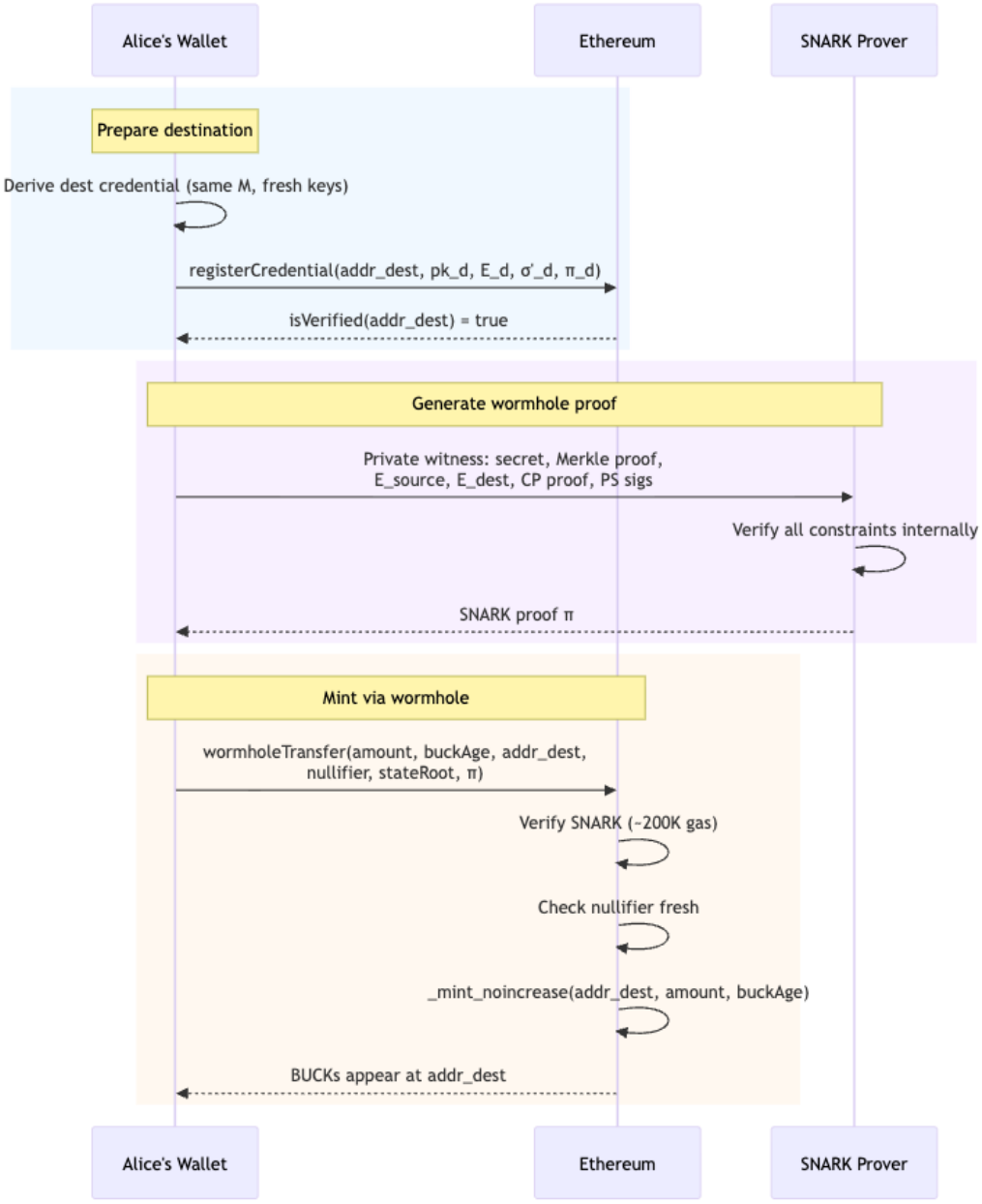

This pattern is not wormhole-specific. It extends the entire BUCK contract with identity-based authorization, using the same cryptographic primitives (Schnorr proofs, Chaum-Pedersen, ElGamal) that the identity system already provides.

5 Data Flows

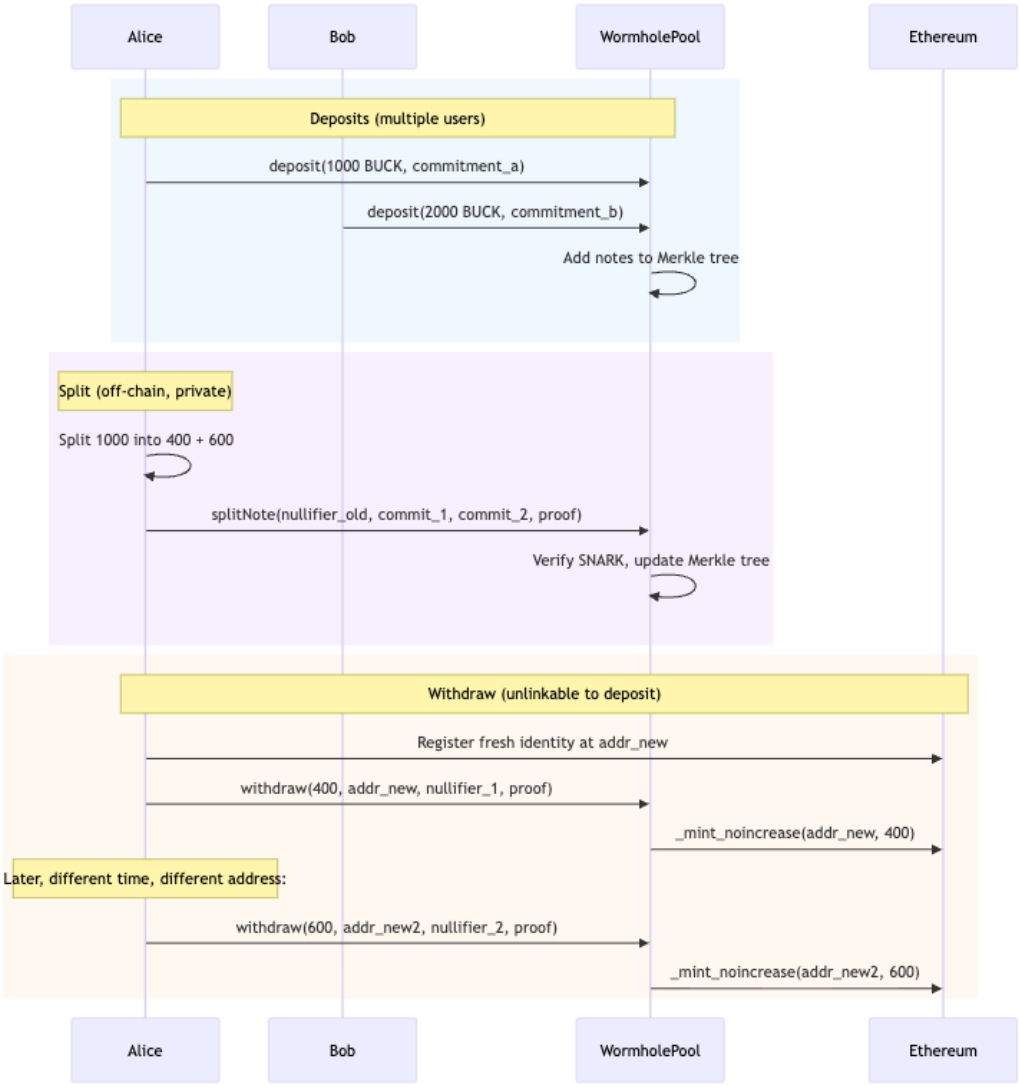
5.1 Wormhole Burn



5.2 Wormhole Mint



5.3 Privacy Pool Flow



6 Worked Cryptographic Examples

These examples implement the wormhole-specific cryptographic components on the alt_bn128 (BN254) curve. For the identity primitives they build on (PS signatures, ElGamal, NIZK, Chaum-Pedersen), see the Identity Examples document.

6.1 Setup

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

ORDER = bc.curve_order
G1 = bc.G1; Z1 = bc.Z1; G2 = bc.G2
multiply = bc.multiply; add = bc.add
neg = bc.neg; eq = bc.eq
```

```
def pt(P, label=""):
    if P == Z1: return f"{label}O (point at infinity)"
    return f"{label}({int(P[0]) % 10**6:>06d}..., {int(P[1]) % 10**6:>06d}...)"

def rand_scalar():
    return secrets.randbelow(ORDER - 1) + 1

# Second generator for Pedersen commitments (nothing-up-my-sleeve)
H = multiply(G1, int(Web3.keccak(text="BUCK_PEDERSEN_H").hex(), 16) % ORDER)

print(f"BN254 curve order: {ORDER}")
print(f"G1 generator: {pt(G1)}")
print(f"H  generator: {pt(H, 'H=')}")
```

```
BN254 curve order: 21888242871839275222246405745257275088548364400416034343698204186575808495617
G1 generator: (000001..., 000002...)
H  generator: H=(749662..., 874314...)
```

6.2 Burn Address and Nullifier

The burn address is derived via SHA-256 (not Keccak-256). Since Ethereum derives EOA addresses from Keccak-256 of a public key, no ECDSA private key can produce a SHA-256-derived address. The address is provably unspendable.

```
MAGIC_BURN = b"BUCK_WORMHOLE_BURN_V1"
MAGIC_NULL = b"BUCK_WORMHOLE_NULL_V1"

secret = secrets.token_bytes(32)

# Burn address: last 20 bytes of SHA-256
burn_hash = hashlib.sha256(MAGIC_BURN + secret).digest()
addr_burn = "0x" + burn_hash[-20:].hex()

# Nullifier: prevents double-mint
nullifier = hashlib.sha256(MAGIC_NULL + secret).digest()

# Demonstrate that SHA-256 and Keccak-256 produce different addresses
keccak_of_same = Web3.keccak(burn_hash[-20:]).hex()[-40:]

print(f"secret:       {secret.hex()[:40]}...")
print(f"addr_burn:    {addr_burn}")
print(f"nullifier:    0x{nullifier.hex()[:40]}...")
print(f"\nSHA-256 addr:  {addr_burn}")
print(f"Keccak check:  0x{keccak_of_same}")
print(f"Different hash families => no ECDSA key maps to addr_burn")
```



```
secret:       08d10e13b4e68cd7d6467a0fa31deface205cdc1...
addr_burn:    0x0b8bfe1dfd0b6d9fa2083e87a70a8847eb6effd8
nullifier:    0xd4c87041192b00fbc44ed3429a03f47029466726...

SHA-256 addr:  0x0b8bfe1dfd0b6d9fa2083e87a70a8847eb6effd8
Keccak check:  0xaee0528ead5927e2ba24370132499017f048d3a6
Different hash families => no ECDSA key maps to addr_burn
```

How to read the output: The SHA-256 address and the Keccak check must be different. Since Ethereum derives addresses exclusively from Keccak-256 of ECDSA public keys, no private key can produce the SHA-256-derived address. The burn address is a one-way trap.

6.3 BUCK-age Decomposition

Alice holds 2,500 BUCK with a BUCK-age of 25,000 BUCK-years (held at constant balance for 10 years). She decomposes this into fixed-denomination wormhole certificates with freely chosen BUCK-ages.

```
# Source account
source_balance = 2500
source_buckage = 25000.0      # BUCK-years (2500 * 10 years)
rate = 0.02

source_net = source_balance - rate * source_buckage
print(f"Source: {source_balance:,} BUCK, BUCK-age {source_buckage:,.1f} BUCK-yrs")
print(f"  Demurrage owed:  {rate * source_buckage:,.2f}")
print(f"  Net available:   {source_net:,.2f}")

# Decompose into fixed denominations with chosen BUCK-ages.
# Constraint: sum(amount_i - 0.02 * buckAge_i) = source_net
# The BUCK-ages are continuous -- they absorb the remainder.
denominations = [
    (1000, 0.0),        # brand new -- full 1,000 net
    (1000, 5000.0),     # appears 5 years old -- 900 net
    (100,  0.0),        # brand new -- full 100 net
]

print(f"\nWormhole certificates:")
print(f"  {'Denomination':>14s}  {'BUCK-age':>14s}  {'Equiv yrs':>10s}"
      f"  {'Demurrage':>10s}  {'Net':>10s}")
print(f"  {'-'*14}  {'-'*14}  {'-'*10}  {'-'*10}  {'-'*10}")
total_net = 0
for amt, age in denominations:
    dem = rate * age
    net = amt - dem
    total_net += net
    equiv_yrs = age / amt if amt > 0 else 0
    print(f"  {amt:14,d}  {age:14,.1f}  {equiv_yrs:10.1f}  {dem:10.2f}  {net:10.2f}")
print(f"  {'-'*14}  {'-'*14}  {'-'*10}  {'-'*10}  {'-'*10}")
print(f"  {'':14s}  {'':14s}  {'':10s}  {'Total:':>10s}  {total_net:10.2f}")

match = abs(total_net - source_net) < 0.01
print(f"\n  Conservation check: {total_net:,.2f} == {source_net:,.2f}?  {match}")
print(f"\n  Source held 2,500 BUCK for 10 years (net 2,000).")
print(f"  Destination: three certificates at different addresses:")
print(f"    - 1,000 BUCK appearing brand-new")
print(f"    - 1,000 BUCK appearing 5 years old (absorbs 100 of demurrage)")
print(f"    - 100 BUCK appearing brand-new")
print(f"  No non-standard denominations.  BUCK-age absorbs the remainder.")
print(f"  An observer cannot correlate by amount OR by account age.")
```

```
Source: 2,500 BUCK, BUCK-age 25,000.0 BUCK-yrs
  Demurrage owed:  500.00
  Net available:   2,000.00

Wormhole certificates:
    Denomination        BUCK-age   Equiv yrs   Demurrage         Net
  --------------  --------------  ----------  ----------  ----------
       1,000             0.0         0.0         0.00     1000.00
       1,000         5,000.0         5.0       100.00      900.00
  100             0.0         0.0         0.00      100.00
  --------------  --------------  ----------  ----------  ----------
           Total:     2000.00

  Conservation check: 2,000.00 == 2,000.00?  True

  Source held 2,500 BUCK for 10 years (net 2,000).
  Destination: three certificates at different addresses:
```

```
    - 1,000 BUCK appearing brand-new
    - 1,000 BUCK appearing 5 years old (absorbs 100 of demurrage)
    - 100 BUCK appearing brand-new
  No non-standard denominations.  BUCK-age absorbs the remainder.
  An observer cannot correlate by amount OR by account age.
```

How to read the output: Every denomination is standard (1,000 or 100), yet the net-available sums match exactly. The second certificate's BUCK-age (5,000 BUCK-years, equivalent to 5 years at 1,000 BUCK) absorbs part of the source's accrued demurrage – no non-standard "change" amount is needed. An observer sees three ordinary-looking wormhole transfers at standard denominations, each deposited into a different account with a plausible BUCK-age.

6.4 Three Identity Credentials (Same M, Unlinkable)

Alice derives three credentials from her Identity Fountain – one for the source account, one for the burn address, one for the destination. All three share the same identity scalar $m = H(\text{identity\_data})$, but every other value (keys, ciphertext, signature) is independent.

```
# Issuer key generation
x = rand_scalar(); y = rand_scalar()
X = multiply(G2, x); Y = multiply(G2, y)

# Alice's identity
identity_data = '{"name":"Alice","jurisdiction":"AB","id":"A1234567"}'
m = int(Web3.keccak(text=identity_data).hex(), 16) % ORDER
M = multiply(G1, m)

# PS signature on m
h = multiply(G1, rand_scalar())
sigma = (h, multiply(h, (x + m * y) % ORDER))

# Derive three unlinkable credentials
creds = {}
for label in ["source", "burn", "dest"]:
    t = rand_scalar()
    sigma_p = (multiply(sigma[0], t), multiply(sigma[1], t))
    sk = rand_scalar(); pk = multiply(G1, sk)
    r = rand_scalar(); R = multiply(G1, r)
    C = add(multiply(G1, m), multiply(pk, r))
    creds[label] = dict(sk=sk, pk=pk, r=r, R=R, C=C, sigma=sigma_p)

print("Three credentials derived from the same identity:")
for label, c in creds.items():
    print(f"\n  {label}:")
    print(f"    pk    = {pt(c['pk'])}")
    print(f"    E.R   = {pt(c['R'])}")
    print(f"    E.C   = {pt(c['C'])}")
    print(f"    sig_1 = {pt(c['sigma'][0])}")

# Verify all decrypt to the same M
for label, c in creds.items():
    M_dec = add(c['C'], neg(multiply(c['R'], c['sk'])))
    assert eq(M_dec, M), f"{label} decrypts to wrong M"
print("\nAll three decrypt to the same M. Credentials are unlinkable")
print("but bound to the same real identity.")
```



```
Three credentials derived from the same identity:

  source:
    pk    = (871028..., 739125...)
    E.R   = (083245..., 237878...)
    E.C   = (626218..., 727464...)
```

```
    sig_1 = (533198..., 625443...)

  burn:
    pk    = (108640..., 898592...)
    E.R   = (061399..., 140540...)
    E.C   = (153173..., 396067...)
    sig_1 = (971031..., 377389...)

  dest:
    pk    = (641982..., 616302...)
    E.R   = (172262..., 561732...)
    E.C   = (974575..., 124345...)
    sig_1 = (205108..., 302236...)

All three decrypt to the same M. Credentials are unlinkable
but bound to the same real identity.
```

How to read the output: Every coordinate fragment (`pk`, `E.R`, `E.C`, `sig_1`) must be completely different across the three credentials. If any pair matched, the unlinkability property would be broken. Despite being independent, all three encrypt the same $M = m \cdot G$ – confirmed by the decryption check.

6.5 Chaum-Pedersen: Wormhole Identity Binding

The wormhole SNARK internally verifies a Chaum-Pedersen proof that the source and destination ElGamal ciphertexts encrypt the same identity point M. This is the same proof used in `approve` (see Identity Examples), but here it runs inside a SNARK rather than directly on-chain, hiding *which* addresses are involved.


```
src = creds["source"]
dst = creds["dest"]

# Alice proves E_source and E_dest encrypt the same M
# She knows sk_source (decrypts E_source) and r_dest (constructs E_dest)
k1 = rand_scalar(); k2 = rand_scalar()
T1 = multiply(G1, k1)
T2 = multiply(G1, k2)
T3 = add(multiply(dst['pk'], k2), neg(multiply(src['R'], k1)))

# Fiat-Shamir challenge
e_input = "".join(str(v) for v in [
    G1, src['pk'], dst['pk'], src['R'], src['C'],
    dst['R'], dst['C'], T1, T2, T3])
e = int(hashlib.sha256(e_input.encode()).hexdigest(), 16) % ORDER

s1 = (k1 - e * src['sk']) % ORDER
s2 = (k2 - e * dst['r']) % ORDER

# Verification (what the SNARK circuit checks internally)
v1 = add(multiply(G1, s1), multiply(src['pk'], e))
v2 = add(multiply(G1, s2), multiply(dst['R'], e))
C_ratio = add(dst['C'], neg(src['C']))
v3 = add(add(multiply(dst['pk'], s2),
             neg(multiply(src['R'], s1))),
         multiply(C_ratio, e))

print("Chaum-Pedersen wormhole identity binding:")
print(f"  Check 1 (owns source key):     {eq(v1, T1)}  PASS")
print(f"  Check 2 (knows dest randomness): {eq(v2, T2)}  PASS")
print(f"  Check 3 (same M):              {eq(v3, T3)}  PASS")
print(f"\nThis proof runs INSIDE the SNARK.  The EVM verifier sees")
print(f"only the SNARK proof -- not which addresses or identities")
print(f"are involved.")
```

```
Chaum-Pedersen wormhole identity binding:
  Check 1 (owns source key):     True  PASS
  Check 2 (knows dest randomness): True  PASS
  Check 3 (same M):              True  PASS

This proof runs INSIDE the SNARK.  The EVM verifier sees
only the SNARK proof -- not which addresses or identities
are involved.
```

6.6 Counter-Example: Different Identity at Destination

If Alice tries to mint BUCKs at an address with a *different* identity, the Chaum-Pedersen proof fails at Check 3 – the SNARK cannot be generated.

```
# Eve tries to wormhole BUCKs to an address with a different identity
m_fake = rand_scalar()
r_fake = rand_scalar()
R_fake = multiply(G1, r_fake)
C_fake = add(multiply(G1, m_fake), multiply(dst['pk'], r_fake))

k1f = rand_scalar(); k2f = rand_scalar()
T1f = multiply(G1, k1f)
T2f = multiply(G1, k2f)
T3f = add(multiply(dst['pk'], k2f), neg(multiply(src['R'], k1f)))

e_input_f = "".join(str(v) for v in [
    G1, src['pk'], dst['pk'], src['R'], src['C'],
    R_fake, C_fake, T1f, T2f, T3f])
ef = int(hashlib.sha256(e_input_f.encode()).hexdigest(), 16) % ORDER

s1f = (k1f - ef * src['sk']) % ORDER
s2f = (k2f - ef * r_fake) % ORDER

v1f = add(multiply(G1, s1f), multiply(src['pk'], ef))
v2f = add(multiply(G1, s2f), multiply(R_fake, ef))
C_ratio_f = add(C_fake, neg(src['C']))
v3f = add(add(multiply(dst['pk'], s2f),
              neg(multiply(src['R'], s1f))),
          multiply(C_ratio_f, ef))

print("Counter-example: different identity at destination")
print(f"  Check 1 (owns source key):     {eq(v1f, T1f)}  PASS")
print(f"  Check 2 (knows dest randomness): {eq(v2f, T2f)}  PASS")
print(f"  Check 3 (same M):              {eq(v3f, T3f)}  FAIL")
print(f"\nChecks 1-2 pass: Alice owns her key and knows the randomness.")
print(f"Check 3 fails: source encrypts m, destination encrypts m'.")
print(f"The SNARK proof cannot be generated -- identity laundering")
print(f"through the wormhole is impossible.")



Counter-example: different identity at destination
  Check 1 (owns source key):     True  PASS
  Check 2 (knows dest randomness): True  PASS
  Check 3 (same M):              False  FAIL

Checks 1-2 pass: Alice owns her key and knows the randomness.
Check 3 fails: source encrypts m, destination encrypts m'.
The SNARK proof cannot be generated -- identity laundering
through the wormhole is impossible.
```

6.7 Merkle Tree for Note Commitments

The Phase 2 privacy pool stores note commitments in a Merkle tree. Each note is a Pedersen commitment $C = v \cdot G + r \cdot H$, where v is the BUCK value and r is a random blinding factor.

The tree root is stored on-chain; the SNARK proves membership without revealing which note was spent.

```
def merkle_hash(a, b):
    return hashlib.sha256(a + b).digest()

def note_commit(value, blinding):
    return add(multiply(G1, value), multiply(H, blinding))

def note_hash(commitment):
    return hashlib.sha256(str(commitment).encode()).digest()

# Four users deposit notes into the pool
deposits = [
    (1000, "Alice"), (500, "Bob"), (2000, "Carol"), (750, "Dave")]

notes = []
for val, owner in deposits:
    r = rand_scalar()
    C = note_commit(val, r)
    notes.append(dict(value=val, owner=owner, r=r, C=C, h=note_hash(C)))

# Build Merkle tree (4 leaves)
L0 = [n['h'] for n in notes]
L1 = [merkle_hash(L0[0], L0[1]), merkle_hash(L0[2], L0[3])]
root = merkle_hash(L1[0], L1[1])

print("Note commitment Merkle tree:")
for n in notes:
    print(f"  {n['owner']:5s}: {n['value']:5d} BUCK  leaf={n['h'].hex()[:16]}...")
print(f"  Root: {root.hex()[:16]}...")

# Merkle proof for Alice's note (index 0)
proof_path = [L0[1], L1[1]]  # sibling at level 0, sibling at level 1
verify = merkle_hash(merkle_hash(notes[0]['h'], proof_path[0]), proof_path[1])
print(f"\n  Alice's membership proof verifies: {verify == root}")
print(f"  Proof size: {len(proof_path)} hashes ({len(proof_path) * 32} bytes)")
print(f"  In production (2^20 tree): 20 hashes (640 bytes)")
```



```
Note commitment Merkle tree:
  Alice:  1000 BUCK  leaf=6cefb0e714d8b7f7...
  Bob  :   500 BUCK  leaf=8e7d2f5be3a5cfc2...
  Carol:  2000 BUCK  leaf=76f145cb3e7a760d...
  Dave :   750 BUCK  leaf=c8058a0d94b6e429...
  Root: e462d912c3107692...

  Alice's membership proof verifies: True
  Proof size: 2 hashes (64 bytes)
  In production (2^20 tree): 20 hashes (640 bytes)
```

How to read the output: The membership proof must verify (`True`). The proof consists of sibling hashes along the path from the leaf to the root. Inside the SNARK, this proof demonstrates that Alice owns a note in the tree without revealing *which* note.

6.8 Pedersen Commitment Splitting

Pedersen commitments are additively homomorphic: splitting a note into two sub-notes preserves the total value, and anyone can verify the conservation equation without knowing the individual values.

```
# Alice splits her 1000 BUCK note into 400 + 600
v_orig = 1000
```

```
r_orig = notes[0]['r']
C_orig = notes[0]['C']

v1 = 400;  r1 = rand_scalar()
v2 = 600;  r2 = (r_orig - r1) % ORDER   # ensures blinding factors sum

C1 = note_commit(v1, r1)
C2 = note_commit(v2, r2)
C_sum = add(C1, C2)

print("Note splitting (Pedersen homomorphism):")
print(f"  Original:  {v_orig} BUCK  {pt(C_orig, 'C=')}")
print(f"  Split 1:   {v1:4d} BUCK  {pt(C1, 'C1=')}")
print(f"  Split 2:   {v2:4d} BUCK  {pt(C2, 'C2=')}")
print(f"  C1 + C2 == C_orig: {eq(C_sum, C_orig)}")
print(f"\n  The SNARK proves:")
print(f"    1. C_orig is in the Merkle tree (membership)")
print(f"    2. C1 + C2 = C_orig (value conservation)")
print(f"    3. v1 > 0 and v2 > 0 (range proofs)")
print(f"    4. Nullifier for C_orig is correctly derived")
print(f"  Without revealing v1, v2, or which note was split.")

# Verify that a dishonest split fails
# Eve tries to create 500 + 600 = 1100 (more than the original)
C_cheat1 = note_commit(500, r1)
C_cheat2 = note_commit(600, r2)
C_cheat_sum = add(C_cheat1, C_cheat2)
print(f"\n  Counter-example: 500 + 600 = 1100 > 1000")
print(f"  C_cheat1 + C_cheat2 == C_orig: {eq(C_cheat_sum, C_orig)}")
print(f"  Conservation check fails -- SNARK cannot be generated.")
```



```
Note splitting (Pedersen homomorphism):
  Original:  1000 BUCK  C=(046891..., 133236...)
  Split 1:    400 BUCK  C1=(068557..., 571725...)
  Split 2:    600 BUCK  C2=(212505..., 613904...)
  C1 + C2 == C_orig: True

  The SNARK proves:
    1. C_orig is in the Merkle tree (membership)
    2. C1 + C2 = C_orig (value conservation)
    3. v1 > 0 and v2 > 0 (range proofs)
    4. Nullifier for C_orig is correctly derived
  Without revealing v1, v2, or which note was split.

  Counter-example: 500 + 600 = 1100 > 1000
  C_cheat1 + C_cheat2 == C_orig: False
  Conservation check fails -- SNARK cannot be generated.
```

6.9 Demurrage Inside the Pool

Notes record their deposit timestamp. At withdrawal, the claimable amount is reduced by demurrage owed.

```
# Simulate demurrage calculation at withdrawal time
deposit_value = 1000
rate = 0.02  # 2% per year, linear

print("Demurrage at withdrawal (2%/yr linear):")
print(f"  {'Years idle':>12s}  {'Demurrage':>10s}  {'Claimable':>10s}  {'Expired?':>8s}")
print(f"  {'-'*12}  {'-'*10}  {'-'*10}  {'-'*8}")
for years in [0, 1, 5, 10, 25, 49, 50, 51]:
    dem = min(deposit_value, int(deposit_value * rate * years))
    claim = deposit_value - dem
    expired = "YES" if claim <= 0 else ""
    print(f"  {years:12d}  {dem:10d}  {claim:10d}  {expired:>8s}")
```

```
print(f"\n  After 50 years, the note is worthless.")
print(f"  The SNARK range proof rejects withdrawal of expired notes.")
print(f"  This prevents the pool from being a demurrage shelter.")
```

```
Demurrage at withdrawal (2%/yr linear):
    Years idle   Demurrage   Claimable  Expired?
  ------------  ----------  ----------  --------
  0           0        1000
  1          20         980
  5         100         900
        10          200         800
        25          500         500
        49          980          20
        50         1000           0       YES
        51         1000           0       YES

  After 50 years, the note is worthless.
  The SNARK range proof rejects withdrawal of expired notes.
  This prevents the pool from being a demurrage shelter.
```

6.10 Schnorr Authorization (`approveFor`)

A Schnorr proof of knowledge of sk authorizes the caller to act on behalf of any address whose registered identity key is $pk = sk \cdot G$. This is how Alice completes the bilateral identity exchange for the burn address (which has no ECDSA key), and how Identity Guardians authorize recovery and cosigning.


```
# Alice proves she knows sk_burn (the alt_bn128 identity key for addr_burn)
# without holding addr_burn's ECDSA key.

# --- Prover (Alice) ---
sk_burn = creds["burn"]['sk']   # burn address credential from earlier
pk_burn = creds["burn"]['pk']
caller = "0xAliceEthAddr"  # msg.sender (Alice's real Ethereum address)
principal = addr_burn      # the address being represented

# Commit
k = rand_scalar()
T = multiply(G1, k)

# Challenge (Fiat-Shamir: binds pk, T, both addresses)
e_input = "".join(str(v) for v in [pk_burn, T, caller, principal])
e = int(hashlib.sha256(e_input.encode()).hexdigest(), 16) % ORDER

# Respond
s = (k - e * sk_burn) % ORDER

print("Schnorr Proof of Knowledge of sk_burn")
print(f"  pk_burn:   {pt(pk_burn)}")
print(f"  principal: {principal}")
print(f"  caller:    {caller}")
print(f"  T:         {pt(T)}")
print(f"  e:         {e}")
print(f"  s:         {s}")

# --- Verifier (on-chain) ---
# Reconstruct: s*G + e*pk should equal T
lhs = add(multiply(G1, s), multiply(pk_burn, e))
print(f"\n  Verify: s*G + e*pk == T?  {eq(lhs, T)}")
print(f"  Authorization granted: caller may act for principal")
```

```
Schnorr Proof of Knowledge of sk_burn
  pk_burn:   (108640..., 898592...)
  principal: 0x0b8bfe1dfd0b6d9fa2083e87a70a8847eb6effd8
  caller:    0xAliceEthAddr
  T:         (869215..., 442407...)
  e:         21404118366995222845980049305572020641566321417502915686985751394779291175052
  s:         51953862004253915300988648204607563123336117903493801846468757199897801354 79

  Verify: s*G + e*pk == T?  True
  Authorization granted: caller may act for principal
```

Counter-example: Eve tries to authorize herself for Alice's burn address without knowing `sk_burn`.

```
# Eve doesn't know sk_burn, tries with a random key
sk_eve = rand_scalar()
caller_eve = "0xEveEthAddr"

k_e = rand_scalar()
T_e = multiply(G1, k_e)

e_input_e = "".join(str(v) for v in [pk_burn, T_e, caller_eve, principal])
e_e = int(hashlib.sha256(e_input_e.encode()).hexdigest(), 16) % ORDER

# Eve responds with her own sk instead of sk_burn
s_e = (k_e - e_e * sk_eve) % ORDER

lhs_e = add(multiply(G1, s_e), multiply(pk_burn, e_e))
result = eq(lhs_e, T_e)

print("Counter-example: Eve uses wrong secret key")
print(f"  Eve's sk:  {sk_eve}")
print(f"  pk_burn:   {pt(pk_burn)}  (unchanged)")
print(f"  T:         {pt(T_e)}")
print(f"  Verify:    {result}  <-- FAIL")
print(f"\n  s*G + e*pk != T because s encodes sk_eve, not sk_burn.")
print(f"  Eve cannot call approveFor on Alice's burn address.")
```



```
Counter-example: Eve uses wrong secret key
  Eve's sk:  19409561467423119366097563770073560070103176100793125658591438891249264388959
  pk_burn:   (108640..., 898592...)  (unchanged)
  T:         (268645..., 699719...)
  Verify:    False  <-- FAIL

  s*G + e*pk != T because s encodes sk_eve, not sk_burn.
  Eve cannot call approveFor on Alice's burn address.
```

6.11 Guardian Scenario: Account Recovery

After 90 days of inactivity, Bob (a designated `RECOVERY` guardian) can redirect Alice's funds. Bob proves knowledge of *his own* identity key, and the contract checks that Bob is registered as a recovery guardian for Alice.

```
# Bob is a RECOVERY guardian for Alice's account
sk_bob = rand_scalar()
pk_bob = multiply(G1, sk_bob)
alice_account = "0xAliceAccount"
caller_bob = "0xBobEthAddr"

# Bob generates a Schnorr proof of his own identity
k_b = rand_scalar()
T_b = multiply(G1, k_b)
```

```
e_input_b = "".join(str(v) for v in [pk_bob, T_b, caller_bob, alice_account])
e_b = int(hashlib.sha256(e_input_b.encode()).hexdigest(), 16) % ORDER
s_b = (k_b - e_b * sk_bob) % ORDER

lhs_b = add(multiply(G1, s_b), multiply(pk_bob, e_b))

print("Guardian Recovery: Bob recovers Alice's account")
print(f"  Bob's pk:     {pt(pk_bob)}")
print(f"  Alice's addr: {alice_account}")
print(f"  Schnorr ok:   {eq(lhs_b, T_b)}")
print()
print("  On-chain checks (pseudocode):")
print("    1. verifySchnorr(proof, pk_bob, msg.sender, alice_account) ")
print("    2. guardians[alice_account] contains pk_bob as RECOVERY   ")
print("    3. lastActivity[alice_account] + 90 days < block.timestamp ")
print("    4. → recoverAccount(alice_account, new_addr) executes")
print()
print("  If Alice is still active (check 3 fails), Bob cannot recover.")
print("  If Bob is not a registered guardian (check 2 fails), rejected.")
print("  The Schnorr proof prevents impersonation of Bob's identity.")
```

```
Guardian Recovery: Bob recovers Alice's account
  Bob's pk:     (658644..., 872299...)
  Alice's addr: 0xAliceAccount
  Schnorr ok:   True

  On-chain checks (pseudocode):
    1. verifySchnorr(proof, pk_bob, msg.sender, alice_account)
    2. guardians[alice_account] contains pk_bob as RECOVERY
    3. lastActivity[alice_account] + 90 days < block.timestamp
    4. → recoverAccount(alice_account, new_addr) executes

  If Alice is still active (check 3 fails), Bob cannot recover.
  If Bob is not a registered guardian (check 2 fails), rejected.
  The Schnorr proof prevents impersonation of Bob's identity.
```


7 Attacks and Defenses

Each attack considers what an adversary can attempt and why the protocol prevents it.

7.1 Attack 1: Amount and Age Correlation

Scenario: Eve observes a 1,000 BUCK burn from an account with BUCK-age 10,000 and looks for a mint with matching amount and age.

Phase 1 defense: Fixed denominations eliminate amount correlation. BUCK-age freedom eliminates age correlation. The destination BUCK-age is freely chosen (not carried from the source), so a 1,000 BUCK mint appearing "brand new" could have come from any source regardless of that source's age. If 50 users each wormhole 1,000 BUCK in a given week, Eve sees 50 burns and 50 mints of identical amounts with *unrelated* ages. The anonymity set is all users who transacted the same denomination in a comparable time window.

Phase 2 defense: The note pool allows arbitrary amounts and splitting. Alice deposits 1,000, splits into 400 + 600, and withdraws at different times to different addresses. With many users interleaving operations, individual deposit-withdrawal links dissolve in the shared Merkle tree.

7.2 Attack 2: Timing Correlation

Scenario: Alice burns at block N and mints at block $N + 5$. Eve correlates by proximity.

Defense: The protocol does not enforce timing constraints, but wallet software should introduce random delays (hours to days). Phase 2 further weakens timing correlation: splits can occur at any time, and withdrawals are decoupled from deposits by the shared pool.

Residual risk: In early adoption (few wormhole users), timing correlation remains a statistical threat. The anonymity set grows with adoption.

7.3 Attack 3: Double-Mint (Nullifier Reuse)

Scenario: Alice tries to mint twice from the same burn.

Defense: Each burn produces a unique nullifier $N = \text{sha256}(\texttt{MAGIC\_NULL} \| \text{secret})$. The contract records used nullifiers in a mapping. The SNARK proves the nullifier is correctly derived from the secret. Reuse is rejected on-chain.

Soundness: The nullifier is deterministic given the secret. Alice cannot produce a second valid nullifier for the same burn without finding a SHA-256 collision.

7.4 Attack 4: Supply Inflation

Scenario: An attacker exploits the wormhole to create BUCKs from nothing, inflating the supply.

Defense: The SNARK proves a Merkle path in Ethereum's state trie, verifying that the burn address holds the claimed balance. The state root is a public input validated against recent block hashes. Forging a Merkle proof requires breaking SHA-256 or the SNARK's soundness.

`_mint_noincrease` does not touch `totalSupply`, so even a successful wormhole does not inflate the economic supply metric. The only inflation is in `sum(balances)`, which is bounded by the total amount wormholed and auditable from event logs.

Critical risk: A bug in the SNARK circuit (e.g., failing to verify the Merkle proof) could enable minting without a corresponding burn. This is the highest-severity risk in any wormhole system, shared by EIP-7503. Mitigation: formal verification of the circuit, rate limiting, and a pause mechanism for the `wormholeTransfer` function.

7.5 Attack 5: Identity Laundering

Scenario: Criminal Alice deposits dirty BUCKs and withdraws clean BUCKs to an address with a different identity.

Defense: The SNARK internally verifies a Chaum-Pedersen proof that the deposit and withdrawal credentials share the same M. The same real person enters and exits. After withdrawal, the destination address has a valid identity registration. If regulators subpoena Alice at the destination address, the standard ElGamal proof chain recovers her identity.

This is the key differentiator from Tornado Cash and Zcash, which provide anonymity without identity continuity.

7.6 Attack 6: Demurrage Avoidance

Scenario: Alice deposits BUCKs into the privacy pool to avoid demurrage, then withdraws after years without paying.

Phase 1 defense: The burn address's balance sits idle forever. Alice loses the full burn amount permanently – demurrage avoidance is moot because the BUCKs are already lost.

Phase 2 defense: Notes record deposit timestamps. The SNARK computes demurrage at withdrawal time and reduces the claimable amount. A note deposited 10 years ago loses 20% of its value. After 50 years, the note is expired and worthless.

7.7 Attack 7: Anonymity Set Degradation

Scenario: Few users adopt the wormhole, making each burn-mint pair trivially linkable.

Defense: This is an adoption problem, not a cryptographic one. The protocol is sound regardless of set size; privacy improves with usage. Mitigations:

- Wallet integration (make wormholing a default option for large transfers)
- Fixed denominations concentrate users into shared pools
- Phase 2's shared Merkle tree accumulates notes from all users, growing the anonymity set over time

7.8 Attack 8: Burn Address Discovery

Scenario: Eve identifies burn addresses by observing addresses that receive transfers but never send.

Defense: Many legitimate accounts are idle (savings, inheritance, lost keys). Burn addresses are indistinguishable from these. They have valid identity registrations, they participated in normal `approve` + `transfer` flows, and their on-chain footprint is identical to any dormant wallet.

Residual risk: Statistical analysis over long time periods might identify addresses that are *always* idle as probable burn addresses. This reveals that *some* wormholing occurred, but not who did it or where the funds went.

7.9 Attack 9: SNARK Circuit Soundness

Scenario: A flaw in the SNARK circuit allows forged proofs.

Defense: This is the existential risk for any ZK-based system. Mitigations:

- Formal verification of the circuit (e.g., using Circom's constraint checking or Lean proofs)
- Trusted setup ceremony (if using Groth16) or transparent setup (if using PLONK/STARK)
- Bug bounties and audit programs
- Rate limiting on `wormholeTransfer` (max N per block/epoch)
- Governance pause mechanism for emergencies
- Incremental deployment: start with Phase 1 (simpler circuit), graduate to Phase 2 after battle-testing

7.10 Attack 10: Malicious Recovery Guardian

Scenario: Bob is designated as Alice's `RECOVERY` guardian. Bob waits for Alice to become temporarily inactive (vacation, illness) and triggers recovery to steal her funds.

Defense: The `activationParam` sets a minimum inactivity period (e.g., 90 days). Any transaction from Alice resets the timer. Mitigations:

- Alice can set a long activation delay (180+ days)
- Alice can revoke or replace her recovery guardian at any time

- The recovery operation emits an event, giving Alice notice to intervene if she returns during the activation window
- Multi-guardian configurations: require 2-of-3 guardians to agree

Residual risk: If Alice is truly incapacitated for the full activation period and her guardian is malicious, funds are at risk. This is the same tradeoff as any recovery system (social recovery wallets, inheritance schemes).

7.11 Attack 11: Compromised Cosigner

Scenario: Alice designates her spouse as a `COSIGNER` for transfers above 10,000 BUCK. The spouse's identity key is compromised (phishing, malware), and an attacker uses it to co-approve unauthorized large transfers.

Defense: The cosigner requirement is additive – an attacker needs *both* Alice's ECDSA key (to initiate the transfer) *and* the cosigner's identity key (to co-approve). Compromising one is insufficient.

Mitigations:

- Key rotation: the cosigner can re-derive a fresh credential (same M, new sk) and update their guardian registration
- Alert on co-approval events
- The cosigner's Schnorr proof binds `msg.sender`, so the attacker must also control an Ethereum address (cannot replay from off-chain)

Residual risk: If both Alice's ECDSA key and the cosigner's alt_bn128 key are compromised simultaneously, the two-factor protection is defeated. This is the fundamental limit of any multi-factor scheme.

7.12 Defense Summary

Attack	Primary defense	Residual risk
Amount + age correlation	Fixed denom + free BUCK-age	Low with adoption
Timing correlation	Random delays / pool mixing	Moderate early on
Double-mint	Nullifier uniqueness (SHA-256)	None (deterministic)
Supply inflation	Merkle proof + `_mint_noincrease`	SNARK soundness
Identity laundering	Chaum-Pedersen in SNARK	None (math)
Demurrage avoidance	In-pool demurrage computation	None (enforced in ZK)
Anonymity set	Shared pool + adoption	Early adoption risk
Burn address discovery	Identical to dormant wallets	Statistical over time
SNARK soundness	Formal verification + audits	Residual (any ZK)
Malicious guardian	Activation delay + revocation	Extended incapacity
Compromised cosigner	Two-factor (ECDSA + alt_bn128)	Dual key compromise

8 Comparison with Existing Privacy Mechanisms

8.1 EIP-7503: Zero-Knowledge Wormholes

EIP-7503[1] proposed the burn-and-mint wormhole concept for ETH. BUCK wormholes adapt the core mechanism but diverge in four ways:

1. **Identity binding**: EIP-7503 provides pure anonymity – anyone can burn and mint. BUCK wormholes prove identity continuity via Chaum-Pedersen inside the SNARK. The same identified person enters and exits.

2. **Supply accounting**: EIP-7503 burns and re-mints ETH (supply neutral at the protocol level). BUCK uses `_mint_noincrease`, leaving `totalSupply` unchanged while the burn balance becomes phantom. This avoids interaction with BUCK-specific economics (demurrage, Jubilee, PID controller).

3. **Identity infrastructure**: BUCK burn addresses carry full identity registrations (PS signature, ElGamal ciphertext, NIZK proof), making them indistinguishable from normal accounts. EIP-7503 burn addresses are bare EOAs with no identity layer.

4. **Regulatory model**: EIP-7503 offers optional "privacy pools" compliance via association sets. BUCK provides cryptographic identity continuity – strictly stronger than social association.

8.2 Tornado Cash

Tornado Cash[2] uses fixed-denomination deposit pools with nullifier-based withdrawal. BUCK's Phase 1 follows the same pattern, with two additions:

- Identity-verified deposits and withdrawals (Chaum-Pedersen binding)
- Demurrage-aware notes (preventing the pool from being an economic shelter)

Tornado Cash was sanctioned by OFAC in 2022 partly because it provided pure anonymity with no identity accountability. BUCK wormholes address this by ensuring identity continuity through the privacy layer.

8.3 Zcash (Sapling/Orchard)

Zcash[3] pioneered note-based shielded transactions using ZK-SNARKs. BUCK's Phase 2 pool architecture draws directly from Zcash's note commitment / nullifier / Merkle tree design.

Key differences:

- Zcash is a standalone blockchain; BUCK operates on Ethereum (gas costs, EVM constraints, shared state trie)
- Zcash has no identity layer; BUCK binds every note to an issuer-signed identity
- Zcash has no demurrage; BUCK notes age and lose value over time
- Zcash's anonymity set is the entire shielded pool; BUCK's is the wormhole pool (smaller, but growing with adoption)

8.4 Aztec Protocol

Aztec[4] brings Zcash-like shielded transactions to Ethereum as an L2 rollup with PLONK-based proofs. BUCK wormholes operate at the L1 application layer (inside the BUCK contract) rather than at the L2 infrastructure layer.

The tradeoffs:

[2]Tornado Cash. Pertsev, A. et al., 2019. Fixed-denomination privacy pool on Ethereum using zkSNARKs and a note commitment / nullifier scheme. Sanctioned by OFAC in August 2022.

[3]Zcash Protocol Specification (Sapling). Hopwood, D. et al., 2018. Shielded transactions using Groth16 proofs over the BLS12-381 curve with a note commitment / nullifier / Merkle tree architecture. `https://zips.z.cash/protocol/protocol.pdf`

[4]Aztec Protocol. Williamson, Z. et al., 2019. Ethereum L2 rollup providing general-purpose private transactions using PLONK proofs. `https://aztec.network/`

- Aztec provides general-purpose privacy for any token; BUCK wormholes are BUCK-specific but deeply integrated with BUCK's economics
- Aztec uses universal PLONK setup; BUCK's SNARK could use the same or a simpler scheme
- Aztec has no demurrage-aware notes or identity binding

8.5 Railgun

Railgun provides shielded ERC-20 transfers on Ethereum L1 using SNARKs. Like BUCK wormholes, it operates at the smart contract level.

Differences:

- Railgun is token-agnostic; BUCK wormholes are BUCK-specific with demurrage integration
- Railgun has no identity binding (pure privacy)
- BUCK's identity layer enables regulatory compliance without compromising privacy for legitimate users

8.6 Comparison Summary

Feature	BUCK Wormhole	EIP-7503	Tornado Cash	Zcash	Aztec	Railgun
Identity binding	**yes** (CP+PS)	no	no	no	no	no
Demurrage-aware	**yes**	n/a	no	no	no	no
Supply accounting	`_mint_noincrease`	burn+mint	deposit/withdraw	UTXO	rollup	deposit/withdraw
Amount privacy	Phase 2	no	fixed denom	full	full	full
Anonymity set	pool users	all EOAs	pool users	shielded pool	rollup users	pool users
Regulatory compliance	**identity**	association	none	none	none	none
On-chain cost	~200K gas	~200K gas	~1M gas	native	rollup	~500K gas
Trusted setup	depends	yes	yes	yes (Powers)	no (PLONK)	yes
L1/L2	L1	L1	L1	L1 (own)	L2	L1

BUCK wormholes are the only mechanism combining transaction graph privacy with cryptographic identity continuity. Every other system provides either full anonymity (regulatory risk) or no privacy (identity exposure). The wormhole occupies the middle ground: privacy for users, accountability for regulators.

9 Implementation Considerations

9.1 Gas Costs

Operation	Estimated gas	Frequency
Burn (approve+transfer)	~120,000	Once per wormhole
`approveFor` (Schnorr)	~41,000	Once per wormhole
SNARK verification	~200,000	Once per wormhole
`wormholeTransfer` total	~250,000	Once per wormhole
Identity registration	~235,000	Once per address
Schnorr authorization	~12,000	Per guardian op
Pool deposit	~150,000	Once per deposit
Pool withdraw	~250,000	Once per withdraw
Note split	~200,000	Per split

The SNARK verification cost depends on the proof system:

- Groth16: ~200K gas (cheapest, requires trusted setup)
- PLONK: ~300K gas (no trusted setup)
- STARK: ~500K+ gas (largest proofs, fully transparent)

9.2 SNARK Circuit Complexity

The wormhole SNARK circuit includes:

- SHA-256 hash verification (burn address and nullifier)
- Ethereum state trie Merkle proof (MPT, ~20 levels)
- Chaum-Pedersen verification (4 EC scalar multiplications)
- PS pairing check (3-4 pairings – expensive inside a SNARK)

The PS pairing check is the most expensive component. Alternatives:

- Replace the PS check inside the SNARK with a hash-based commitment (prove knowledge of m such that $H(m)$ matches a committed value, without the pairing)
- Use a SNARK-friendly signature scheme (e.g., EdDSA) for the identity binding, with PS verification done separately on-chain

9.3 Phased Deployment

Phase 1 (near-term):

- Fixed denominations: 100, 1,000, 10,000 BUCK
- SNARK circuit: burn address + Merkle proof + Chaum-Pedersen (no pairings inside the SNARK – PS verification happens at registration time, before the wormhole)
- `wormholeTransfer` contract function
- Wallet UI for wormhole operations

Phase 2 (future):

- Note-based privacy pool contract
- Pedersen commitment tree
- Split/merge SNARK circuit
- Demurrage-aware notes
- Range proofs (Bulletproofs or SNARK-native)

10 Summary

BUCK wormholes provide transaction graph privacy while preserving the identity guarantees that define Alberta Buck. The mechanism adapts EIP-7503's burn-and-mint wormhole for BUCK's demurrage economics (`_mint_noincrease`, no supply inflation, Jubilee-safe) and extends it with Chaum-Pedersen identity binding inside a SNARK (same person in, same person out, provable without revealing who).

The wormhole also introduces *Identity Guardians* – a general framework for identity-based authorization that decouples ECDSA key possession from the right to act on an account. The same Schnorr proof mechanism that lets Alice manage burn addresses also enables cold wallet management, account recovery after key loss, and cosigned transfers requiring spousal or partner approval above a threshold.

Phase 1 (fixed-denomination wormholes) requires modest SNARK engineering and can be deployed incrementally. Phase 2 (note-based privacy pool) enables full amount privacy via Pedersen commitment splitting, drawing on proven designs from Zcash and Tornado Cash but adding identity continuity and demurrage awareness.

The result is a privacy layer that occupies a unique position: stronger privacy than ERC-20 (transaction graph is hidden), stronger accountability than Tornado Cash or Zcash (identity continuity is cryptographically enforced), and compatible with BUCK's distinctive economic model (demurrage, Jubilee, PID stabilization).