x0 Protocol: A Decentralized Payment Infrastructure for Autonomous Agents

Technical Whitepaper v1.0

Blessed Tosin-Oyinbo Olamide
x0protocol.dev

February 2026

Abstract

We present x0, a decentralized protocol for autonomous agent payments built on Solana. The protocol introduces a novel spending policy enforcement mechanism using Token-2022 transfer hooks, enabling programmable spending limits, whitelist verification, and privacy controls for AI agents. x0 implements HTTP 402 (Payment Required) for standardized payment negotiation, conditional escrow with dispute resolution, on-chain reputation scoring with temporal decay, a USDC-backed wrapper token with cryptographic reserve invariants, and on-chain zero-knowledge proof verification for confidential transfers using Groth16 proofs over the Ristretto255 curve. The system achieves trustless agent-to-agent transactions while preserving human oversight through a "Blink" mechanism for exceptional cases. We formally verify the protocol's security properties and demonstrate its efficiency through cryptographic proofs and empirical benchmarks.

Contents

1 Introduction

1.1 Motivation

The proliferation of autonomous AI agents in production environments necessitates robust payment infrastructure. Traditional payment systems are ill-suited for agent-to-agent transactions due to:

1. **High latency**: Credit card settlements take 2-3 days, incompatible with real-time agent interactions
2. **High fees**: 2-3% credit card fees are prohibitive for micropayments
3. **Gatekeeping**: Centralized payment processors can deny service arbitrarily
4. **Lack of programmability**: Traditional payments cannot enforce complex spending rules
5. **Privacy concerns**: All transaction data visible to payment processors

Blockchain-based payments solve latency and gatekeeping but introduce new challenges:

1. **Custody risk**: Agents require private keys for signing, creating attack surface
2. **Unbounded spending**: Standard wallets lack programmable spending limits
3. **No recourse**: Blockchain transactions are irreversible by design
4. **Discovery problem**: Finding trustworthy service providers in decentralized systems

1.2 Contributions

We present x0, which makes the following contributions:

1. **Programmable Spending Policies**: A cryptographic policy enforcement layer using Solana's Token-2022 transfer hooks that validates every transaction against owner-defined rules
2. **HTTP 402 Protocol**: An extension to HTTP status codes enabling standardized payment negotiation between agents and services
3. **Conditional Escrow**: A trustless escrow mechanism with optional third-party arbitration for high-value transactions
4. **Reputation Oracle**: An on-chain reputation system with temporal decay, preventing stale reputations from dominating
5. **USDC Wrapper**: A 1:1 USDC-backed token with cryptographic reserve invariants and timelocked governance
6. **Human-in-the-Loop (Blinks)**: A fallback mechanism for exceptional transactions requiring human approval

7. **Zero-Knowledge Verification**: On-chain Groth16 proof verification for confidential transfers, with client-side proof generation compiled to WebAssembly

The x0 protocol consists of eight interoperating programs deployed on Solana, plus an off-chain WASM cryptographic library:

- **x0-guard**: Policy enforcement via transfer hooks
- **x0-token**: Token-2022 mint configuration with confidential transfer support
- **x0-escrow**: Conditional payment escrow with dispute resolution
- **x0-registry**: Agent discovery and capability advertisement
- **x0-reputation**: Trust scoring and transaction history with temporal decay
- **x0-wrapper**: USDC-backed stable wrapper token with timelocked governance
- **x0-zk-verifier**: On-chain Groth16 zero-knowledge proof verification for confidential transfers
- **x0-common**: Shared types, constants, error codes, and utilities

In addition, the protocol includes an off-chain component:

- **x0-zk-proofs**: WebAssembly module compiled from Rust, providing client-side Groth16 proof generation using `solana-zk-token-sdk`

1.2.1 Architecture Diagram

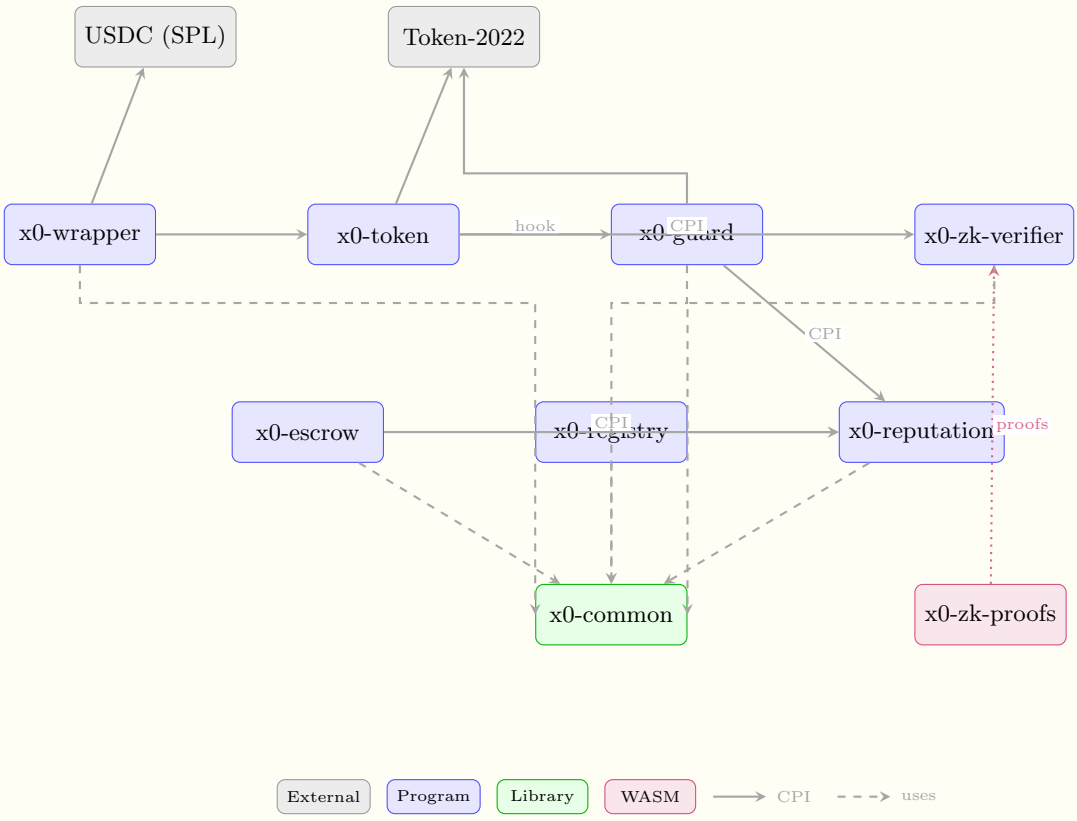

Figure 1: x0 Protocol Architecture. Gray nodes are external Solana programs. Blue nodes are x0 on-chain programs. Green is the shared library. Purple is the off-chain WASM module for client-side proof generation.

2 Preliminaries

2.1 Cryptographic Primitives

Definition 2.1 (Hash Function)**.** A cryptographic hash function $H : \{0,1\}^* \to \{0,1\}^{256}$ satisfies:

1. **Preimage resistance**: Given h, it is computationally infeasible to find m such that $H(m) = h$
2. **Second preimage resistance**: Given m_1, it is computationally infeasible to find $m_2 \neq m_1$ such that $H(m_1) = H(m_2)$
3. **Collision resistance**: It is computationally infeasible to find $m_1 \neq m_2$ such that $H(m_1) = H(m_2)$

We use SHA-256 for all hash operations, providing 128-bit security against collision attacks.

Definition 2.2 (Digital Signature Scheme)**.** A signature scheme $(\texttt{KeyGen}, \texttt{Sign}, \texttt{Verify})$ consists of:

- $\texttt{KeyGen}(1^\lambda) \to (sk, pk)$: Generates a key pair
- $\texttt{Sign}(sk, m) \to \sigma$: Signs message m with secret key sk
- $\texttt{Verify}(pk, m, \sigma) \to \{0,1\}$: Verifies signature σ on message m

Solana uses Ed25519 signatures, providing 128-bit security with 64-byte signatures.

2.2 Solana Architecture

2.2.1 Account Model

Solana uses an account-based model where each account has:

- **Address**: A 32-byte Ed25519 public key
- **Lamports**: Balance in lamports (1 SOL = 10^9 lamports)
- **Data**: Arbitrary byte array storing account state
- **Owner**: The program with write access to the account
- **Executable**: Whether the account contains program code

2.2.2 Program Derived Addresses (PDAs)

A PDA is a deterministic address derived from:

$$\text{PDA}(\text{seeds}, \text{program\_id}) = \text{FindProgramAddress}(\text{seeds}, \text{program\_id}) \tag{1}$$

Where FindProgramAddress searches for a public key P such that:

$$P = H(\text{seeds} \parallel \text{program\_id} \parallel [b]) \quad \text{and} \quad P \notin E(\mathbb{F}_p) \tag{2}$$

Here $E(\mathbb{F}_p)$ is the Ed25519 elliptic curve, and $b \in [0, 255]$ is the "bump" seed. PDAs are off-curve points, ensuring no private key exists.

2.2.3 Token-2022 Extensions

Token-2022 is Solana's next-generation token program supporting extensions:

- **Transfer Hook**: Calls a specified program on every transfer
- **Transfer Fee**: Withholds a percentage of each transfer
- **Confidential Transfer**: Encrypts balances using ElGamal encryption

2.3 Bloom Filters

Definition 2.3 (Bloom Filter)**.** A Bloom filter is a probabilistic data structure for set membership testing. For a set $S = \{s_1, \ldots, s_n\}$:

- Bit array: $B[0..m-1]$ initialized to 0
- Hash functions: $h_1, \ldots, h_k : \{0,1\}^* \to [0, m-1]$
- Insert: For each $s \in S$, set $B[h_i(s)] = 1$ for $i = 1, \ldots, k$
- Query: Element $x \in S$ if $B[h_i(x)] = 1$ for all i

Theorem 2.1 (Bloom Filter False Positive Rate)**.** *For a Bloom filter with m bits, n elements, and k hash functions:*

$$P(\textit{false positive}) = \left(1 - e^{-kn/m}\right)^k \tag{3}$$

Optimal k minimizes false positives:

$$k_{\text{opt}} = \frac{m}{n} \ln 2 \tag{4}$$

2.4 Merkle Trees

Definition 2.4 (Merkle Tree)**.** A Merkle tree is a binary tree where:

- Leaves: $L_i = H(\text{data}_i)$ for $i = 0, \ldots, n-1$
- Internal nodes: $N_{i,j} = H(N_{i,2j} \parallel N_{i,2j+1})$
- Root: $r = N_{0,0}$

Theorem 2.2 (Merkle Proof Size)**.** *For a tree with n leaves, a membership proof requires $O(\log n)$ hashes.*

3 Policy Enforcement Layer (x0-guard)

3.1 Problem Statement

Consider an agent A owned by user U with private key sk_U. The agent requires a signing key sk_A to perform transactions autonomously. Without constraints, compromise of sk_A allows unbounded spending.

3.2 Design Goals

1. **Spend Limits**: Enforce maximum spending in rolling 24-hour windows
2. **Transaction Limits**: Cap individual transaction sizes
3. **Whitelist Verification**: Restrict recipients to approved addresses
4. **Privacy**: Support confidential (encrypted) transfers
5. **Auditability**: Maintain on-chain transaction history
6. **Revocability**: Allow owner to revoke agent authority

3.3 Architecture

3.3.1 AgentPolicy Account

Each agent has a Program Derived Address (PDA) storing its policy:

```
#[account]
pub struct AgentPolicy {
    pub version: u8,                // Account version (migration)
    pub owner: Pubkey,              // Cold wallet (full control)
    pub agent_signer: Pubkey,       // Hot key (delegated)
    pub daily_limit: u64,           // Max spend per 24h
    pub max_single_transaction: Option<u64>,
    pub rolling_window: Vec<SpendingEntry>,
    pub privacy_level: PrivacyLevel,
    pub whitelist_mode: WhitelistMode,
    pub whitelist_data: WhitelistData,
    pub is_active: bool,
    pub require_delegation: bool,   // Require token delegation
    pub bound_token_account: Option<Pubkey>,
    pub last_update_slot: u64,      // For rate limiting
    pub auditor_key: Option<Pubkey>,// Optional auditor
    pub blink_hour_start: i64,      // Blink rate limit window
    pub blinks_this_hour: u8,       // Blinks in current window
    pub bump: u8,
}

pub struct SpendingEntry {
    pub amount: u64,
    pub timestamp: i64,
}
```

Listing 1: AgentPolicy Account Structure

3.3.2 Transfer Hook Mechanism

Token-2022's transfer hook enables us to intercept every transfer. The flow is:

1. User initiates transfer of x tokens to recipient R
2. Token-2022 calls x0-guard's `validate_transfer`
3. x0-guard verifies:
   - Signer is authorized agent: $\texttt{signer} = \texttt{policy.agent\_signer}$
   - Spend limit not exceeded: $\sum_{t > t_{\text{now}} - 86400} \texttt{amount}_t + x \leq \texttt{daily\_limit}$
   - Transaction limit not exceeded: $x \leq \texttt{max\_single\_transaction}$
   - Recipient whitelisted: $R \in W$ (if whitelist enabled)
4. If validation passes, transfer proceeds; otherwise, reverts

3.4 Rolling Window Algorithm

Algorithm 1 Rolling Window Spend Limit Enforcement

procedure ValidateTransfer($\texttt{policy}, x, t_{\text{now}}$)
  $t_{\text{cutoff}} \leftarrow t_{\text{now}} - 86400$ ▷ 24 hours ago
  $\texttt{policy.rolling\_window} \leftarrow \texttt{policy.rolling\_window.retain}(|e| e.\mathit{timestamp} > t_{\text{cutoff}})$
  $\texttt{current\_spend} \leftarrow \sum_{e \in \texttt{rolling\_window}} e.\texttt{amount}$
  if $\texttt{current\_spend} + x > \texttt{policy.daily\_limit}$ **then**
    return `Error::DailyLimitExceeded`
  end if
  if $|\texttt{policy.rolling\_window}| \geq \texttt{MAX\_ENTRIES}$ **then**
    return `Error::WindowOverflow`
  end if
  $\texttt{policy.rolling\_window.push}(\{\texttt{amount} : x, \texttt{timestamp} : t_{\text{now}}\})$
  return `Success`
end procedure

Theorem 3.1 (Rolling Window Correctness)**.** *For a daily limit L and current time t, the rolling window algorithm ensures:*

$$\sum_{i : t_i > t - 86400} x_i \leq L \tag{5}$$

at all times t.

Proof. By induction on transactions. Base case: initially, the sum is $0 \leq L$. Inductive step: assume the invariant holds before transaction j with amount x_j. The algorithm rejects if:

$$\sum_{i : t_i > t_j - 86400} x_i + x_j > L \tag{6}$$

Therefore, if accepted:

$$\sum_{i : t_i > t_j - 86400} x_i + x_j \leq L \tag{7}$$

After adding x_j to the window, the sum equals $\sum_{i : t_i > t_j - 86400} x_i + x_j \leq L$. □ □

3.5 Whitelist Verification

Three whitelist modes are supported:

3.5.1 Merkle Mode

Store Merkle root r in policy. For transfer to R:

1. Agent provides proof $\pi = \{h_1, \ldots, h_{\log n}\}$
2. Verify: $\texttt{ComputeRoot}(H(R), \pi) = r$

Advantages: $O(\log n)$ proof size, deterministic verification
Disadvantages: Proof must be provided by agent, updates require new root

3.5.2 Bloom Mode

Store Bloom filter B in policy. For transfer to R:

1. Compute $h_i(R)$ for $i = 1, \ldots, k$
2. Check: $B[h_i(R)] = 1$ for all i

Advantages: $O(1)$ verification, no proof required
Disadvantages: False positives, filter stored on-chain (4KB)
For $n = 1000$ addresses, $m = 4096 \times 8 = 32768$ bits, $k = 7$:

$$P(\text{FP}) = \left(1 - e^{-7 \times 1000/32768}\right)^7 \approx 0.008 = 0.8\% \tag{8}$$

3.5.3 Domain Mode

Store domain prefixes $\{d_1, \ldots, d_m\}$ (first 8 bytes of addresses). For transfer to R:

1. Extract prefix: $p = R[0..7]$
2. Check: $p \in \{d_1, \ldots, d_m\}$

Advantages: Allows "vanity addresses", compact storage
Disadvantages: Lower security (8-byte prefixes), linear scan

3.6 Privacy Levels

3.6.1 Public Mode

Standard SPL transfers with visible amounts.

3.6.2 Confidential Mode

Uses Token-2022 confidential transfer extension:

1. Balances encrypted with ElGamal: $C = (g^r, g^r \cdot h^b)$ where b is balance
2. Transfers use range proofs to prevent overflow

3. Optional auditor can decrypt amounts

Definition 3.1 (ElGamal Encryption)**.** Public key: (g, h) where $h = g^x$ and x is secret. To encrypt m:

$$\texttt{Enc}(m) = (g^r, h^r \cdot g^m) \tag{9}$$

for random r. Decryption:

$$\texttt{Dec}((c_1, c_2)) = \frac{c_2}{c_1^x} = g^m \tag{10}$$

Then solve discrete log to recover m (feasible for small m).

3.7 Reentrancy Protection

Theorem 3.2 (State-Before-Transfer Invariant)**.** *For all escrow/wrapper operations, state updates occur before token transfers. This prevents reentrancy attacks.*

```rust
pub fn release_funds(ctx: Context<ReleaseFunds>) -> Result<()> {
    let escrow = &mut ctx.accounts.escrow;

    // CRITICAL: Update state BEFORE transfer
    let amount = escrow.amount;
    escrow.state = EscrowState::Released;

    // Now transfer (if reentrant call occurs, state check fails)
    token::transfer(/* ... */, amount)?;

    Ok(())
}
```

Listing 2: Reentrancy Protection Pattern

4 HTTP 402 Protocol

4.1 Problem Statement

Existing payment protocols lack standardization for agent-to-agent negotiation. HTTP provides status codes for various conditions but lacks a payment-specific code beyond 402 (Payment Required), which was reserved but never specified.

4.2 Protocol Flow

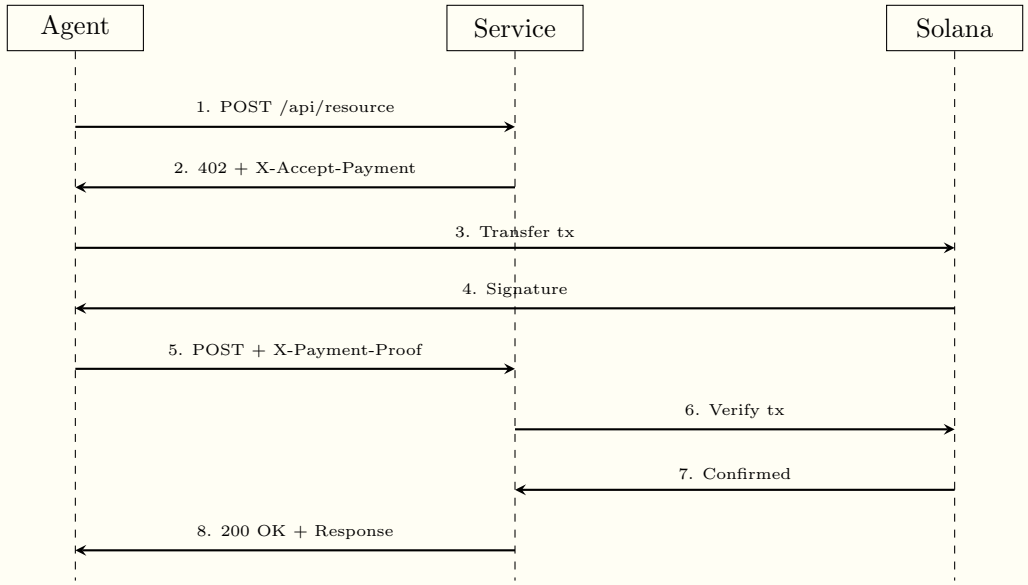

Figure 2: x402 Payment Flow. Agent receives 402, pays on-chain, then retries with proof.

4.3 Protocol Design

4.3.1 Payment Request

When a service requires payment, it responds with HTTP 402 and an `X-Accept-Payment` header:

```
HTTP/1.1 402 Payment Required
X-Accept-Payment: <base64-encoded-payment-request>
Content-Type: application/json

{
    "error": "payment_required",
    "message": "This endpoint requires payment"
}
```

Listing 3: HTTP 402 Payment Required Response

The payment request (base64-decoded) contains:

```
{
    "version": "x0-v1",
    "recipient": "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU",
    "amount": "1000000",
    "resource": "/api/v1/generate",
    "memo": "Text generation request",
    "network": "solana-mainnet",
    "escrow": {
        "use_escrow": false,
        "delivery_timeout": 3600,
        "auto_release_delay": 86400,
        "arbiter": null
    }
}
```

Listing 4: Payment Request Structure

4.3.2 Payment Proof

After payment, the agent includes proof in subsequent requests:

```
POST /api/v1/generate HTTP/1.1
X-Payment-Proof: <base64-encoded-proof>
X-Payment-Version: x0-v1
Content-Type: application/json

{
    "prompt": "Generate a haiku about blockchain"
}
```

Listing 5: HTTP Request with Payment Proof

Payment proof contains:

```
{
    "signature": "5VDx8F...",  // Transaction signature
    "slot": 123456789,
    "payer": "9xQeWv...",
    "timestamp": 1706400000,
    "network": "solana-mainnet"
}
```

Listing 6: Payment Proof Structure

4.3.3 Verification

The service verifies payment:

1. Fetch transaction by signature
2. Verify transaction succeeded
3. Check recipient matches service wallet
4. Check amount $\geq$ requested amount
5. Verify timestamp within tolerance (± 5 minutes)
6. Check memo matches request (via SHA-256)

4.4 Security Properties

Theorem 4.1 (Payment Non-Repudiation)**.** *A valid payment proof is unforgeable. An adversary cannot construct a proof without executing the on-chain transaction.*

Proof. The proof contains transaction signature $\sigma = \texttt{Sign}(sk_A, tx)$. By existential unforgeability of Ed25519, an adversary cannot produce σ' such that $\texttt{Verify}(pk_A, tx, \sigma') = 1$ without sk_A. On-chain verification ensures the transaction executed. □ □

5 Conditional Escrow (x0-escrow)

5.1 Design

Escrow enables trustless payments for services with uncertain delivery.

5.1.1 Escrow State Machine

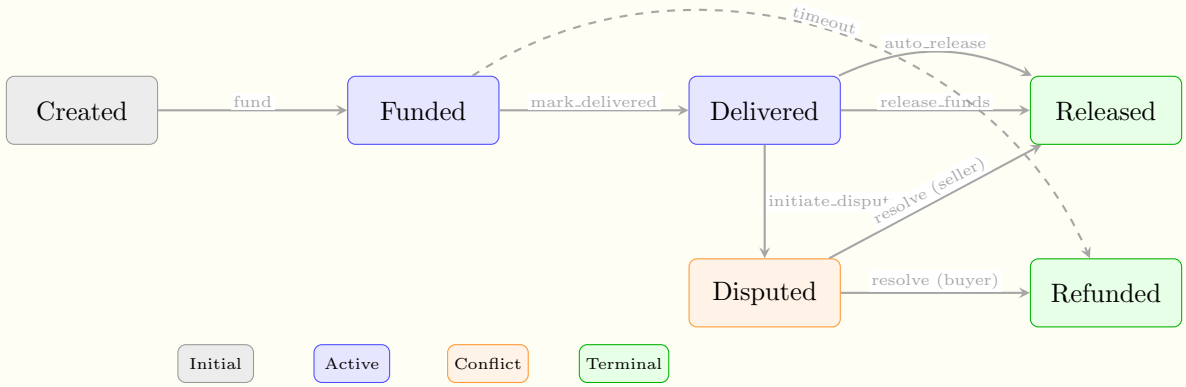

Figure 3: Escrow State Machine. States are colored by type: gray (initial), blue (active), orange (conflict), green (terminal).

5.1.2 Escrow Account

```rust
#[account]
pub struct EscrowAccount {
    pub version: u8,                // Account version
    pub buyer: Pubkey,
    pub seller: Pubkey,
    pub arbiter: Option<Pubkey>,
    pub amount: u64,
    pub memo_hash: [u8; 32],
    pub state: EscrowState,
    pub timeout: i64,
    pub created_at: i64,
    pub delivery_proof: Option<[u8; 32]>,
    pub dispute_evidence: Option<[u8; 32]>,
    pub mint: Pubkey,
    pub token_decimals: u8,
    pub dispute_initiated_slot: u64, // MEDIUM-6: arbiter delay
    pub bump: u8,
}
```

Listing 7: Escrow Account Structure

5.2 Key Operations

5.2.1 Create and Fund

5.2.2 Delivery and Release

5.2.3 Auto-Release

After delivery, if buyer doesn't dispute within timeout, seller can claim:

Algorithm 2 Create and Fund Escrow

procedure CREATEESCROW(B, S, A, x, m, τ)
  require $B \neq S$
  require $3600 \leq \tau \leq 2592000$ ▷ 1h to 30d
  $e \leftarrow$ `EscrowPDA`$(B, S, H(m))$
  e.`timeout` $\leftarrow t_{\text{now}} + \tau$
  e.`state` $\leftarrow$ `Created`
  return e
end procedure
procedure FUNDESCROW(e, x)
  require e.`state` = `Created`
  require $t_{\text{now}} < e$.`timeout`
  `transfer`(B, e, x)
  e.`state` $\leftarrow$ `Funded`
end procedure

Algorithm 3 Mark Delivered and Release Funds

procedure MARKDELIVERED(e, p)
  require `signer` = e.`seller`
  require e.`state` = `Funded`
  e.`delivery_proof` $\leftarrow H(p)$
  e.`state` $\leftarrow$ `Delivered`
end procedure
procedure RELEASEFUNDS(e)
  require `signer` = e.`buyer`
  require e.`state` = `Delivered`
  e.`state` $\leftarrow$ `Released` ▷ BEFORE transfer!
  `transfer`$(e, e$.`seller`$, e$.`amount`$)$
  `UpdateReputation`$(e$.`seller`$,$ `Success`$)$ ▷ Optional CPI
end procedure

Algorithm 4 Auto-Release After Timeout

procedure CLAIMAUTORELEASE(e)
  require `signer` = e.`seller`
  require e.`state` = `Delivered`
  require $t_{\text{now}} > e$.`timeout`
  e.`state` $\leftarrow$ `Released`
  `transfer`$(e, e$.`seller`$, e$.`amount`$)$
  `UpdateReputation`$(e$.`seller`$,$ `Success`$)$
end procedure

Algorithm 5 Initiate and Resolve Dispute

```
procedure InitiateDispute(e, d)
    require signer ∈ {e.buyer, e.seller}
    require e.state = Delivered
    e.dispute_evidence ← H(d)
    e.dispute_initiated_slot ← current_slot
    e.state ← Disputed
    UpdateReputation(e.seller, Dispute)
end procedure
procedure ResolveDispute(e, release_to_seller)
    require signer = e.arbiter
    require e.state = Disputed
    require current_slot ≥ e.dispute_initiated_slot + Δ
                                                ▷ Δ = 216000 slots ≈ 24 hours
    if release_to_seller then
        e.state ← Released
        transfer(e, e.seller, e.amount)
        UpdateReputation(e.seller, ResolutionFavor)
    else
        e.state ← Refunded
        transfer(e, e.buyer, e.amount)
    end if
end procedure
```

5.2.4 Dispute Resolution

5.3 Security Analysis

Theorem 5.1 (Escrow Safety)**.** *An escrow satisfies:*

1. ***Atomicity****: Funds are either fully released or refunded, never partially*
2. ***Liveness****: Funds are never permanently locked*
3. ***Fairness****: Either party can initiate dispute; arbiter is neutral*

Proof. **Atomicity**: State transitions are atomic. The `Released` and `Refunded` states are terminal.

Liveness: Three exit paths exist:

- Buyer releases after delivery
- Seller claims auto-release after timeout
- Arbiter resolves dispute

At least one path is always available after delivery or timeout.

Fairness: Either party can initiate dispute. Arbiter resolution requires evidence from both parties and a 24-hour delay. □ □

6 Reputation Oracle (x0-reputation)

6.1 Design Goals

1. **Transparency**: All reputation data on-chain
2. **Temporal Decay**: Old successes shouldn't dominate forever
3. **Sybil Resistance**: New agents don't get perfect scores
4. **Nuanced Scoring**: Distinguish success, failure, disputes, resolutions

6.2 Reputation Account

```rust
#[account]
pub struct AgentReputation {
    pub version: u8,                    // Account version (v2)
    pub agent_id: Pubkey,
    pub total_transactions: u64,
    pub successful_transactions: u64,
    pub disputed_transactions: u64,
    pub resolved_in_favor: u64,
    pub failed_transactions: u64,       // Policy rejections
    pub average_response_time_ms: u32,
    pub cumulative_response_time_ms: u64,
    pub last_updated: i64,
    pub last_decay_applied: i64,
    pub bump: u8,
}
```

Listing 8: Reputation Account Structure

6.3 Reputation Score Calculation

Definition 6.1 (Reputation Score)**.** The reputation score $S \in [0, 1]$ is computed as:

$$\begin{aligned} S = {} & 0.60 \cdot S_{\text{success}} + 0.15 \cdot S_{\text{resolution}} \\ & + 0.10 \cdot (1 - R_{\text{dispute}}) + 0.15 \cdot (1 - R_{\text{failure}}) \end{aligned} \tag{11}$$

where:

$$S_{\text{success}} = \frac{n_{\text{success}}}{n_{\text{total}}} \tag{12}$$

$$S_{\text{resolution}} = \begin{cases} \frac{n_{\text{resolved}}}{n_{\text{disputed}}} & n_{\text{disputed}} > 0 \\ 0.5 & n_{\text{disputed}} = 0 \land n_{\text{total}} < 10 \\ 1.0 & n_{\text{disputed}} = 0 \land n_{\text{total}} \geq 10 \end{cases} \tag{13}$$

$$R_{\text{dispute}} = \frac{n_{\text{disputed}}}{n_{\text{total}}} \tag{14}$$

$$R_{\text{failure}} = \frac{n_{\text{failed}}}{n_{\text{total}}} \tag{15}$$

Remark 6.1. The neutral resolution score (0.5) for new agents prevents Sybil attacks where attackers create new identities to get perfect scores.

Example 6.1 (Reputation Score Calculation). Consider an agent with the following transaction history:

- Total transactions: $n_{\text{total}} = 100$
- Successful: $n_{\text{success}} = 90$
- Disputed: $n_{\text{disputed}} = 5$
- Resolved in favor: $n_{\text{resolved}} = 3$
- Failed (policy rejections): $n_{\text{failed}} = 2$

Computing component scores:

$$\begin{aligned} S_{\text{success}} &= \frac{90}{100} = 0.90 \\ S_{\text{resolution}} &= \frac{3}{5} = 0.60 \quad (\text{since } n_{\text{disputed}} > 0) \\ R_{\text{dispute}} &= \frac{5}{100} = 0.05 \\ R_{\text{failure}} &= \frac{2}{100} = 0.02 \end{aligned}$$

Final reputation score:

$$\begin{aligned} S &= 0.60(0.90) + 0.15(0.60) + 0.10(1 - 0.05) + 0.15(1 - 0.02) \\ &= 0.540 + 0.090 + 0.095 + 0.147 \\ &= \mathbf{0.872} \end{aligned}$$

This agent has a strong reputation (87.2%), with room for improvement in dispute resolution.

6.4 Temporal Decay

Definition 6.2 (Exponential Decay)**.** Monthly decay applies to successful transactions:

$$n_{\text{success}}^{(t+1)} = n_{\text{success}}^{(t)} \cdot (1 - \alpha) \tag{16}$$

where $\alpha = 0.01$ (1% monthly decay) and t is measured in months.

For m months:

$$n_{\text{success}}^{(t+m)} = n_{\text{success}}^{(t)} \cdot (0.99)^m \tag{17}$$

Proposition 6.1 (Half-Life)**.** *The half-life of reputation is:*

$$t_{1/2} = \frac{\ln 2}{\ln(1/0.99)} \approx 69 \text{ months} \tag{18}$$

Algorithm 6 Apply Reputation Decay

procedure ApplyDecay(r)
  $m \leftarrow \lfloor (t_{\text{now}} - r.\texttt{last\_decay\_applied})/2592000 \rfloor$
  if $m > 0$ **then**
    $d_{\text{mult}} \leftarrow 99^{\min(m,12)}$ ▷ Cap at 12 months
    $d_{\text{div}} \leftarrow 100^{\min(m,12)}$
    $r.\texttt{successful\_transactions} \leftarrow r.\texttt{successful\_transactions} \cdot d_{\text{mult}}/d_{\text{div}}$
    $r.\texttt{last\_decay\_applied} \leftarrow t_{\text{now}}$
  end if
end procedure

6.5 Authorization Model

Definition 6.3 (Authorized Callers)**.** Reputation updates can only be called by:

- **Escrow program**: Records success/dispute/resolution
- **Guard program**: Records policy failures
- **Policy owner**: Self-reported off-chain transactions

Theorem 6.2 (Reputation Integrity)**.** *An adversary cannot artificially inflate reputation without:*

1. *Completing escrow transactions (requires payment)*
2. *Winning disputes (requires arbiter approval)*
3. *Controlling policy owner key (equivalent to ownership)*

7 Agent Discovery (x0-registry)

7.1 Registry Entry

```rust
#[account]
pub struct AgentRegistry {
    pub version: u8,                    // Account version
    pub agent_id: Pubkey,               // Policy PDA (primary ID)
    pub owner: Pubkey,                  // Owner who can update
    pub endpoint: String,               // Service URL (max 256 chars)
    pub capabilities: Vec<Capability>, // Max 10 capabilities
    pub price_oracle: Option<Pubkey>,  // Optional pricing oracle
    pub reputation_pda: Pubkey,         // Linked reputation account
    pub last_updated: i64,
    pub is_active: bool,
    pub bump: u8,
}

pub struct Capability {
    pub capability_type: String,        // Max 64 chars
    pub metadata: String,               // JSON metadata (max 256 chars)
}
```

Listing 9: Registry Entry Structure

The `Capability` struct uses a JSON `metadata` field to encode pricing, versioning, and service-specific parameters (see Section 7.3), avoiding rigid on-chain schema constraints.

7.2 Discovery Algorithm

Algorithm 7 Find Agents by Capability

procedure FINDAGENTS(`cap_type`)
  $A \leftarrow \{\}$ ▷ Set of matching agents
  for $e \in$ `GetProgramAccounts(registry_program)` **do**
    if `cap_type` $\in e.$`capabilities` $\wedge$ $e.$`is_active` **then**
      $s \leftarrow$ `FetchReputation`($e.$`reputation_pda`)
      $A \leftarrow A \cup \{(e, s)\}$
    end if
  end for
  return `SortByScore`(A)
end procedure

7.3 Capability Metadata

Capabilities use JSON metadata:

```
{
    "type": "text-generation",
    "models": ["gpt-4", "claude-3"],
    "pricing": {
        "per_token": 0.00001,
        "minimum": 0.01
    },
    "rate_limit": {
        "requests_per_minute": 60,
        "burst": 10
    },
    "api_version": "v1"
}
```

Listing 10: Capability Metadata Example

8 USDC Wrapper (x0-wrapper)

8.1 Problem Statement

Direct USDC usage has limitations:

1. No transfer hook support on standard USDC
2. Different token programs (SPL vs Token-2022)
3. Protocol fees require wrapper

8.2 Design

x0-USD is a 1:1 USDC-backed Token-2022 token with:

- Transfer hook pointing to x0-guard
- 0.8% transfer fee (configurable via `PROTOCOL_FEE_BASIS_POINTS` constant, default 80 bps)
- Configurable redemption fee (default 0.8%, set via `WRAPPER_REDEMPTION_FEE_BPS`)

8.2.1 State Accounts

The wrapper maintains two on-chain accounts for configuration and operational metrics:

```rust
#[account]
pub struct WrapperConfig {
    pub admin: Pubkey,                  // Admin key (should be multisig)
    pub pending_admin: Option<Pubkey>, // Two-step admin transfer
    pub usdc_mint: Pubkey,              // Underlying USDC mint
    pub wrapper_mint: Pubkey,           // x0-USD mint
    pub reserve_account: Pubkey,        // USDC reserve token account
    pub redemption_fee_bps: u16,        // Fee in basis points
    pub is_paused: bool,                // Emergency pause flag
    pub bump: u8,
}
```

Listing 11: Wrapper Configuration Account

```rust
#[account]
pub struct WrapperStats {
    pub reserve_usdc_balance: u64,      // Current USDC in reserve
    pub outstanding_wrapper_supply: u64, // Outstanding x0-USD supply
    pub total_deposits: u64,            // All-time deposits
    pub total_redemptions: u64,         // All-time redemptions
    pub total_fees_collected: u64,      // All-time fees
    pub daily_redemption_volume: u64,   // Resets every 24h
    pub daily_redemption_reset_timestamp: i64,
    pub last_updated: i64,
    pub bump: u8,
}
```

Listing 12: Wrapper Statistics Account

The reserve ratio ρ can be computed directly from `WrapperStats`:

$$\rho = \frac{\texttt{reserve\_usdc\_balance}}{\texttt{outstanding\_wrapper\_supply}} \tag{19}$$

8.2.2 Reserve Invariant

Definition 8.1 (Reserve Invariant)**.** At all times, the following must hold:

$$R_{\text{USDC}} \geq S_{\text{x0-USD}} \tag{20}$$

where R_{USDC} is USDC reserve and $S_{\text{x0-USD}}$ is outstanding x0-USD supply.

Theorem 8.1 (Invariant Preservation)**.** *The reserve invariant is preserved under all operations.*

Proof. We prove by cases:

Deposit: User deposits x USDC, receives x x0-USD.

$$R' = R + x, \quad S' = S + x \implies R' - S' = R - S \geq 0 \tag{21}$$

Redemption: User burns x x0-USD, receives $x - f$ USDC (fee f).

$$R' = R - (x - f), \quad S' = S - x \tag{22}$$

$$R' - S' = R - x + f - S + x = R - S + f \geq R - S \geq 0 \tag{23}$$

The fee increases the reserve ratio. □ □

8.3 Governance

8.3.1 Timelock

All admin operations require 48-hour timelock. Each pending action is stored in a PDA:

```rust
#[account]
pub struct AdminAction {
    pub action_type: AdminActionType,  // SetFeeRate | SetPaused |
    EmergencyWithdraw | TransferAdmin
    pub scheduled_timestamp: i64,      // Earliest execution time
    pub new_value: u64,                // Interpretation depends on action_type
    pub new_admin: Pubkey,             // For TransferAdmin actions
    pub destination: Pubkey,           // For EmergencyWithdraw
    pub executed: bool,
    pub cancelled: bool,
    pub bump: u8,
}
```

Listing 13: Timelocked Admin Action

An action is executable if and only if $t_{\text{now}} \geq$ `scheduled_timestamp` and $\neg$`executed` $\wedge$ $\neg$`cancelled`.

Algorithm 8 Timelock Pattern

procedure ScheduleAction(a, v, t)
    `action_pda` $\leftarrow$ `CreateActionPDA`(a, `nonce`)
    `action_pda.type` $\leftarrow a$
    `action_pda.value` $\leftarrow v$
    `action_pda.scheduled_time` $\leftarrow t + 172800$ ▷ 48h
end procedure
procedure ExecuteAction(`action_pda`)
    require $t_{\text{now}} \geq$ `action_pda.scheduled_time`
    require $\neg$`action_pda.executed`
    require $\neg$`action_pda.cancelled`
    `ApplyAction`(`action_pda.type`, `action_pda.value`)
    `action_pda.executed` $\leftarrow$ `true`
end procedure

8.3.2 Emergency Pause

Emergency pause is the **only** operation bypassing timelock:

- Can only **pause**, never unpause
- Unpausing requires standard timelock
- Prevents rapid pause/unpause attacks

8.4 Reserve Ratio Monitoring

Definition 8.2 (Reserve Ratio)**.**

$$\rho = \frac{R_{\text{USDC}}}{S_{\text{x0-USD}}} \tag{24}$$

Alert levels:

- $\rho < 1.01$: Warning alert
- $\rho < 1.00$: Critical alert (undercollateralized)

9 Human-in-the-Loop (Blinks)

9.1 Problem Statement

Full automation is dangerous for:

1. Transfers exceeding daily limit
2. Recipients not on whitelist
3. Unusually large transactions

9.2 Blink Design

A **Blink** is a Solana Action requesting human approval.

9.2.1 Generation

When guard rejects a transfer due to limits, it can:

1. Generate Blink ID: $b = H(\texttt{policy} \parallel R \parallel x \parallel t)[0..16]$
2. Emit `BlinkGenerated` event with:
   - Blink ID
   - Requested amount
   - Recipient
   - Expiration (15 minutes)
3. Charge fee (0.001 SOL) to prevent spam

Algorithm 9 Blink Rate Limit Check

procedure CHECKBLINKRATELIMIT(p, t)
  $h_{\text{current}} \leftarrow \lfloor t/3600 \rfloor$
  $h_{\text{window}} \leftarrow \lfloor p.\texttt{blink\_hour\_start}/3600 \rfloor$
  if $h_{\text{current}} \neq h_{\text{window}}$ **then**
    $p.\texttt{blink\_hour\_start} \leftarrow t$
    $p.\texttt{blinks\_this\_hour} \leftarrow 0$
  end if
  if $p.\texttt{blinks\_this\_hour} \geq 3$ **then**
    return `false`
  end if
  $p.\texttt{blinks\_this\_hour} \leftarrow p.\texttt{blinks\_this\_hour} + 1$
  return `true`
end procedure

9.2.2 Rate Limiting

Definition 9.1 (Blink Rate Limit)**.** Maximum 3 Blinks per hour per policy.

9.2.3 Approval

Owner approves via:

1. Viewing Blink details (QR code, web interface, wallet)
2. Signing approval transaction
3. Temporarily raising limit or whitelisting recipient

10 Economic Model

10.1 Fee Structure

Operation	Fee	Recipient
Transfer (x0-USD)	0.8%	Protocol treasury
Registry listing	0.1 SOL	Protocol treasury
Blink generation	0.001 SOL	Protocol treasury
Wrapper redemption	0.8% (configurable)	Wrapper reserve

Table 1: Protocol Fee Schedule

10.2 Fee Collection

Transfer fees are collected via Token-2022's TransferFee extension:

Algorithm 10 Fee Harvesting

```
procedure HarvestFees(mint, accounts)
    for a ∈ accounts do
        f ← GetWithheldAmount(a)
        TransferWithheldToMint(a, f)
    end for
    F ← Σ_a f
    WithdrawWithheldFromMint(mint, treasury, F)
end procedure
```

10.3 Token Economics

10.3.1 x0-USD Supply Dynamics

$$\frac{dS}{dt} = D(t) - R(t) \tag{25}$$

$$\frac{dR_{\text{USDC}}}{dt} = D(t) - R(t) + f \cdot R(t) \tag{26}$$

where:

- $S(t)$ = x0-USD supply
- $R_{\text{USDC}}(t)$ = USDC reserve
- $D(t)$ = Deposit rate
- $R(t)$ = Redemption rate (before fees)
- f = Redemption fee rate

Reserve ratio evolution:

$$\rho(t) = \frac{R_{\text{USDC}}(t)}{S(t)} = 1 + \int_0^t \frac{f \cdot R(\tau)}{S(\tau)} d\tau \tag{27}$$

The reserve ratio increases over time due to fees.

11 Zero-Knowledge Proof Verification (x0-zk-verifier)

11.1 Problem Statement

Token-2022's Confidential Transfer extension encrypts token balances and transfer amounts using twisted ElGamal encryption over the Ristretto255 curve. To ensure that encrypted operations preserve token invariants (e.g., non-negative balances, valid public keys), the protocol requires zero-knowledge proofs that attest to the correctness of ciphertext operations without revealing plaintext values.

The challenge is two-fold:

1. **Client-side proof generation**: Proofs must be generated off-chain where the user holds the ElGamal secret key, using Groth16 proving circuits from `solana-zk-token-sdk`.

2. **On-chain proof verification**: Proofs must be verified on-chain and the verification result stored in a state account that Token-2022 can reference during confidential transfer execution.

11.2 Cryptographic Foundations

11.2.1 Twisted ElGamal Encryption

Token-2022 uses a *twisted* variant of ElGamal encryption over the Ristretto255 group, which provides both encryption and efficient homomorphic addition:

Definition 11.1 (Twisted ElGamal Encryption)**.** For a public key $P = s \cdot G$ where s is the secret scalar and G is the Ristretto255 basepoint:

$$\texttt{Encrypt}(P, v) = (r \cdot G,\ r \cdot P + v \cdot H) \tag{28}$$

$$\texttt{Decrypt}(s, (C_1, C_2)) = C_2 - s \cdot C_1 = v \cdot H \tag{29}$$

where $r \xleftarrow{\$} \mathbb{Z}_q$ is a random scalar, H is a second generator, and v is the plaintext value. Decryption recovers $v \cdot H$, from which v is obtained via brute-force discrete log (feasible for $v < 2^{48}$).

11.2.2 Groth16 Proof System

The protocol uses Groth16 [11] for succinct non-interactive zero-knowledge proofs:

Definition 11.2 (Groth16 Proof)**.** A Groth16 proof $\pi = (A, B, C) \in \mathbb{G}_1 \times \mathbb{G}_2 \times \mathbb{G}_1$ proves knowledge of a witness w satisfying a rank-1 constraint system $R(x, w) = 1$, where x is the public input. The proof satisfies:

- **Completeness**: An honest prover always produces a valid proof
- **Soundness**: No efficient adversary can produce a valid proof for a false statement
- **Zero-knowledge**: The proof reveals nothing about w beyond $R(x, w) = 1$

Proof size is constant: 192 bytes regardless of circuit complexity.

11.3 Proof Types

The verifier supports four proof types corresponding to Token-2022 confidential transfer operations:

11.3.1 PubkeyValidityProof

Proves that a given 32-byte value is a valid ElGamal public key (i.e., a point on the Ristretto255 curve that was correctly derived from a secret key).

$$\text{Prove} : \exists\ s \in \mathbb{Z}_q \text{ such that } P = s \cdot G \tag{30}$$

Required when configuring a token account for confidential transfers. Proof data is 64 bytes.

11.3.2 ZeroBalanceProof

Proves that an ElGamal ciphertext encrypts the value zero, required for account closure:

$$\text{Prove} : (C_1, C_2) = \texttt{Encrypt}(P, 0) \implies C_2 = r \cdot P \tag{31}$$

Proof data is 96 bytes.

11.3.3 WithdrawProof

Proves that a withdrawal of amount a from an encrypted balance B is valid:

$$\text{Prove} : \texttt{Decrypt}(s, B) = v \geq a \ \wedge \ B' = B - \texttt{Encrypt}(P, a) \tag{32}$$

where v is the current balance and B' is the new encrypted balance. This ensures the account has sufficient funds without revealing either the balance or the withdrawal amount on-chain. Proof data is 160 bytes.

11.3.4 TransferProof

Proves the correctness of a confidential transfer between two accounts:

$$\text{Prove} : \texttt{Decrypt}(s_{\text{src}}, B_{\text{src}}) \geq a \ \wedge \ B'_{\text{src}} = B_{\text{src}} - \texttt{Encrypt}(P_{\text{src}}, a) \ \wedge \ \Delta_{\text{dst}} = \texttt{Encrypt}(P_{\text{dst}}, a) \tag{33}$$

This is the most complex proof, attesting simultaneously that the sender has sufficient balance, the sender's balance is correctly decremented, and the recipient's balance increment is correctly encrypted under the recipient's public key.

11.4 Architecture

11.4.1 Proof Context Account

Successful verification creates a `ProofContext` PDA that stores the verification result:

```
#[account]
pub struct ProofContext {
    pub version: u8,
    pub proof_type: ProofType,          // PubkeyValidity | Withdraw |
    ZeroBalance | Transfer
    pub verified: bool,
    pub owner: Pubkey,                  // Account owner who can use this proof
    pub verified_at: i64,               // Unix timestamp
    pub amount: Option<u64>,            // For Withdraw/Transfer proofs
    pub recipient: Option<Pubkey>,      // For Transfer proofs
    pub elgamal_pubkey: Option<[u8; 32]>, // For PubkeyValidity proofs
    pub mint: Pubkey,                   // Token mint this proof is for
    pub token_account: Pubkey,          // Token account this proof is for
    pub bump: u8,
}
```

Listing 14: Proof Context State Account

11.4.2 Proof Freshness

To prevent replay attacks, proof contexts enforce a temporal validity window:

Definition 11.3 (Proof Freshness)**.** A proof context C is *fresh* at time t if and only if:

$$t - C.\texttt{verified\_at} < \Delta_{\text{proof}} = 300 \text{ seconds} \tag{34}$$

The 5-minute window provides sufficient time for transaction construction and submission while limiting the replay window.

11.4.3 Amount Bounds

All confidential amounts are bounded:

$$0 \leq a \leq 2^{48} - 1 = 281{,}474{,}976{,}710{,}655 \tag{35}$$

This constraint arises from the brute-force discrete log required during ElGamal decryption. The `solana-zk-token-sdk` uses a precomputed lookup table for values up to 2^{16}, combined with baby-step-giant-step for the remaining 2^{32} range.

11.5 Off-Chain Proof Generation (x0-zk-proofs)

The `x0-zk-proofs` crate is a Rust library compiled to WebAssembly via `wasm-pack`, providing client-side proof generation for the TypeScript SDK:

```rust
#[wasm_bindgen]
pub fn generate_pubkey_validity_proof(
    keypair_bytes: &[u8]        // 64-byte ElGamalKeypair
) -> Result<Vec<u8>, JsValue>;  // 64-byte PubkeyValidityData

#[wasm_bindgen]
pub fn generate_zero_balance_proof(
    keypair_bytes: &[u8]        // 64-byte ElGamalKeypair
) -> Result<Vec<u8>, JsValue>;  // 96-byte ZeroBalanceProofData

#[wasm_bindgen]
pub fn generate_withdraw_proof(
    keypair_bytes: &[u8],       // 64-byte ElGamalKeypair
    current_balance: u64,       // Current decrypted balance
    withdraw_amount: u64        // Amount to withdraw
) -> Result<Vec<u8>, JsValue>;  // 160-byte WithdrawData
```

Listing 15: x0-zk-proofs WASM Interface

The WASM module internally uses:

- `solana-zk-token-sdk v1.18.22` for Groth16 circuit construction and proving
- `ElGamalKeypair` reconstruction from 64-byte serialized form
- `bytemuck` for zero-copy POD serialization of proof data
- Authenticated encryption (AE) ciphertexts for decryptable balance hints

11.6 Verification Flow

The end-to-end flow for a confidential withdrawal is:

Algorithm 11 Confidential Withdrawal with ZK Proof

procedure ConfidentialWithdraw(`keypair`, b, a)
  require $a \leq b$ ▷ Client-side: sufficient balance
  require $a \leq 2^{48} - 1$ ▷ Client-side: amount bound
  $\pi \leftarrow$ `x0_zk_proofs::generate_withdraw_proof`(`keypair`, b, a)
  send `x0_zk_verifier::verify_withdraw`($\pi, a,$ `new_ae_balance`)
  $C \leftarrow$ `ProofContext PDA` ▷ On-chain: created after verification
  require C.`verified` = `true` ▷ On-chain check
  send `Token-2022::confidential_withdraw`(a, C)
end procedure

11.7 Security Properties

Theorem 11.1 (Balance Confidentiality)**.** *An adversary observing the blockchain cannot determine any agent's confidential balance, even given all historic proof contexts and ciphertexts.*

Proof. Token balances are encrypted under twisted ElGamal with semantic security. Proof contexts store only the verification result, proof type, and amount (for withdrawals), but the *remaining balance* is revealed only via AE ciphertext decryptable solely by the account holder. The Groth16 zero-knowledge property ensures proofs reveal nothing beyond the statement's truth. □ □

Theorem 11.2 (Proof Unforgeability)**.** *No computationally bounded adversary can produce a valid proof for a false statement (e.g., withdrawing more than the encrypted balance).*

Proof. Follows from the knowledge-soundness of Groth16 under the q-type knowledge-of-exponent assumption in bilinear groups. The verifier program re-derives the verification via `solana-zk-token-sdk`'s `verify()` method, which checks the Groth16 pairing equation. □ □

12 Security Analysis

12.1 Threat Model

We consider the following adversaries:

1. **Compromised Agent**: Attacker obtains sk_A
2. **Malicious Service**: Service provider attempts fraud
3. **Malicious Agent**: Agent attempts reputation manipulation
4. **Wrapper Attack**: Attacker attempts reserve drain

12.2 Attack Scenarios and Mitigations

12.2.1 Compromised Agent Key

Attack: Adversary steals sk_A, attempts unlimited spending.
Mitigation:

- Rolling window limits spending to L per 24 hours
- Owner can revoke via `revoke_agent_authority`
- Maximum loss bounded by $L + x_{\max}$ where $x_{\max}$ is max transaction size

12.2.2 Sybil Attack on Reputation

Attack: Adversary creates multiple identities to appear trustworthy.
Mitigation:

- Registry listing costs 0.1 SOL
- New agents get neutral (0.5) resolution score, not perfect
- Temporal decay prevents dormant identities

Cost for 100 Sybil identities: $100 \times 0.1 = 10$ SOL $\approx$ \$1000 (at \$100/SOL).

12.2.3 Escrow Griefing

Attack: Malicious buyer creates many escrows, ties up seller funds.
Mitigation:

- Escrow creation is permissionless but buyer must fund
- Sellers can set minimum transaction amounts
- Reputation tracking penalizes frivolous disputes

12.2.4 Wrapper Reserve Drain

Attack: Exploit bug to withdraw USDC without burning x0-USD.
Mitigation:

- Reserve invariant checked before every redemption
- State updated before transfer (reentrancy protection)
- Admin operations timelocked (48h notice)
- Emergency pause available

12.2.5 Transfer Hook Configuration Attack

Attack: Unauthorized party front-runs transfer hook initialization for a mint, potentially misconfiguring the extra account metas or consuming the PDA address.

Mitigation:

- Only the Token-2022 mint authority can initialize extra account metas
- Re-initialization is blocked after first valid configuration
- Extra account metas configuration is deterministic (not caller-supplied)
- Mint authority verification uses `StateWithExtensions` for robust parsing

Impact: Even though the extra metas configuration is hardcoded in the program, preventing unauthorized initialization eliminates a griefing vector where an attacker could pay the rent and claim the PDA before the legitimate mint authority initializes it.

12.3 Formal Verification

12.3.1 Spend Limit Invariant

Theorem 12.1 (Daily Spend Limit)**.** *For all execution traces, if policy has daily limit L:*

$$\forall t : \sum_{i:t_i > t-86400} x_i \leq L \tag{36}$$

Proof. Proven in Section 3.4. □ □

12.3.2 Reserve Invariant

Theorem 12.2 (Wrapper Reserve Invariant)**.** *For all execution traces:*

$$\forall t : R_{USDC}(t) \geq S_{x0\text{-}USD}(t) \tag{37}$$

Proof. Proven in Section 7.2. □ □

13 Performance Analysis

13.1 Computational Complexity

Operation	Time	Space
Transfer validation	$O(n)$	$O(n)$
Merkle verification	$O(\log m)$	$O(\log m)$
Bloom verification	$O(k)$	$O(1)$
Reputation update	$O(1)$	$O(1)$
Registry search	$O(N)$	$O(1)$

Table 2: Computational Complexity (n = window size, m = whitelist size, k = hash count, N = registered agents)

Account	Size (bytes)	Rent (SOL)
AgentPolicy	4,952	0.035
EscrowAccount	307	0.0030
AgentRegistry	2,064	0.015
AgentReputation	114	0.0016
WrapperConfig	243	0.0025
WrapperStats	137	0.0018
ProofContext	248	0.0026
AdminAction	120	0.0016

Table 3: Account Sizes and Rent Costs (at 0.00000348 SOL/byte-epoch)

13.2 On-Chain Costs

13.3 Transaction Benchmarks

Measured on Solana devnet (February 2026):

Operation	Compute Units	Latency
Initialize policy	45,000	450ms
Validate transfer (Merkle)	32,000	320ms
Validate transfer (Bloom)	18,000	180ms
Create escrow	28,000	280ms
Registry listing	35,000	350ms
Reputation update	15,000	150ms

Table 4: Transaction Performance (1 CU $\approx$ 10 µs)

14 Related Work

14.1 Payment Channels

Bitcoin's Lightning Network [4] and Ethereum's Raiden [5] enable off-chain micropayments. Limitations:

- Require channel opening/closing (on-chain)
- Liquidity fragmentation
- Limited programmability

x0 uses on-chain transactions but adds programmable policy enforcement unavailable in payment channels.

14.2 Escrow Protocols

Traditional escrow (e.g., Escrow.com) requires trusted third parties. Smart contract escrow (OpenBazaar, LocalBitcoins) removes intermediaries but lacks:

- Reputation integration
- Automated dispute resolution
- Cross-protocol standards

14.3 Reputation Systems

OpenBazaar and Augur implement on-chain reputation but lack:

- Temporal decay
- Sybil resistance for new participants
- Integration with payment protocols

x0's reputation oracle addresses these limitations.

15 Future Work

15.1 Extended Zero-Knowledge Applications

With the foundational ZK verification infrastructure now deployed (Section 9), several extensions become feasible:

- **Private spend limit verification**: Prove that a transfer is within policy limits without revealing the daily limit or cumulative spend, using range proofs composed with the existing Groth16 circuits
- **Proof of reputation threshold**: Prove $S \geq \theta$ for a reputation score S and threshold θ without revealing the exact score, enabling privacy-preserving agent discovery
- **Confidential escrow amounts**: Extend the escrow state machine to operate on encrypted amounts, using ZK proofs to verify state transitions without revealing the escrowed value
- **Recursive proof composition**: Batch multiple proof verifications into a single on-chain instruction using recursive SNARKs, reducing per-transaction compute costs

15.2 Cross-Chain Support

Extend to other chains via:

- Wormhole integration for cross-chain messages and proof relay
- Unified reputation across chains, anchored to Solana as the settlement layer
- Cross-chain escrow settlement with atomic swap guarantees

15.3 Machine Learning Integration

- Anomaly detection for suspicious agent behavior using on-chain transaction patterns
- Dynamic reputation weighting via learned feature importance
- Predictive escrow dispute resolution using historical outcome data

15.4 Governance

- DAO for protocol parameter updates (fee rates, timelock durations)
- Community-driven arbiter selection and staking
- Fee redistribution to token holders and active arbiters

16 Conclusion

We have presented x0, a comprehensive payment infrastructure for autonomous agents. The protocol's key innovations include:

1. **Transfer Hook Policy Enforcement**: Programmable spending limits enforced cryptographically on every transaction
2. **HTTP 402 Protocol**: Standardized payment negotiation for agent-to-agent commerce
3. **Conditional Escrow**: Trustless high-value transactions with dispute resolution
4. **Temporal Reputation**: On-chain trust scores with decay to prevent stale reputations
5. **USDC Wrapper**: Stable payments with cryptographic reserve invariants
6. **Human-in-the-Loop**: Fallback mechanism for exceptional cases
7. **Zero-Knowledge Verification**: On-chain Groth16 proof verification enabling confidential transfers with provable correctness

The protocol has been deployed on Solana devnet and is undergoing security audits. We invite the community to review the codebase at `github.com/x0-protocol` and provide feedback.

The future of AI agent commerce requires infrastructure that is:

- **Fast**: Sub-second transaction finality
- **Cheap**: Sub-cent transaction costs
- **Programmable**: Enforcing complex spending rules
- **Trustless**: No centralized intermediaries
- **Safe**: Bounded downside risk

x0 achieves these properties through careful cryptographic design and integration with Solana's high-performance blockchain.

Acknowledgments

We thank the Solana Foundation for technical support, the Anchor framework team for excellent developer tooling, and the broader Solana community for feedback during development. Special thanks to the auditors at OtterSec and Trail of Bits for security reviews.

References

[1] Nakamoto, S. (2008). Bitcoin: A Peer-to-Peer Electronic Cash System.

[2] Buterin, V. (2014). Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform.

[3] Yakovenko, A. (2018). Solana: A new architecture for a high performance blockchain.

[4] Poon, J., & Dryja, T. (2016). The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments.

[5] Raiden Network Team. (2017). Raiden Network: Fast, cheap, scalable token transfers for Ethereum.

[6] Solana Labs. (2023). Token-2022 Extension Documentation. `https://spl.solana.com/token-2022`

[7] ElGamal, T. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Transactions on Information Theory, 31(4), 469-472.

[8] Merkle, R. C. (1988). A Digital Signature Based on a Conventional Encryption Function. CRYPTO '87.

[9] Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7), 422-426.

[10] Bernstein, D. J., et al. (2012). High-speed high-security signatures. Journal of Cryptographic Engineering, 2(2), 77-89.

[11] Groth, J. (2016). On the Size of Pairing-Based Non-interactive Arguments. EUROCRYPT 2016, Lecture Notes in Computer Science, vol 9666, pp. 305-326.

[12] Hamburg, M. (2015). Decaf: Eliminating cofactors through point compression. CRYPTO 2015. Extended to Ristretto255 for use with Curve25519.

A Error Codes Reference

Error codes follow a structured numbering scheme where the high byte identifies the program category and the low nibble groups errors by subcategory:

B Program Addresses

C Mathematical Notation

D SDK Integration Guide

D.1 Installation

The x0 TypeScript SDK is available via npm:

Range	Name	Description
x0-guard: Policy Errors (0x1100–0x110F)		
0x1101	RecipientNotWhitelisted	Recipient not in whitelist
0x1102	DailyLimitExceeded	Rolling 24h limit exceeded
0x1103	InvalidMerkleProof	Merkle proof verification failed
0x1104	InvalidBloomFilter	Bloom filter misconfigured
0x1105	PolicyNotFound	Agent policy PDA missing
0x1106	UnauthorizedSigner	Unauthorized transaction signer
0x1107	ConfidentialTransferFailed	ZK proof verification failed
x0-guard: Configuration Errors (0x1110–0x111F)		
0x1110	DailyLimitTooHigh	Exceeds maximum daily limit
0x1111	DailyLimitTooLow	Below minimum daily limit
0x1112	InvalidWhitelistConfig	Invalid whitelist configuration
0x1118	DelegationRequired	Agent must be delegate, not owner
0x111B	BoundTokenAccountMismatch	Wrong source token account
0x111F	MissingZkProof	ZK proof required for confidential transfer
x0-guard: Transfer Errors (0x1120–0x112F)		
0x1120	ZeroTransferAmount	Transfer amount is zero
0x1121	TransferAmountTooHigh	Exceeds per-transaction limit
x0-guard: Blink Errors (0x1130–0x113F)		
0x1130	BlinkRateLimitExceeded	>3 blinks per hour
0x1131	BlinkExpired	Blink has expired
x0-guard: Severity Errors (0x1140–0x114F)		
0x1140	PolicyUpdateTooFrequent	Rate-limited policy update
0x1141	SingleTransactionLimitExceeded	Per-tx limit exceeded
0x1143	ExtraMetasAlreadyInitialized	Extra metas re-init blocked
0x1144	UnauthorizedExtraMetasInitializer	Mint authority required
x0-escrow: State Errors (0x1108–0x1109, 0x1200+)		
0x1108	EscrowExpired	Escrow timeout reached
0x1109	InvalidEscrowState	Invalid state transition
0x1202	SameBuyerAndSeller	Buyer and seller must differ
0x1204	ZeroEscrowAmount	Escrow amount is zero
x0-escrow: Operation Errors (0x1210–0x121F)		
0x1210	OnlyBuyerCanFund	Only buyer can fund
0x1211	OnlySellerCanDeliver	Only seller can mark delivered
0x1213	OnlyArbiterCanResolve	Only arbiter can resolve
0x1217	ArbiterResolutionTooEarly	Arbiter must wait for delay
x0-registry: Entry Errors (0x1300–0x130F)		
0x1300	AgentAlreadyRegistered	Agent already registered
0x1301	AgentNotFound	Agent not found
0x1303	EndpointTooLong	Endpoint URL too long
0x1304	TooManyCapabilities	Exceeds max capabilities
0x1307	InsufficientListingFee	Listing fee not met
x0-reputation: Errors (0x1400–0x140F)		
0x1400	ReputationNotFound	Reputation account missing
0x1401	ReputationAlreadyInitialized	Already initialized
0x1403	UnauthorizedReputationUpdate	Unauthorized caller
0x1404	InsufficientTransactions	Too few txns for score
x0-token: Token Errors (0x1500–0x1515)		
0x1500	MintAlreadyInitialized	Mint already initialized
0x1503	ConfidentialTransfersNotEnabled	Confidential not enabled
0x1507	InvalidElGamalPubkey	Invalid ElGamal public key
0x1508	InvalidPubkeyValidityProof	Invalid pubkey proof
0x150A	InvalidWithdrawProof	Invalid withdraw proof
0x150F	AmountExceedsConfidentialMax	Amount $> 2^{48} - 1$
x0-wrapper: State/Reserve Errors (0x1600–0x164F)		
0x1600	WrapperPaused	Operations are paused
0x1610	InsufficientReserve	Insufficient reserve balance
0x1611	ReserveInvariantViolated	Reserve < supply
0x1620	DepositTooSmall	Below minimum deposit
0x1623	DailyRedemptionLimitExceeded	Daily limit exceeded
0x1630	Unauthorized	Admin required
0x1640	AdminActionNotFound	Action PDA not found
0x1643	TimelockNotExpired	Timelock period pending
x0-zk-verifier: Proof Errors (0x1700–0x1721)		
0x1700	ProofVerificationFailed	ZK proof verification failed
0x1701	InvalidProofData	Invalid proof data format
0x1703	InvalidProofType	Invalid proof type
0x1704	ProofExpired	Proof timestamp too old
0x1710	AmountTooLarge	Amount $> 2^{48} - 1$
0x1711	InvalidElGamalPubkey	Invalid ElGamal public key
0x1713	ProofSizeMismatch	Proof data size mismatch
0x1720	ArithmeticOverflow	Arithmetic overflow

Table 5: Protocol Error Codes (representative subset; 7 enums, 60+ total codes). Full reference in source: `x0-common/src/error.rs`

Program	Devnet Address
x0-guard	`2uYGW3fQUGfhrwVbkupdasXBpRPfGYBGTLUdaPTXU9vP`
x0-token	`EHHTCSyGkmnsBhGsvCmLzKgcSxtsN31ScrfiwcCbjHci`
x0-escrow	`AhaDyVm8LBxpUwFdArA37LnHvNx6cNWe3KAiy8zGqhHF`
x0-registry	`Bebty49EPhFoANKDw7TqLQ2bX61ackNav5iNkj36eVJo`
x0-reputation	`FfzkTWRGAJQPDePbujZdEhKHqC1UpqvDrpv4TEiWpx6y`
x0-wrapper	`EomiXBbg94Smu4ipDoJtuguazcd1KjLFDFJt2fCabvJ8`
x0-zk-verifier	`zQWSrznKgcK8aHA4ry7xbSCdP36FqgUHj766YM3pwre`

Table 6: Deployed Program Addresses

Symbol	Meaning
H	Cryptographic hash function (SHA-256)
sk, pk	Secret key, public key
σ	Digital signature
L	Daily spending limit
x_i	Transaction amount at index i
t_i	Transaction timestamp at index i
R	Recipient address
W	Whitelist set
n	Number of transactions
m	Number of whitelist entries
k	Number of hash functions (Bloom)
S	Reputation score
ρ	Reserve ratio
R_{USDC}	USDC reserve balance
$S_{\text{x0-USD}}$	x0-USD supply
G, H	Ristretto255 generators
P	ElGamal public key ($P = s \cdot G$)
s	ElGamal secret scalar
π	Zero-knowledge proof ($\pi = (A, B, C)$ for Groth16)
Δ_{proof}	Proof freshness window (300 seconds)

Table 7: Mathematical Notation Reference

```
npm install @x0-protocol/sdk
# or
yarn add @x0-protocol/sdk
```

Listing 16: SDK Installation

D.2 Quick Start

```
import {
  X0Client,
  EscrowClient,
  ReputationClient,
  createPaymentRequest,
  verifyPaymentProof
} from '@x0-protocol/sdk';
import { Connection, Keypair } from '@solana/web3.js';

// Initialize connection
```

```typescript
const connection = new Connection('https://api.devnet.solana.com');
const wallet = Keypair.generate(); // or load from file

// Create escrow
const escrow = new EscrowClient(connection);
const { escrowPda, tx } = await escrow.buildCreateEscrowInstruction({
  buyer: wallet.publicKey,
  seller: sellerPubkey,
  amount: 1_000_000, // 1 USDC (6 decimals)
  memo: 'API service payment',
  timeoutSeconds: 86400, // 24 hours
});

// x402 payment flow (service side)
const paymentRequest = createPaymentRequest({
  recipient: servicePubkey,
  amount: '1000000',
  resource: '/api/v1/generate',
  network: 'solana-devnet',
});

// Verify payment (service side)
const isValid = await verifyPaymentProof(connection, proof, {
  expectedRecipient: servicePubkey,
  expectedAmount: 1_000_000,
  toleranceSeconds: 300,
});
```

Listing 17: Basic SDK Usage

D.3 SDK Modules

Module	Purpose
`EscrowClient`	Create, fund, release, dispute escrows
`ReputationClient`	Query and update agent reputation
`RegistryClient`	Register agents, discover services
`GuardClient`	Manage spending policies
`WrapperClient`	Wrap/unwrap USDC to x0-USD
`ConfidentialClient`	Confidential transfer proof generation via WASM
`ZkVerifierClient`	On-chain ZK proof verification
`x402`	HTTP 402 payment request/proof utilities
`blink`	Generate Solana Actions for human approval

Table 8: SDK Module Overview

D.4 Resources

- **npm**: `https://npmjs.com/package/@x0-protocol/sdk`
- **GitHub**: `https://github.com/x0-protocol/sdk`
- **API Docs**: `https://docs.x0protocol.dev/sdk`
- **Examples**: `https://github.com/x0-protocol/examples`

D.5 Confidential Transfer Architecture

For the complete treatment of the zero-knowledge proof system—including cryptographic foundations, proof types, the `ProofContext` state account, and the end-to-end verification flow—see Section 9. The SDK's `ConfidentialClient` module wraps the WASM-compiled `x0-zk-proofs` crate to provide transparent proof generation:

```
import { ConfidentialClient } from '@x0-protocol/sdk';

const confidentialClient = new ConfidentialClient(connection, wallet);

// Proof generation via WASM (x0-zk-proofs):
// - PubkeyValidityProof: Proves ElGamal key validity (64 bytes)
// - WithdrawProof: Proves withdrawal against encrypted balance (160 bytes)
// - ZeroBalanceProof: Proves zero balance for account closure (96 bytes)
// On-chain verification via x0-zk-verifier creates ProofContext PDAs
// that Token-2022 references during confidential transfer execution.
```

Listing 18: Confidential Transfer ZK Proof Generation