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, and a USDC-backed wrapper token with cryptographic reserve invariants. 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

1.3 System Overview

The x0 protocol consists of seven interoperating programs deployed on Solana:

- **x0-guard**: Policy enforcement via transfer hooks
- **x0-token**: Token-2022 mint configuration
- **x0-escrow**: Conditional payment escrow
- **x0-registry**: Agent discovery and capability advertisement
- **x0-reputation**: Trust scoring and transaction history
- **x0-wrapper**: USDC-backed stable wrapper token
- **x0-common**: Shared types, constants, and utilities

1.3.1 Architecture Diagram

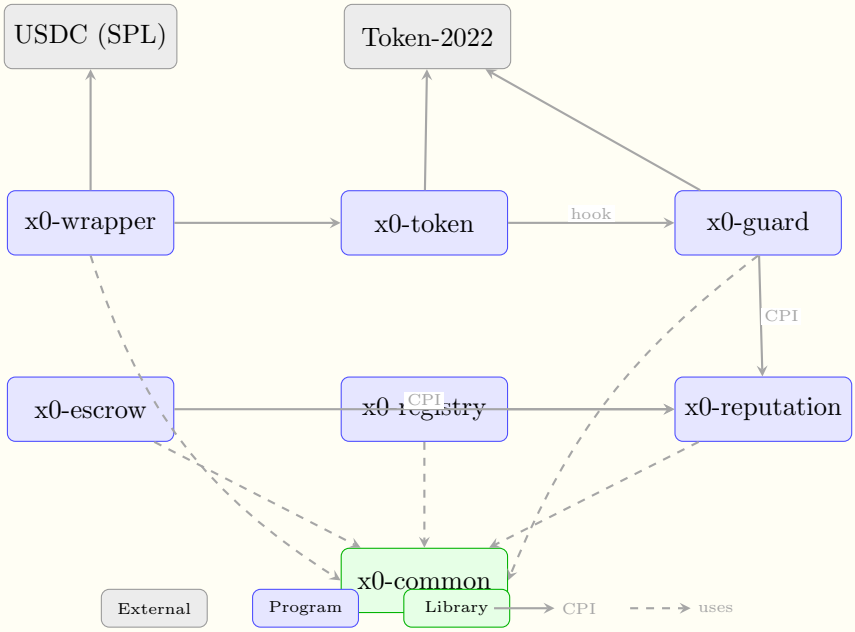

Figure 1: x0 Protocol Architecture. Gray nodes are external Solana programs. Blue nodes are x0 programs. Green is the shared library.

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:

```rust
#[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

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

3.4 Rolling Window Algorithm

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

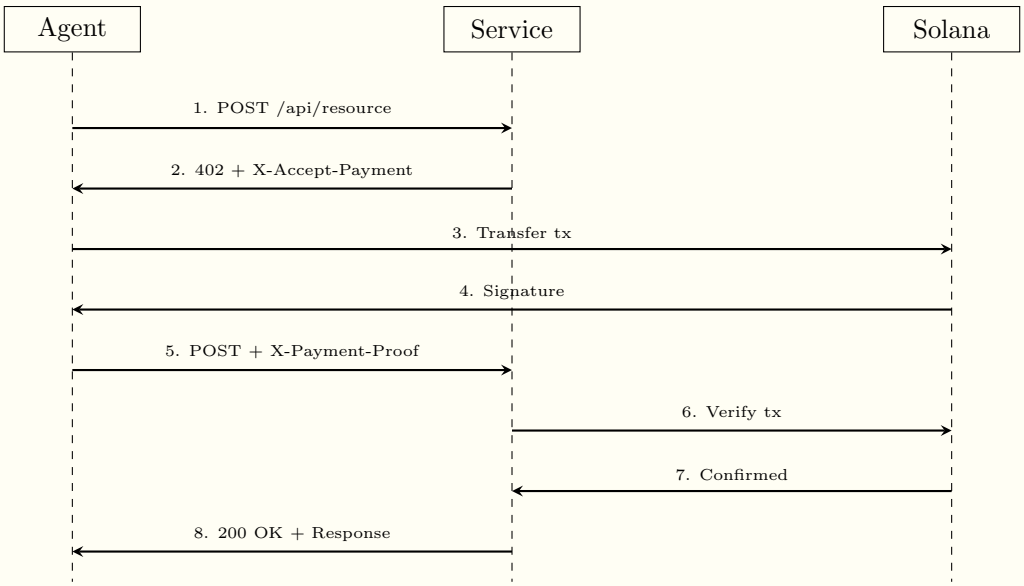

Figure 2: x402 Payment Flow. Agent receives 402, pays on-chain, then retries with proof.

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

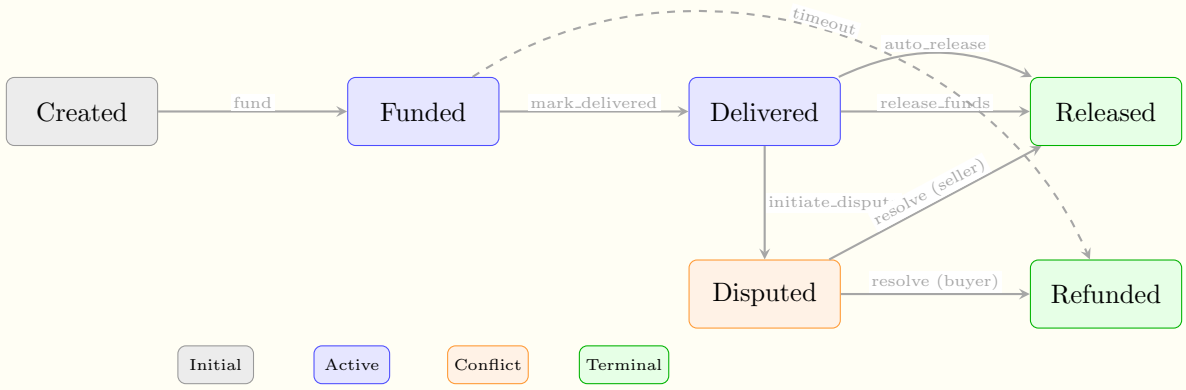

Figure 3: Escrow State Machine. States are colored by type: gray (initial), blue (active), orange (conflict), green (terminal).

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,
    pub protocol_fee_paid: u64,     // Fee paid on creation
    pub bump: u8,
}
```

Listing 7: Escrow Account Structure

5.2 Key Operations

5.2.1 Create and Fund

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
  $e \leftarrow \texttt{EscrowPDA}(B, S, H(m))$
  $e.\texttt{timeout} \leftarrow t_{\text{now}} + \tau$
  $e.\texttt{state} \leftarrow \texttt{Created}$
  return e
end procedure
procedure FundEscrow(e, x)
  require $e.\texttt{state} = \texttt{Created}$
  require $t_{\text{now}} < e.\texttt{timeout}$
  $\texttt{transfer}(B, e, x)$
  $e.\texttt{state} \leftarrow \texttt{Funded}$
end procedure

5.2.2 Delivery and Release

5.2.3 Auto-Release

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

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 $\in \{e$.buyer, e.seller$\}$
  require e.state $=$ Disputed
  e.dispute_evidence $\leftarrow H(d)$
  e.dispute_initiated_slot $\leftarrow$ current_slot
  e.state $\leftarrow$ Disputed
  UpdateReputation(e.seller, Dispute)
end procedure
procedure ResolveDispute(e, release_to_seller)
  require signer $= e$.arbiter
  require e.state $=$ Disputed
  require current_slot $\geq e$.dispute_initiated_slot $+ \Delta$
    ▷ $\Delta = 216000$ slots ≈ 24 hours
  if release_to_seller **then**
    e.state $\leftarrow$ Released
    transfer(e, e.seller, e.amount)
    UpdateReputation(e.seller, ResolutionFavor)
  else
    e.state $\leftarrow$ 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:

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

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

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

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

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 ← ⌊(t_now − r.last_decay_applied)/2592000⌋
    if m > 0 then
        d_mult ← 99^min(m,12)                          ▷ Cap at 12 months
        d_div ← 100^min(m,12)
        r.successful_transactions ← r.successful_transactions · d_mult/d_div
        r.last_decay_applied ← t_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,
    pub owner: Pubkey,
    pub endpoint: String,               // Max 256 chars
    pub capabilities: Vec<Capability>, // Max 10 capabilities
    pub metadata_uri: Option<String>,  // Extended metadata
    pub price_oracle: Option<Pubkey>,
    pub reputation_pda: Option<Pubkey>,
    pub min_transaction_amount: u64,
    pub max_transaction_amount: Option<u64>,
    pub supported_tokens: Vec<Pubkey>, // Max 5 tokens
    pub last_updated: i64,
    pub created_at: i64,
    pub total_transactions: u64,
    pub is_active: bool,
    pub is_verified: bool,              // Protocol verification
    pub bump: u8,
}

pub struct Capability {
    pub capability_type: String,        // Max 64 chars
    pub version: String,                // Max 32 chars
    pub price_lamports: u64,
    pub metadata: String,               // Max 256 chars
}
```

Listing 9: Registry Entry Structure

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:

```json
{
    "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 Reserve Invariant

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

$$R_{\text{USDC}} \geq S_{\text{x0-USD}} \tag{19}$$

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.

$$R' = R + x, \quad S' = S + x \implies R' - S' = R - S \geq 0 \tag{20}$$

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

$$R' = R - (x - f), \quad S' = S - x \tag{21}$$

$$R' - S' = R - x + f - S + x = R - S + f \geq R - S \geq 0 \tag{22}$$

The fee increases the reserve ratio. □ □

8.3 Governance

8.3.1 Timelock

All admin operations require 48-hour timelock:

Algorithm 8 Timelock Pattern

```
procedure ScheduleAction(a, v, t)
    action_pda ← CreateActionPDA(a, nonce)
    action_pda.type ← a
    action_pda.value ← v
    action_pda.scheduled_time ← t + 172800                ▷ 48h
end procedure
procedure ExecuteAction(action_pda)
    require t_now ≥ action_pda.scheduled_time
    require ¬action_pda.executed
    require ¬action_pda.cancelled
    ApplyAction(action_pda.type, action_pda.value)
    action_pda.executed ← 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)**.**

$$\rho = \frac{R_{\text{USDC}}}{S_{\text{x0-USD}}} \tag{23}$$

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

9.2.2 Rate Limiting

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

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

9.2.3 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

$$\frac{dS}{dt} = D(t) - R(t) \tag{24}$$

$$\frac{dR_{\text{USDC}}}{dt} = D(t) - R(t) + f \cdot R(t) \tag{25}$$

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:

$$\rho(t) = \frac{R_{\text{USDC}}(t)}{S(t)} = 1 + \int_0^t \frac{f \cdot R(\tau)}{S(\tau)} d\tau \tag{26}$$

The reserve ratio increases over time due to fees.

11 Security Analysis

11.1 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

11.2 Attack Scenarios and Mitigations

11.2.1 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

11.2.2 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).

11.2.3 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

11.2.4 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

11.3 Formal Verification

11.3.1 Spend Limit Invariant

Theorem 11.1 (Daily Spend Limit)**.** *For all execution traces, if policy has daily limit L:*

$$\forall t : \sum_{i:t_i > t-86400} x_i \leq L \tag{27}$$

Proof. Proven in Section 3.4. □ □

11.3.2 Reserve Invariant

Theorem 11.2 (Wrapper Reserve Invariant)**.** *For all execution traces:*

$$\forall t : R_{USDC}(t) \geq S_{x0\text{-}USD}(t) \tag{28}$$

Proof. Proven in Section 7.2. □ □

12 Performance Analysis

12.1 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)

12.2 On-Chain Costs

12.3 Transaction Benchmarks

Measured on Solana devnet (February 2026):

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

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

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)

13 Related Work

13.1 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.

13.2 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

13.3 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.

14 Future Work

14.1 Zero-Knowledge Proofs

Future versions **may** use ZK-SNARKs for:

- Private spend limit verification
- Proof of reputation without revealing details
- Confidential escrow amounts

14.2 Cross-Chain Support

Extend to other chains via:

- Hyperlane integration for cross-chain x0-USDC operations
- Unified reputation across chains
- Cross-chain escrow settlement

14.3 Machine Learning Integration

- Anomaly detection for suspicious agent behavior
- Dynamic reputation weighting
- Predictive escrow dispute resolution

14.4 Governance

- DAO for protocol parameter updates
- Community-driven arbiter selection
- Fee redistribution to token holders

15 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

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

—

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.

A Error Codes Reference

B Program Addresses

C Mathematical Notation

D SDK Integration Guide

D.1 Installation

The x0 TypeScript SDK is available via npm:

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

Listing 11: 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
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 12: Basic SDK Usage

D.3 SDK Modules

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`

Code	Name	Description
x0-common errors (0x1770+)		
0x1770	Unauthorized	Unauthorized access
0x1771	InvalidAmount	Invalid amount specified
0x1772	InvalidTimestamp	Invalid timestamp
0x1773	AccountNotInitialized	Account not initialized
0x1774	InvalidProgramId	Invalid program ID
0x1775	MathOverflow	Math overflow error
0x1776	InvalidMint	Invalid token mint
0x1777	InsufficientFunds	Insufficient funds
0x1778	InvalidState	Invalid state transition
0x1779	Expired	Operation has expired
0x177A	NotExpired	Not yet expired
0x177B	AlreadyInitialized	Already initialized
0x177C	InvalidOwner	Invalid owner
0x177D	InvalidSigner	Invalid signer
0x177E	InvalidAccount	Invalid account
0x177F	InvalidInstruction	Invalid instruction
0x1780	RateLimitExceeded	Rate limit exceeded
0x1781	InvalidProof	Invalid cryptographic proof
0x1782	InvalidHash	Invalid hash value
0x1783	InvalidSignature	Invalid signature
0x1784	InvalidPublicKey	Invalid public key
0x1785	InvalidSeed	Invalid PDA seed
0x1786	InvalidBump	Invalid PDA bump
0x1787	InvalidData	Invalid account data
0x1788	InvalidLength	Invalid data length
0x1789	InvalidVersion	Invalid account version
Guard-specific errors		
0x178A	PolicyNotActive	Policy is deactivated
0x178B	DailyLimitExceeded	Transfer exceeds 24h limit
0x178C	SingleTxLimitExceeded	Exceeds per-tx limit
0x178D	DestinationNotWhitelisted	Recipient not in whitelist
0x178E	WindowOverflow	Rolling window overflow
Escrow-specific errors		
0x178F	InvalidEscrowState	Invalid state transition
0x1790	DisputeDelayNotMet	24h dispute delay not met
0x1791	ArbiterRequired	Arbiter required for dispute
Wrapper-specific errors		
0x1792	WrapperPaused	Wrapper operations paused
0x1793	InsufficientReserve	Reserve undercollateralized
0x1794	TimelockNotExpired	Timelock period not expired
0x1795	ActionAlreadyExecuted	Timelock action already executed
0x1796	ActionCancelled	Timelock action was cancelled

Table 5: Protocol Error Codes (Anchor offset: 6000 + index)

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

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

Table 7: Mathematical Notation Reference

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
`x402`	HTTP 402 payment request/proof utilities
`blink`	Generate Solana Actions for human approval

Table 8: SDK Module Overview