REINEIRA LABS LTD.

ReineiraOS

Programmable Infrastructure for Stablecoins: Confidential Settlement and Cross-Chain Orchestration via Fully Homomorphic Encryption

Version 0.1 — March 2026

Reineira Protocol Team
Reineira Labs Ltd.

TECHNICAL WHITEPAPER

Abstract

Stablecoins facilitate over $10 trillion in annual transaction volume, yet the infrastructure supporting them remains limited to transparent, irrevocable transfers with no native support for conditional settlement, privacy, or automated verification. This paper presents ReineiraOS, a programmable infrastructure layer for stablecoins that addresses these gaps through six integrated components: (1) a confidential token wrapper that converts any ERC-20 stablecoin into an FHE-encrypted equivalent with homomorphic arithmetic over ciphertext, with USDC as the reference implementation (cUSDC); (2) an escrow engine where all sensitive state — owner addresses, amounts, payment status — is stored as TFHE ciphertexts, with a novel silent failure pattern that prevents information leakage from unauthorized redemption attempts; (3) a decentralized operator network with stake-weighted task routing, tiered execution windows, and optimistic dispute resolution; (4) a pluggable cross-chain settlement layer with a bridge-agnostic architecture, currently implemented with Circle CCTP V2 and hook data encoding for automatic escrow funding; (5) a verification-agnostic condition resolution framework that supports any proof system — including zkTLS, oracle attestations, multisig approvals, and time-locks — through a pluggable interface; and (6) a plugin ecosystem for compliance, automation, and payment protection. The architecture is designed to be stablecoin-agnostic and bridge-agnostic: any ERC-20 token can be wrapped into a confidential counterpart, and any cross-chain bridge can be integrated through the handler plugin interface. The protocol is deployed on Arbitrum Sepolia with USDC and CCTP V2 as the initial integration. We present the architecture, formal security analysis, and implementation details, demonstrating that practical confidential stablecoin infrastructure is achievable with gas overhead of approximately $0.05–$0.50 per operation on Arbitrum L2.

Contents

1. Introduction

1.1 Motivation

Stablecoins have emerged as one of the most consequential innovations in digital finance, facilitating over $10 trillion in annual transaction volume as of 2025 [1]. USDC alone accounts for a significant portion of this volume, serving as a programmable dollar primitive across dozens of blockchain networks. Yet the infrastructure supporting stablecoin transactions remains fundamentally limited by the transparency model of public blockchains.

Every USDC transfer on Ethereum, Arbitrum, or any EVM-compatible chain is fully visible to all network participants. Transaction amounts, sender addresses, recipient addresses, and timing information are permanently recorded in an immutable public ledger. While this transparency serves important functions for auditability and trustlessness, it creates untenable conditions for commercial adoption. Businesses cannot conduct payroll operations, vendor payments, or revenue settlements on-chain without exposing commercially sensitive information to competitors, counterparties, and the general public.

The problem extends beyond mere privacy. The absence of programmable escrow infrastructure means that stablecoin payments lack the conditional settlement, dispute resolution, and automated verification capabilities that traditional payment systems provide. Cross-chain stablecoin transfers remain fragmented across bridge protocols with varying trust assumptions, and there exists no unified framework for coordinating operators who facilitate these transfers.

ReineiraOS is motivated by the conviction that stablecoins will only achieve their potential as a global settlement layer when the infrastructure supporting them provides confidentiality, programmability, and cross-chain composability as first-class primitives. The protocol addresses these requirements through a vertically integrated stack that combines Fully Homomorphic Encryption, decentralized operator coordination, and cross-chain settlement into a coherent programmable infrastructure.

1.2 Problem Statement

We identify five critical deficiencies in current stablecoin infrastructure:

P1: Transactional Transparency. All ERC-20 token transfers expose the sender address, recipient address, and transfer amount on-chain. This precludes enterprise adoption for payroll, vendor payments, inter-company settlements, and any transaction where amount or counterparty confi-

dentiality is required. Existing privacy solutions (mixers, zero-knowledge proofs) either compromise compliance or introduce unacceptable latency.

P2: Absence of Programmable Escrow. Stablecoin payments are irrevocable upon execution. There exists no native mechanism for conditional settlement, payment verification, or dispute-mediated release. Applications requiring escrow must build bespoke smart contract solutions that re-expose all transaction details through plaintext state variables.

P3: Cross-Chain Fragmentation. USDC exists natively on multiple chains, and Circle's Cross-Chain Transfer Protocol (CCTP) enables native burn-and-mint bridging. However, there is no infrastructure layer that combines cross-chain transfers with confidential settlement, escrow awareness, or automated operator coordination.

P4: Operator Incentive Misalignment. Cross-chain relay, attestation fetching, and transaction execution require off-chain operators. Existing relay networks lack formal staking requirements, dispute resolution mechanisms, and fee distribution models that align operator incentives with protocol liveness and correctness.

P5: Compliance-Privacy Tension. Privacy-preserving payment systems have historically existed in tension with regulatory compliance requirements. There is a need for infrastructure that provides transactional confidentiality while supporting pluggable compliance modules that can enforce jurisdiction-specific requirements without compromising the core privacy guarantees.

1.3 Contributions

This paper presents the following contributions:

1. **Confidential Token Protocol.** We design a stablecoin-agnostic confidential token wrapper based on the ERC-7984 standard. Any ERC-20 stablecoin can be wrapped into an FHE-encrypted equivalent using Fhenix CoFHE. The wrapper maintains a configurable conversion rate, supports encrypted transfers, encrypted balances, and permissioned decryption. The reference implementation wraps USDC into cUSDC with a 1:1 peg at 6-decimal precision.
2. **FHE-Encrypted Escrow Engine.** We present a novel escrow architecture where all state variables—owner addresses, payment amounts, paid amounts, and redemption status—are stored as FHE ciphertexts. The engine performs payment verification through homomorphic comparison operations and implements a silent failure pattern that prevents information leakage from unauthorized redemption attempts.
3. **Decentralized Orchestration Network.** We design a stake-weighted operator network with formal task routing, exclusive execution windows, permissionless fallback, and optimistic dispute resolution. The network coordinates cross-chain relay operations with configurable fee distribution.
4. **Cross-Chain Settlement Protocol.** We design a bridge-agnostic cross-chain settlement layer that supports any bridge protocol through a handler plugin interface. The current implemen-

tation integrates Circle CCTP V2 with hook data encoding for automatic escrow funding. The settlement flow wraps received stablecoins into their confidential counterparts and funds the target escrow using balance-based verification.

5. **Verification-Agnostic Condition Resolution and Payment Protection.** We present a pluggable verification framework where any proof system — zkTLS, oracle attestations, multisig approvals, time-locks, or FHE computations — can serve as a condition resolver for escrow operations. We combine this with a two-layer payment protection model using cryptographic verification and economic insurance.
6. **Plugin Ecosystem.** We describe an extensible plugin architecture supporting compliance modules, condition resolvers, and automation agents that can be composed with the core protocol without modifying its trust assumptions.

---

2. Background

2.1 Stablecoin Infrastructure

Stablecoins are blockchain-native tokens that maintain a stable value relative to a reference asset, typically the US dollar. They exist in three primary categories: fiat-collateralized (USDC, USDT), crypto-collateralized (DAI), and algorithmic (FRAX). As of 2025, the total stablecoin market capitalization exceeds $200 billion, with USDC representing approximately $45 billion [1].

USDC, issued by Circle, is the most widely adopted fiat-collateralized stablecoin with native deployment across Ethereum, Arbitrum, Avalanche, Solana, Polygon, Base, and numerous other networks. Each USDC token is backed 1:1 by US dollar reserves held in regulated financial institutions and is subject to monthly attestation by independent accounting firms.

The ERC-20 standard that governs USDC and most stablecoins defines a minimal interface for fungible token transfers:

This interface inherently exposes all balances and transfer amounts as plaintext uint256 values stored in contract storage. Any party with access to the blockchain state—which, by design, is every network participant—can observe these values. This transparency, while essential for the trustless verification properties of public blockchains, creates a fundamental barrier to commercial adoption.

The stablecoin infrastructure stack can be decomposed into four layers: (1) the token layer, which defines the unit of value; (2) the transfer layer, which handles movement of value between addresses; (3) the settlement layer, which provides finality and programmable conditions for value release; and (4) the coordination layer, which manages off-chain operations required for cross-chain transfers and external verification. ReineiraOS addresses deficiencies at each of these layers.

2.2 Fully Homomorphic Encryption

Before the formalism: the practical intuition behind FHE is straightforward. Imagine a spreadsheet where every cell is locked. You can still run SUM, AVERAGE, and IF formulas on the locked cells, and the result is itself a locked cell. Only someone with the master key can unlock any cell to see the actual number. FHE does exactly this for on-chain data – smart contracts perform arithmetic and comparisons on encrypted values and produce encrypted results, without the contract (or anyone observing the chain) ever seeing the plaintext.

Formally, FHE is a class of encryption schemes that permit arbitrary computation over ciphertexts without access to the decryption key [2]. Given a public key pk, a secret key sk, and an encryption function Enc_{pk}, an FHE scheme satisfies:

$$Enc_{pk}(f(m_1, m_2, ..., m_n)) = Eval(f, Enc_{pk}(m_1), Enc_{pk}(m_2), ..., Enc_{pk}(m_n))$$

for any computable function f and messages $m_1, ..., m_n$, where $Eval$ is a public evaluation function that operates solely on ciphertexts.

The practical implications for blockchain applications are profound. FHE enables smart contracts to operate over encrypted data—performing additions, comparisons, and conditional selections on ciphertexts—without ever revealing the underlying plaintext values to the blockchain state, validators, or observers. This stands in contrast to zero-knowledge proof systems, which prove statements about encrypted data but do not permit ongoing computation over that data within the contract execution environment.

FHE schemes have evolved through four generations since Gentry's seminal construction in 2009 [2]:

1. **First generation** (Gentry, 2009): Ideal lattice-based, impractical due to massive key sizes and computation times.
2. **Second generation** (BGV [3], BFV [4]): Introduced batching and modulus switching, reducing ciphertext sizes by orders of magnitude.
3. **Third generation** (GSW [5]): Approximate eigenvector method, enabling more efficient bootstrapping.
4. **Fourth generation** (TFHE [6], CKKS [7]): Gate-by-gate bootstrapping (TFHE) and approximate arithmetic (CKKS), enabling practical applications.

ReineiraOS employs the TFHE scheme via the Fhenix CoFHE framework, which is optimized for blockchain execution environments and supports the encrypted types required for financial applications.

2.3 TFHE Scheme

TFHE (Torus Fully Homomorphic Encryption), introduced by Chillotti et al. [6], operates over the real torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ and is distinguished by its ability to perform bootstrapping in under 10

milliseconds on commodity hardware, making it the most practical FHE scheme for interactive applications.

The TFHE scheme supports the following encrypted types relevant to ReineiraOS:

Encrypted Type	Plaintext Equivalent	Bit Width	Primary Use
euint8	uint8	8 bits	Flags, small counters
euint16	uint16	16 bits	Compact identifiers
euint32	uint32	32 bits	Timestamps, medium values
euint64	uint64	64 bits	Token amounts (6-decimal USDC)
eaddress	address	160 bits	Encrypted Ethereum addresses
ebool	bool	1 bit	Encrypted boolean flags

Key homomorphic operations used in ReineiraOS include:

- **Addition** (FHE.add): $Enc(a) + Enc(b) = Enc(a + b)$, used for accumulating escrow payments.
- **Subtraction** (FHE.sub): $Enc(a) - Enc(b) = Enc(a - b)$, used for balance-based verification.
- **Comparison** (FHE.gte, FHE.eq): $Enc(a) \geq Enc(b) = Enc(a \geq b)$, used for payment sufficiency checks and address equality verification.
- **Conditional selection** (FHE.select): $select(Enc(c), Enc(a), Enc(b)) = Enc(c\ ?\ a : b)$, used for branchless conditional logic that prevents timing-based information leakage.
- **Boolean logic** (FHE.and, FHE.or, FHE.not): Encrypted boolean operations for composing multiple conditions.

The Fhenix CoFHE implementation provides these operations as Solidity precompiles, enabling smart contracts to perform FHE computations within the EVM execution environment. The CoFHE model delegates computationally intensive FHE operations to a co-processor network while maintaining cryptographic integrity through attestation and verification protocols.

2.4 ERC-7984 Confidential Token Standard

ERC-7984 defines a standard interface for confidential tokens that extends the ERC-20 model with encrypted balances and transfers [8]. The standard introduces two key abstractions:

1. **Confidential balances**: Token balances are stored as FHE ciphertexts (euint64) rather than plaintext uint256 values. The confidentialBalanceOf function returns an encrypted balance handle that can only be decrypted by the authorized owner.
2. **Confidential transfers**: The confidentialTransfer and confidentialTransferFrom functions accept encrypted amounts as input, enabling transfers where the amount is never exposed on-chain.

The FHERC20 base contract, which ConfidentialUSDC extends, implements the following core interface:

The FHERC20Wrapper extension adds wrapping and unwrapping capabilities for converting between plaintext ERC-20 tokens and their confidential counterparts. This wrapping model is analogous to the WETH/ETH relationship but with the addition of FHE encryption at the wrapping boundary.

2.5 Cross-Chain Transfer Protocol V2

Circle's Cross-Chain Transfer Protocol Version 2 (CCTP V2) enables native USDC transfers between supported blockchain networks through a burn-and-mint mechanism [9]. Unlike bridge protocols that lock tokens on the source chain, CCTP V2 burns USDC on the source chain and mints an equivalent amount on the destination chain, maintaining the canonical status of USDC across all supported networks.

The CCTP V2 message structure comprises:

Byte Offset	Length	Field	Description
0–215	216 bytes	Header	Version, source domain, destination domain, nonce, sender, recipient, destination caller
216–247	32 bytes	Amount	Transfer amount as uint256
248–279	32 bytes	Mint Recipient	Recipient address on destination chain (as bytes32)
280–311	32 bytes	Fee Payer	Address paying for destination execution
312–343	32 bytes	Max Fee	Maximum fee for destination execution
344–375	32 bytes	Finality	Finality threshold and additional fields
376+	Variable	Hook Data	Application-specific data appended to the message

CCTP V2 introduces the concept of *hook data*, which allows application-specific data to be appended to the burn message. ReineiraOS leverages hook data to encode escrow identifiers, enabling the destination chain settlement contract to automatically route received USDC to the correct escrow.

The attestation model requires an attestation from Circle's off-chain attestation service before the message can be received on the destination chain. The MessageTransmitterV2 contract on the destination chain verifies the attestation signature before minting USDC to the specified recipient.

2.6 zkTLS

Zero-Knowledge Transport Layer Security (zkTLS) is an emerging protocol that enables cryptographic proofs about data transmitted over TLS connections without revealing the underlying data [10]. The protocol operates by decomposing the TLS session into a multi-party computation (MPC) between the client and a notary, where:

1. The client and notary jointly establish a TLS session with the target server.
2. The server's response is committed to by the notary without the notary learning the response content.
3. The client generates a zero-knowledge proof that specific assertions about the response data are true.

This construction enables verifiable claims about Web2 data sources—such as bank balances, payment confirmations, identity verifications, and API responses—to be brought on-chain without exposing the underlying data. In the context of ReineiraOS, zkTLS serves as a bridge between traditional financial infrastructure and on-chain settlement, enabling:

- Verification that a fiat payment has been confirmed by a bank before releasing escrowed funds.
- Proof of identity from identity verification services without exposing personal data.
- Attestation of external conditions (delivery confirmations, service completions) from Web2 APIs.

The security model of zkTLS relies on the assumption that the notary does not collude with the client to forge proofs, and that the TLS connection to the target server is authentic. These assumptions are discussed in detail in Section 12.

---

3. System Architecture

3.1 Design Principles

The ReineiraOS architecture is governed by five design principles:

Principle 1: Confidentiality by Default. All token balances, transfer amounts, escrow amounts, payment amounts, and owner addresses are encrypted using FHE. Plaintext values exist only at the system boundary—during wrapping (stablecoin to confidential token) and unwrapping (confidential token to stablecoin)—and are never stored in contract state.

Principle 2: Composability with Existing Infrastructure. ReineiraOS does not require modifications to underlying stablecoins, bridge protocols, or the destination chain. All components are deployed as standard smart contracts that interact with existing infrastructure through canonical interfaces. The confidential token wrapper is stablecoin-agnostic—any ERC-20 token can be wrapped—and the cross-chain settlement layer is bridge-agnostic, supporting any bridge through the handler plugin interface. The reference implementation integrates USDC and Circle CCTP V2. Confidential tokens maintain ERC-7984 compliance, ensuring interoperability with FHE-aware applications.

Principle 3: Separation of Concerns. The protocol stack is decomposed into independent, upgradeable components: confidential tokens, escrow engine, orchestration network, and cross-chain settlement. Each component can be upgraded independently via UUPS proxy patterns without affecting the others.

Principle 4: Economic Security. The orchestration network is secured through staking requirements, slashing conditions, and fee incentives that align operator behavior with protocol correctness and liveness. The dispute resolution mechanism uses optimistic assumptions with bonded challenges to minimize governance overhead.

Principle 5: Extensibility through Plugins. Core protocol behavior can be extended through external plugin contracts—condition resolvers, compliance modules, automation agents—without modifying the core contracts. This enables jurisdiction-specific compliance, custom settlement conditions, and automated workflows.

3.2 Protocol Stack

The ReineiraOS protocol stack is organized into five layers, each building upon the capabilities of the layer below:

Layer 1 (Base Infrastructure) comprises the foundational services: Arbitrum L2 for EVM execution, Fhenix CoFHE for FHE operations, Circle USDC as the base stablecoin, and CCTP V2 MessageTransmitter for cross-chain message verification.

Layer 2 (Confidential Token Layer) provides the encrypted token primitive. ConfidentialUSDC wraps USDC into cUSDC with FHE encryption, FHERC20Wrapper handles the wrapping/unwrapping boundary, and the FHE Permission System governs access to encrypted values.

Layer 3 (Settlement Layer) implements programmable escrow. ConfidentialEscrow manages encrypted escrow state, CCTPV2EscrowReceiver handles cross-chain settlement, ConditionResolver enables conditional redemption, and the Unwrap Extension allows direct redemption to USDC.

Layer 4 (Orchestration Layer) coordinates off-chain operations. OperatorRegistry manages operator staking and authorization, TaskExecutor routes tasks to registered handlers, FeeManager distributes fees, and SlashingManager enforces operator accountability.

Layer 5 (Application Layer) provides user-facing capabilities. Agentic Automation enables programmable workflows, Compliance Plugins enforce regulatory requirements, Payment Protection combines zkTLS verification with economic insurance, and the zkTLS Bridge connects Web2 verification to on-chain settlement.

3.3 Component Overview

The following diagram illustrates the contract-level interactions between ReineiraOS components:

The component architecture reflects a clear separation between the confidential data plane (tokens and escrow) and the coordination plane (orchestration network). The two planes intersect at the CCTPHandler and CCTPV2EscrowReceiver, which bridge cross-chain operations into confidential settlement.

3.4 Data Flow

The following sequence diagram illustrates the end-to-end data flow for a cross-chain confidential payment:

The data flow demonstrates several key properties: (1) plaintext USDC amounts are visible only during the cross-chain transfer phase; (2) the wrapping boundary at ConfidentialUSDC is the last point where amounts are in plaintext; (3) all subsequent operations on the escrow are performed over ciphertexts; and (4) the operator never gains access to decrypted escrow state.

---

4. Confidential Token Protocol

4.1 Token Design

The confidential token layer of ReineiraOS is stablecoin-agnostic: any ERC-20 token can be wrapped into an FHE-encrypted equivalent using the FHERC20Wrapper base contract from the Fhenix confidential contracts library. Each wrapper inherits encrypted balance management, confidential transfers, and the wrapping/unwrapping lifecycle while maintaining ERC-7984 compliance.

The reference implementation, ConfidentialUSDC (cUSDC), wraps Circle USDC. Additional wrappers can be deployed for other stablecoins (USDT, DAI, EURC, etc.) by instantiating new wrapper contracts pointing to the underlying token. The escrow engine is parameterized by a payment token address, allowing it to operate with any confidential wrapper.

Key design parameters for the reference cUSDC wrapper:

Parameter	Value	Rationale
Name	Reineira Confidential USDC	Brand identification
Symbol	cUSDC	“confidential USDC” denomination
Decimals	6	Matches USDC decimal precision
Conversion Rate	1:1	No scaling factor needed
Max Amount	$2^{64} - 1$ (~18.4 quintillion)	FHE-encrypted 64-bit upper bound
Underlying Token	Circle USDC	Canonical stablecoin backing

The 1:1 conversion rate is a direct consequence of both USDC and cUSDC using 6-decimal precision. This eliminates rounding errors at the wrapping boundary and simplifies the mental model for users: 1,000,000 units of USDC (representing $1.00) wraps to exactly 1,000,000 units of cUSDC. Other wrappers may use different conversion rates depending on the underlying token’s decimal configuration.

The euint64 type constrains the maximum representable amount to $2^{64} - 1 = 18,446,744,073,709,551,615$ base units, which at 6 decimals represents approximately 18.4 trillion. This upper bound is sufficient for any practical stablecoin amount and matches the uint64 casting performed during the wrapping operation.

4.2 Wrapping and Unwrapping

The wrapping process converts plaintext USDC into encrypted cUSDC:

The wrapping flow proceeds as follows:

1. The caller approves the ConfidentialUSDC contract to spend their USDC.
2. The caller invokes wrap(recipient, amount) with the desired amount.
3. The contract transfers USDC from the caller to itself via transferFrom.
4. The contract encrypts the amount using FHE.asEuint64(amount) and adds it to the recipient’s encrypted balance.
5. The USDC is held in custody by the ConfidentialUSDC contract.

The unwrapping process is asynchronous due to the decrypt-claim pattern required by FHE:

1. The caller invokes unwrap(to, amount) with an encrypted amount.
2. The contract initiates a decryption request for the encrypted amount.
3. A decrypt claim is created that, once the decryption is finalized by the CoFHE network, allows the recipient to claim the corresponding USDC.
4. The recipient calls the claim function to receive their USDC.

This asynchronous pattern is necessitated by the FHE decryption process, which requires coordination with the Fhenix CoFHE co-processor network and cannot complete within a single transaction.

4.3 Encrypted Transfers

cUSDC supports two forms of encrypted transfers:

Direct Confidential Transfer: The confidentialTransfer(address to, euint64 amount) function transfers an encrypted amount from the caller to the specified recipient. The amount is never decrypted during the transfer—the contract performs encrypted subtraction from the sender's balance and encrypted addition to the recipient's balance.

Delegated Confidential Transfer: The confidentialTransferFrom(address from, address to, euint64 amount) function enables transfers on behalf of another party, subject to operator approval. The escrow engine uses this mechanism to fund escrows: the caller (e.g., CCTPV2EscrowReceiver) transfers cUSDC from itself to the escrow contract using an encrypted amount.

Both transfer variants operate entirely in the encrypted domain. The FHE permission system (Section 4.4) governs which contracts and addresses may interact with specific ciphertext handles.

4.4 Permission Model

The Fhenix FHE framework implements a fine-grained permission system that controls access to encrypted values. The system distinguishes between two types of permissions:

Persistent Permissions (FHE.allow): Grant a specified address the ability to use an encrypted value handle in future transactions. These permissions persist across transactions and are stored in the FHE access control mapping.

Transient Permissions (FHE.allowTransient): Grant a specified address the ability to use an encrypted value handle within the current transaction only. These permissions are automatically revoked at the end of the transaction and are used for intermediate operations such as transfers and comparisons.

Contract Self-Permissions (FHE.allowThis): A convenience function that grants the current contract persistent access to an encrypted value. This is used extensively in the escrow engine to ensure the contract can access its own encrypted state variables in future transactions.

The permission model is critical for security. When an escrow is created, the contract grants itself (FHE.allowThis) and the creator (FHE.allow(value, sender)) access to the encrypted owner, amount, paid amount, and redemption status. Without explicit permissions, no party—including the contract itself—can read or operate on encrypted values.

The FHERC20 token supports delegation of transfer authority with a configurable expiry timestamp. The CCTPV2EscrowReceiver uses this mechanism to pre-authorize the escrow contract as an operator with indefinite expiry, enabling the receiver to fund escrows on behalf of cross-chain senders.

---

5. Escrow Engine

5.1 Escrow Data Structure

The ConfidentialEscrow contract stores each escrow as a struct with all sensitive fields encrypted:

The design choices for each field merit examination:

- **eaddress owner**: The escrow owner's address is encrypted, preventing external observers from determining who will receive the escrowed funds. This is essential for preventing front-running and counterparty identification.
- **euint64 amount**: The expected payment amount is encrypted, preventing observers from inferring the value of the underlying transaction. Combined with the encrypted owner, this ensures that neither the transaction value nor the recipient is publicly known.
- **euint64 paidAmount**: The cumulative payment amount is encrypted and updated homomorphically when payments are received. This prevents observers from tracking partial payment progress.
- **ebool isRedeemed**: The redemption status is encrypted to prevent observers from distinguishing between redeemed and unredeemed escrows, which could leak information about transaction completion.
- **bool exists**: The existence flag is the only plaintext field. It is intentionally left unencrypted because the existence of an escrow is already observable through the EscrowCreated event emission. Encrypting this flag would add computational overhead without providing additional privacy.

The escrow is indexed by a monotonically increasing uint256 identifier (_nextId), which is publicly visible. The sequential nature of escrow IDs reveals the total number of escrows created but does not leak information about any individual escrow's contents.

5.2 Escrow Lifecycle

The following state diagram describes the complete lifecycle of an escrow:

It is important to note that the state transitions from "FullyFunded" to "Redeemed" are determined entirely through homomorphic computation. The contract never explicitly checks whether paidAmount >= amount in plaintext; instead, it computes FHE.gte(escrow.paidAmount, escrow.amount) which returns an encrypted boolean. The actual state of the escrow—whether it is partially funded, fully funded, or redeemed—is known only to authorized parties who can decrypt the relevant ciphertext handles.

5.3 Creation Flow

Escrow creation is initiated by a caller who provides encrypted parameters:

The InEaddress and InEuint64 types are input ciphertext types that are encrypted client-side and transmitted to the contract. The creation flow proceeds as follows:

1. The caller encrypts the owner address and expected amount client-side using the network's FHE public key.
2. The caller submits the encrypted values along with an optional condition resolver address.
3. The contract converts the input ciphertexts to internal FHE handles using FHEMeta.asEaddress and FHEMeta.asEuint64, which verify the ciphertext integrity and associate them with the caller's address.
4. A new escrow ID is assigned from the monotonic counter.
5. The escrow struct is initialized with the encrypted owner, encrypted amount, zero encrypted paid amount (FHE.asEuint64(0)), and false encrypted redemption status (FHE.asEbool(false)).
6. If a condition resolver is specified, it is associated with the escrow via _setCondition.
7. The contract grants itself (FHE.allowThis) and the creator (FHE.allow(…, sender)) access to all encrypted fields.
8. An EscrowCreated(escrowId) event is emitted.

The event emission reveals only the escrow ID, not the owner, amount, or any other sensitive parameter. This is a deliberate design choice: the creation event enables indexing and monitoring of escrow activity without compromising the confidentiality of the escrow contents.

5.4 Payment Flow

The escrow engine supports two payment mechanisms:

External Funding (fund): The payer provides an encrypted payment amount as input ciphertext:

The contract performs balance-based verification to determine the actual payment received:

1. Records the contract's cUSDC balance before the transfer: balanceBefore = paymentToken.confidentialBalanceOf(address(this)).
2. Performs the encrypted transfer: paymentToken.confidentialTransferFrom(sender, address(this), paymentAmount).
3. Records the contract's cUSDC balance after the transfer: balanceAfter = paymentToken.confidentialBalanceOf(address(this)).
4. Computes the actual payment: actualPayment = FHE.sub(balanceAfter, balanceBefore).
5. Updates the cumulative paid amount: newPaidAmount = FHE.add(escrow.paidAmount, actualPayment).

This balance-based verification pattern is critical for security. It prevents scenarios where the encrypted transfer amount differs from the actual amount transferred (e.g., due to fee-on-transfer tokens or other edge cases). By measuring the actual change in the contract's encrypted balance, the escrow engine ensures that the recorded payment amount accurately reflects the received value.

Internal Funding (fundFrom): Used by the CCTPV2EscrowReceiver to fund escrows with already-encrypted amounts:

This variant accepts a pre-computed euint64 handle rather than input ciphertext, enabling programmatic funding from other contracts that already hold encrypted values.

5.5 Redemption Flow

The redemption mechanism is the most cryptographically sophisticated component of the escrow engine. When a caller attempts to redeem an escrow, the contract performs all authorization checks homomorphically:

Three conditions must be satisfied for successful redemption:

1. **Ownership verification**: FHE.eq(escrow.owner, callerEncrypted) — the caller's address must match the encrypted owner address.
2. **Payment sufficiency**: FHE.gte(escrow.paidAmount, escrow.amount) — the cumulative paid amount must be greater than or equal to the expected amount.
3. **Non-duplication**: FHE.not(escrow.isRedeemed) — the escrow must not have been previously redeemed.

The redemption amount is then computed conditionally:

If all conditions are met, redemptionAmount equals the paid amount; otherwise, it equals zero. The transfer is then executed regardless:

This unconditional transfer of a conditionally-computed amount is the foundation of the silent failure pattern, discussed in Section 5.6.

Two redemption variants are provided:

- **redeem(escrowId)**: Redeems a single escrow and transfers cUSDC to the caller.
- **redeemAndUnwrap(escrowId, recipient)**: Redeems and converts cUSDC back to USDC, sending it to the specified recipient address (which may differ from the caller).

5.6 Silent Failure Pattern

The silent failure pattern is a deliberate design choice that prevents information leakage from failed redemption attempts. In a traditional smart contract, a failed authorization check would result in a revert with an error message, revealing to the caller (and any observer) that the conditions were not met. This creates an oracle attack vector:

1. An adversary attempts to redeem an escrow they do not own.
2. If the transaction reverts, the adversary learns that the escrow exists and has a different owner.
3. If the transaction succeeds with zero transfer, the adversary learns nothing.

In the traditional model, the adversary can enumerate escrow IDs and use the revert/success distinction to map which escrows are owned by which addresses (through process of elimination or correlation with known addresses).

The ReineiraOS silent failure pattern eliminates this oracle by ensuring that all redemption attempts complete successfully—the transaction always executes without reverting. The key insight is that the transfer of an encrypted zero amount is indistinguishable from the transfer of a non-zero amount to any observer who does not hold the decryption key. The FHE.select operation produces a ciphertext that, from the perspective of any observer, could represent any value. Only the caller, who can decrypt the result, knows whether they received a non-zero amount.

This pattern has several important properties:

- **No information leakage from reverts**: Failed redemptions are indistinguishable from successful ones on-chain.
- **Gas cost uniformity**: All redemption attempts consume approximately the same gas, preventing gas-based timing attacks.
- **Event uniformity**: The EscrowRedeemed(escrowId) event is emitted regardless of whether the redemption was successful, preventing event-based analysis.

The trade-off is that the caller must decrypt their resulting balance to determine whether the redemption succeeded, introducing a latency requirement for redemption confirmation.

Formal Definition

We formalize this property for clarity and future reference:

> ***Definition 1 (Silent Failure).*** *An escrow redemption protocol satisfies the silent failure property if, for any escrow E and any callers C_1, C_2 where $C_1 = owner(E)$ and $C_2 \neq owner(E)$, the on-chain observable outputs of $redeem(E, C_1)$ and $redeem(E, C_2)$ are computationally indistinguishable under the semantic security of the underlying encryption scheme.*
>
> ***Claim 1.*** *The ReineiraOS escrow redemption protocol satisfies Definition 1 under the LWE hardness assumption.*

Sketch. The observable output of a redemption consists of: (1) the transaction receipt (success in both cases), (2) the emitted event (identical EscrowRedeemed(escrowId) in both cases), (3) the gas consumed (identical FHE operations executed in both cases), and (4) the encrypted transfer amount (a TFHE ciphertext). By the semantic security of TFHE, which reduces to the LWE problem [6], ciphertexts $Enc(paidAmount)$ and $Enc(0)$ are computationally indistinguishable to any polynomial-time adversary without the decryption key. Since the first three outputs are identical by construction and the fourth is indistinguishable by semantic security, the overall outputs are indistinguishable.

This is, to our knowledge, the first application of homomorphic conditional selection as an anti-oracle mechanism in escrow protocols. The pattern is generalizable to any smart contract function where authorization decisions should not be observable.

5.7 Batch Operations

The escrow engine supports batch redemption through redeemMultiple and redeemMultipleAndUnwrap:

The batch operation iterates over the provided escrow IDs, accumulating a total redemption amount homomorphically:

1. Initialize totalRedemption = FHE.asEuint64(0).
2. Encrypt the caller's address once: callerEncrypted = FHE.asEaddress(sender).
3. For each escrow ID:
    a. Skip if the escrow does not exist or the condition is not met.
    b. Perform the same homomorphic authorization checks as single redemption.
    c. Compute the per-escrow redemption amount using FHE.select.
    d. Accumulate: totalRedemption = FHE.add(totalRedemption, escrowRedemption).
    e. Update the escrow's redemption status: escrow.isRedeemed = FHE.or(escrow.isRedeemed, canRedeem).
4. Transfer the total accumulated amount in a single encrypted transfer.

Batch operations provide significant gas savings compared to individual redemptions because they amortize the cost of caller address encryption, the token transfer, and the transaction overhead across multiple escrows.

5.8 Conditional Escrow

The ConfidentialEscrowCondition extension enables escrows to be gated on external conditions evaluated by resolver contracts:

When a resolver is attached to an escrow at creation time, every redemption attempt must first pass the condition check:

Note that the condition check operates in plaintext—the resolver's isConditionMet function returns a plaintext boolean. This is a deliberate architectural decision: condition evaluation may involve external state (timestamps, oracle values, governance decisions) that cannot be encrypted. The condition check occurs before the homomorphic authorization checks, and a failed condition results in a revert (not a silent failure) because the existence of a condition on an escrow is not itself confidential information.

The condition resolver pattern enables a wide range of use cases:

- **Time-locked escrows**: A resolver that checks block.timestamp >= releaseTime.

- **Oracle-gated escrows**: A resolver that checks an external oracle for a specific condition (e.g., delivery confirmation).
- **Multi-signature escrows**: A resolver that checks whether a sufficient number of signers have approved.
- **zkTLS-verified escrows**: A resolver that checks whether a zkTLS proof has been submitted for a specific Web2 verification (see Section 8).

The condition storage uses ERC-7201 namespaced storage to avoid storage collisions in the upgradeable proxy pattern:

5.9 End-to-End Walkthrough

To make the preceding sections concrete, consider a complete escrow lifecycle involving two parties, Alice (buyer on Ethereum) and Bob (seller on Arbitrum).

Creation. Bob calls create(encryptedOwner=Bob, encryptedAmount=500_000000, resolver=DeliveryResolver). The contract assigns escrow ID 42, stores the encrypted owner and amount, initializes encrypted paid amount to Enc(0) and redemption status to Enc(false). On-chain, the only observable fact is: escrow 42 exists. The amount, the owner, and the resolver address are visible, but the encrypted values are opaque.

Cross-chain funding. Alice, on Ethereum, calls depositForBurnWithHook(500_000000, destinationDomain=3, recipient=CCTPV2EscrowReceiver, hookData=abi.encode(42)). Circle validators observe the burn and produce an attestation. The coordinator service distributes the relay task to an operator via SSE. The operator fetches the attestation from Circle's Iris API, claims the task on Arbitrum (claimTask(taskHash)), and executes it. The CCTPHandler calls the CCTPV2EscrowReceiver, which validates the attestation, mints 500 USDC, wraps it to 500 cUSDC, and calls fundFrom(42, Enc(500_000000)). The escrow's paid amount is updated: ePaid = FHE.add(Enc(0), Enc(500_000000)) = Enc(500_000000).

An observer sees: a CCTP transfer arrived and an escrow was funded. They do not know the amount.

Condition resolution. Alice marks the delivery as approved on the platform. A zkTLS proof is generated attesting that the platform's API returned {"deliverable_42": "approved"}. The proof is submitted to the DeliveryResolver contract, which stores the approval.

Redemption. Bob calls redeem(42). The contract:

1. Encrypts Bob's address: eCaller = FHE.asEaddress(Bob).
2. Checks ownership: eIsOwner = FHE.eq(eCaller, escrow.eOwner) – evaluates to Enc(true).
3. Checks payment: eIsPaid = FHE.gte(Enc(500_000000), Enc(500_000000)) – evaluates to Enc(true).
4. Checks status: eNotRedeemed = FHE.not(Enc(false)) – evaluates to Enc(true).
5. Combines: eValid = FHE.and(Enc(true), Enc(true), Enc(true)) – evaluates to Enc(true).

6. Selects amount: eTransfer = FHE.select(Enc(true), Enc(500_000000), Enc(0)) – evaluates to Enc(500_000000).
7. Transfers: confidentialTransfer(Bob, Enc(500_000000)).

Bob receives 500 cUSDC. He can hold it as cUSDC or unwrap to USDC.

Failed redemption (adversary). Eve calls redeem(42). The same six operations execute, but eIsOwner = FHE.eq(Enc(Eve), Enc(Bob)) evaluates to Enc(false). The final eTransfer = FHE.select(Enc(false), …, Enc(0)) = Enc(0). Eve receives a transfer of Enc(0) – a transaction that looks identical to Bob's successful redemption. No revert, no error, no indication of failure.

---

6. Orchestration Network

6.1 Network Design

The orchestration network is a permissionless, stake-weighted network of operators that coordinate off-chain operations required by the protocol. Its primary job today is relaying cross-chain CCTP V2 messages, but the handler architecture is pluggable – new task types can be registered without modifying the core contracts.

The network comprises four on-chain components:

1. **OperatorRegistry**: Manages operator registration, staking, unbonding, task claiming, and authorization.
2. **TaskExecutor**: Routes incoming tasks to registered handlers, verifies operator authorization, and coordinates fee collection.
3. **FeeManager**: Calculates and distributes fees between the protocol and operators.
4. **OperatorSlashingManager**: Implements optimistic dispute resolution for operator misbehavior.

Off-chain, the network comprises:

1. **Coordinator Service**: A NestJS application that monitors source chains for CCTP events, fetches attestations, and distributes relay tasks to operators via Server-Sent Events (SSE).
2. **Operator Nodes**: NestJS applications that receive tasks from the coordinator, claim tasks on-chain, execute cross-chain relays, and report results.
3. **Operator CLI**: A command-line interface for operator management (registration, staking, unbonding, status queries).

6.2 Operator Lifecycle

Registration. An operator registers by calling registerOperator(amount) with a stake amount that meets or exceeds the minimum stake requirement (currently 5,000 GOV tokens). The registration process:

1. Verifies the operator is not already registered.
2. Verifies the operator has not been previously slashed (permanent exclusion).
3. Verifies the stake amount meets the minimum requirement.
4. Checks the sanctions oracle (if configured) to ensure the operator is not on a sanctioned address list.
5. Transfers the staking tokens from the operator to the registry contract.
6. Adds the operator to the active operator set.

Staking. Active operators can increase their stake at any time via addStake(amount). Additional stake cannot be added during an unbonding period.

Unbonding. An operator initiates withdrawal by calling requestUnbond(). This immediately removes the operator from the active set, preventing them from claiming or executing new tasks. The operator's stake remains locked for a 7-day unbonding period, during which slashing may still occur.

Withdrawal. After the 7-day unbonding period, the operator calls withdrawStake() to reclaim their staking tokens. The operator's record is zeroed out, effectively completing their exit from the network.

Slashing. The SlashingManager (or the protocol owner) can slash an operator's stake, partially or fully, upon evidence of misbehavior. Slashing permanently excludes the operator from re-registration.

6.3 Task Execution

The task execution flow involves coordination between the off-chain coordinator, operator nodes, and on-chain contracts:

The task execution model implements a tiered authorization system:

1. **Task Claiming**: Any active operator can claim an unclaimed task by calling claimTask(taskHash). The claim records the operator's address and the block timestamp.
2. **Exclusive Window**: For 60 seconds after claiming, only the claiming operator can execute the task. This window provides the operator with guaranteed exclusivity to submit the transaction without competing with other operators.
3. **Active Operator Window**: After the exclusive window expires (60–600 seconds), any active operator can execute the task. This provides redundancy if the claiming operator fails to execute.

4. **Permissionless Window**: After 600 seconds, any address (including non-operators) can execute the task. This ensures liveness—even if all operators are offline, the task can still be completed by any participant.

6.4 Authorization Tiers

The canExecuteTask function in OperatorRegistry implements a multi-tiered authorization model:

Time Since Claim	Who Can Execute	Rationale
0–60s	Claiming operator only	Exclusive execution right for the operator who claimed
60–600s	Any active operator	Redundancy among staked operators
600s+	Any address	Permissionless fallback for liveness
Unclaimed task	Any active operator	First-come, first-served for unclaimed tasks

The authorization logic is implemented as follows:

This tiered model balances several competing requirements: (1) operators must have economic incentive to claim and execute tasks promptly; (2) the system must remain live even if individual operators fail; (3) permissionless fallback prevents operator collusion from halting the system.

6.5 Fee Model

The FeeManager implements a basis-point fee model with separate protocol and operator fee tiers:

Fee Tier	Basis Points	Percentage	Recipient
Protocol Fee	30 bps	0.30%	Protocol treasury (accumulated)
Operator Fee	50 bps	0.50%	Executing operator (immediate)
Total Fee	80 bps	0.80%	Combined

Fees are calculated on the USDC amount of the cross-chain transfer:

Fee collection is triggered by the TaskExecutor upon successful task execution. The operator fee is immediately transferred to the executing operator, while the protocol fee is accumulated in the FeeManager contract for periodic withdrawal by the protocol owner.

The fee model is configurable by the protocol owner via setFeeConfig(protocolFeeBps_, operatorFeeBps_), with the constraint that the total fee cannot exceed 100% (10,000 basis points). This configurability enables the protocol to adjust fee parameters in response to market conditions, operator supply/demand dynamics, and competitive pressure.

6.6 Slashing and Dispute Resolution

The OperatorSlashingManager implements an optimistic dispute resolution mechanism with the following parameters:

Parameter	Value	Description
Challenge Period	3 days	Time window for challenging a slash proposal
Voting Period	4 days	Time window for stake-weighted voting after challenge
Expiry Period	14 days	Maximum proposal lifetime before automatic expiry
Quorum	10% (1,000 bps)	Minimum stake participation for vote validity
Proposer Bond	5% (500 bps)	Bond required from the proposer (% of slash amount)
Challenger Bond	5% (500 bps)	Bond required from the challenger (% of slash amount)
Slasher Reward	10% (1,000 bps)	Reward for successful slash proposer (% of slashed amount)

The dispute resolution proceeds through the following phases:

Phase 1: Proposal. Any address can propose slashing an operator by calling proposeSlash(operator, amount, evidence). The proposer must post a bond equal to 5% of the proposed slash amount. The proposal enters the "Pending" state.

Phase 2: Challenge (Optional). During the 3-day challenge period, any address can challenge the proposal by calling challenge(proposalId) and posting a challenger bond. If no challenge is received, the proposal can be executed directly after the challenge period expires.

Phase 3: Voting (If Challenged). If challenged, the proposal enters a 4-day voting period where active operators vote using their stake as weight. Votes are binary (for or against) and each operator can vote once.

Phase 4: Resolution. After the voting period, the proposal is resolved:

- **If slash is approved** (quorum reached and majority supports slash): The operator's stake is slashed, the proposer receives their bond back plus a 10% reward from the slashed amount, and the challenger's bond is forfeited to the proposer.
- **If slash is rejected** (quorum not reached or majority opposes): The challenger receives their bond back plus the proposer's forfeited bond.
- **If expired** (14-day expiry reached): All bonds are returned and the proposal is cancelled.

This mechanism ensures that (1) anyone can report operator misbehavior, (2) false reports are economically disincentivized through bond forfeiture, (3) the operator community has the final say through stake-weighted voting, and (4) proposals cannot persist indefinitely.

7. Cross-Chain Settlement

7.1 Bridge-Agnostic Architecture

The cross-chain settlement layer is designed to be bridge-agnostic. Any cross-chain bridge can be integrated through the task handler plugin interface, which defines a standard contract for processing incoming cross-chain messages and routing them to the escrow engine. The core protocol does not depend on any specific bridge protocol.

The current implementation integrates Circle's CCTP V2 to enable cross-chain USDC transfers that settle into confidential escrows on the destination chain. This integration is implemented through two contracts:

1. **CCTPV2Forwarder** (source chain): A convenience contract on the source chain that encodes escrow IDs into CCTP V2 hook data before initiating the burn.
2. **CCTPV2EscrowReceiver** (destination chain): A receiver contract on Arbitrum that processes incoming CCTP V2 messages, wraps received USDC into cUSDC, and funds the target escrow.

Additional bridge integrations (e.g., LayerZero, Axelar, Wormhole) can be implemented as separate handler plugins registered with the task executor, without modifying the core escrow or token contracts.

The integration leverages CCTP V2′s hook data extension to carry application-specific payload alongside the standard burn message. This enables the destination chain receiver to automatically route received USDC to the correct escrow without requiring a separate transaction or off-chain coordination for escrow identification.

7.2 Settlement Flow

The settlement flow exhibits several important security properties:

1. **Atomic execution**: The entire settlement flow—message reception, USDC wrapping, and escrow funding—occurs within a single transaction. If any step fails, the entire transaction reverts, preventing partial settlement states.
2. **Balance-based verification**: The receiver measures its actual USDC balance change rather than trusting the message-encoded amount. This protects against message manipulation and ensures the funded amount matches the actual received value.

3. **Encryption boundary**: The USDC amount is in plaintext only during the wrapping step. Once wrapped into cUSDC and used to fund the escrow, the amount exists only as an FHE ciphertext.

7.3 Hook Data Encoding

CCTP V2 hook data is a variable-length byte array appended to the burn message at byte offset 376. ReineiraOS uses a simple ABI encoding for the hook data:

The extraction function on the receiver side:

The hook data offset (376 bytes) accounts for the complete CCTP V2 message header structure:

Component	Size	Cumulative Offset
CCTP Header	216 bytes	216
Amount	32 bytes	248
Mint Recipient	32 bytes	280
Fee Payer	32 bytes	312
Max Fee	32 bytes	344
Finality + Extra	32 bytes	376
Hook Data Start	–	**376**

This encoding is deliberately minimal. The escrow ID is the only application-specific data required for settlement routing. Additional metadata (sender identity, payment reference, etc.) is not needed because the escrow itself encapsulates all necessary state in its encrypted fields.

7.4 Balance-Based Verification

The CCTPV2EscrowReceiver employs a balance-based verification pattern rather than relying on the message-encoded amount:

This pattern provides defense-in-depth against several attack vectors:

1. **Message replay**: Even if an attestation were somehow replayed, the MessageTransmitter prevents double-minting of the same message. The balance check provides a secondary verification.
2. **Amount mismatch**: If the CCTP V2 implementation were to mint a different amount than encoded in the message (due to fee deductions or other modifications), the balance-based verification captures the actual received amount.
3. **Rate-adjusted wrapping**: The wrapping step accounts for the conversion rate between USDC and cUSDC:

Since the current rate is 1:1, this calculation is a no-op, but the code is future-proofed for scenarios where the rate might differ (e.g., if a future version of cUSDC uses a different decimal precision).

---

8. Condition Resolution Framework

8.1 Design

The condition resolution framework is verification-agnostic: the protocol holds no opinion on how a condition is checked. Any proof system — zkTLS proofs, oracle attestations, multisig votes, time-locks, or FHE computations — can serve as a condition resolver through the IConditionResolver interface (Section 5.8). The protocol only requires that a resolver returns a boolean indicating whether the condition is met.

This section describes the zkTLS integration as a reference implementation. The same pattern applies to any verification technology.

zkTLS Bridge. The zkTLS Bridge is one verification layer that enables cryptographic proofs about Web2 data to be used as conditions for on-chain escrow operations. The bridge architecture connects three domains:

1. **Web2 Domain**: Traditional internet services (banks, payment processors, identity providers, delivery services) that serve data over TLS.
2. **Notary Domain**: A network of notary nodes that co-sign TLS sessions and produce commitments to server responses without learning the response content.
3. **On-Chain Domain**: Condition resolver contracts that verify zkTLS proofs and update escrow conditions accordingly.

The zkTLS Bridge is designed as an extension to the escrow condition resolver pattern (Section 5.8). A zkTLS-backed condition resolver accepts zero-knowledge proofs that attest to specific facts about Web2 data—for example, that a bank transfer was completed, that an identity verification passed, or that a delivery was confirmed—and sets the corresponding escrow condition to "met."

The design follows a "verify then unlock" pattern:

1. A zkTLS proof is generated off-chain by the client in collaboration with a notary network.
2. The proof is submitted to a zkTLS verifier contract on Arbitrum.
3. Upon successful verification, the verifier calls the condition resolver to mark the condition as met.
4. The escrow owner can now redeem the escrow (subject to the other homomorphic conditions).

8.2 Verification Flow

The following diagram illustrates the zkTLS verification flow:

The zkTLS verification flow involves the following steps:

1. **Session Establishment**: The client and notary cooperatively establish a TLS connection to the target Web2 server. The notary participates in the TLS handshake but cannot read the encrypted data.
2. **Data Commitment**: The server's response is received by the client. The notary produces a commitment to the ciphertext it observed during the TLS session, without learning the plaintext.
3. **Selective Disclosure**: The client decrypts the response and generates a zero-knowledge proof that specific assertions about the response data are true. For example: "the JSON response from api.bank.com/transfers contains a field status with value completed for transfer ID TX12345."
4. **Proof Submission**: The zero-knowledge proof, along with the notary's commitment, is submitted to the on-chain verifier contract.
5. **Verification**: The verifier contract checks: (a) the notary's signature on the commitment is valid; (b) the zero-knowledge proof is valid with respect to the commitment; (c) the proof asserts the required facts.
6. **Condition Update**: Upon successful verification, the verifier updates the condition resolver for the associated escrow, enabling redemption.

8.3 Use Cases

The zkTLS Bridge enables several categories of Web2-verified escrow conditions:

Fiat Payment Verification. A seller creates an escrow for a crypto-to-fiat trade. The buyer makes a bank transfer and generates a zkTLS proof that the bank's API confirms the transfer. The proof is submitted on-chain, unlocking the escrow for the seller to redeem. This eliminates the need for trusted intermediaries in peer-to-peer fiat-crypto exchanges.

Identity Verification. A compliance-gated escrow requires the recipient to prove they have passed KYC verification with an approved identity provider. The recipient generates a zkTLS proof from the identity provider's API response (proving verified status without revealing personal data) and submits it on-chain.

Service Delivery Confirmation. A freelance platform escrow is conditioned on delivery confirmation. The freelancer generates a zkTLS proof from the platform's API confirming that the deliverable was submitted and approved, unlocking the escrow for payment.

Insurance Claim Verification. Payment protection insurance claims (Section 9) use zkTLS proofs to verify that a disputed payment was not received by the intended recipient's bank, enabling automated claim resolution without manual review.

API Rate and Price Verification. For DeFi applications, zkTLS can verify exchange rates, market prices, or API responses from centralized exchanges. An escrow conditioned on a price

feed can use zkTLS to verify that a specific price was quoted at a specific time by a specific API endpoint, providing a decentralized alternative to oracle-based price feeds for specific use cases.

Document Verification. zkTLS can verify the presence of specific fields in structured documents served over HTTPS. For example, verifying that a government-issued electronic document (tax filing, business registration) contains specific attributes without revealing the full document content.

---

9. Payment Protection

9.1 Two-Layer Model

ReineiraOS implements a two-layer payment protection model that combines cryptographic verification with economic insurance to protect participants in escrow-based transactions:

Layer 1: Cryptographic Verification. The first layer uses the condition resolution framework (Section 8) to cryptographically verify the state of off-chain payments. Any verification technology supported by the framework — zkTLS proofs, oracle attestations, or other proof systems — can provide deterministic resolution for disputes where the underlying facts can be verified.

Layer 2: Insurance Pool. The second layer provides economic protection for cases where cryptographic verification is insufficient or where the verified outcome results in a loss for one party. The insurance pool operates as a decentralized fund that compensates affected parties according to predefined coverage parameters.

The two layers are complementary: Layer 1 resolves the majority of disputes through objective verification, while Layer 2 provides a safety net for edge cases, system failures, and scenarios that fall outside the scope of automated verification.

9.2 Verification Layer

The verification layer operates as the primary dispute resolution mechanism. The following describes the zkTLS implementation as a reference, though any condition resolver can serve this role:

1. **Pre-conditions**: When an escrow is created for a fiat-crypto exchange, a zkTLS-backed condition resolver is attached. The resolver specifies the target bank API endpoint, the expected transfer parameters, and the verification criteria.
2. **Dispute Initiation**: If the buyer claims to have made a fiat payment but the seller has not received it (or vice versa), either party can initiate dispute resolution.

3. **Evidence Submission**: The disputing party generates a zkTLS proof from the relevant bank or payment processor API. The proof attests to either: (a) the payment was completed (supporting the buyer's claim), or (b) the payment was not received (supporting the seller's claim).
4. **Automated Resolution**: The on-chain verifier validates the zkTLS proof and automatically resolves the dispute by either releasing the escrowed funds to the seller or returning them to the buyer.
5. **Appeal Period**: A configurable appeal period allows the counterparty to submit contradicting evidence before the resolution is finalized.

9.3 Insurance Pool

The insurance pool is a protocol-managed fund that provides coverage for losses that cannot be resolved through cryptographic verification alone. The insurance pool architecture includes:

Pool Funding. The insurance pool is funded through:

- A percentage of protocol fees directed to the insurance reserve.
- Direct contributions from participants who opt into enhanced coverage.
- Governance-approved allocations from the protocol treasury.

Coverage Parameters. Each insured escrow has associated coverage parameters:

- Maximum coverage amount (denominated in USDC).
- Coverage period (time window for claim submission).
- Deductible (minimum loss threshold before coverage applies).
- Premium rate (percentage of escrow value paid for coverage).

Claim Process. Insurance claims follow a structured process:

1. The claimant submits a claim with supporting evidence.
2. The claim enters a review period during which zkTLS proofs and other evidence are evaluated.
3. If the claim meets the coverage criteria, the insurance pool disburses the covered amount to the claimant.
4. The protocol governance can override claim decisions in exceptional circumstances.

Solvency Management. The insurance pool maintains solvency through:

- Exposure limits that cap the total outstanding coverage relative to pool reserves.
- Premium pricing that reflects the actuarial risk of insured transactions.
- Reinsurance arrangements with external insurance providers for catastrophic loss scenarios.

The interaction between the two layers is illustrated below:

Risk Categorization. Insured transactions are categorized by risk level:

Risk Category	Description	Premium Rate	Max Coverage
Low	Verified counterparty, small amount, domestic	0.1%	100% of escrow
Medium	Known counterparty, medium amount, cross-border	0.3%	80% of escrow
High	New counterparty, large amount, high-risk jurisdiction	0.5%	50% of escrow
Custom	Governance-approved custom risk profile	Variable	Variable

---

10. Agentic Layer

10.1 Design

The agentic layer provides programmable automation for ReineiraOS operations. Agents are autonomous software entities that execute predefined workflows in response to on-chain events, temporal triggers, or external conditions. The agentic layer operates within the orchestration network, leveraging the same operator infrastructure for execution and the same economic security model for accountability.

The agent architecture follows a "trigger-action" pattern:

1. **Triggers**: On-chain events (escrow creation, payment receipt, condition changes), temporal events (block timestamps, intervals), or external events (oracle updates, API webhooks).
2. **Actions**: On-chain transactions (escrow operations, token transfers, condition updates), off-chain operations (notification delivery, API calls), or composite workflows combining multiple actions.
3. **Constraints**: Gas limits, execution deadlines, authorization requirements, and budget caps that govern agent behavior.

Agents are registered as task types within the orchestration network, enabling them to be executed by operators through the standard task execution flow.

10.2 Agent Types

ReineiraOS defines three categories of agents:

Escrow Agents. These agents automate escrow lifecycle operations:

- **Auto-Fund Agent**: Monitors for new escrow creation events and automatically initiates CCTP transfers to fund escrows based on predefined rules.
- **Auto-Redeem Agent**: Monitors funded escrows and automatically initiates redemption when conditions are met.
- **Expiry Agent**: Monitors escrow age and triggers refund workflows for escrows that exceed configurable time limits.
- **Notification Agent**: Sends off-chain notifications (webhooks, emails, push notifications) when escrow state changes occur.

Settlement Agents. These agents optimize cross-chain settlement:

- **Batch Settlement Agent**: Aggregates multiple pending CCTP messages and executes them in batch to reduce gas costs.
- **Fee Optimization Agent**: Monitors gas prices and schedules settlements during low-fee periods.
- **Route Optimization Agent**: Selects optimal cross-chain routes based on current liquidity, fees, and confirmation times.

Compliance Agents. These agents enforce regulatory requirements:

- **Sanctions Screening Agent**: Monitors operator registrations and escrow creations against sanctions lists.
- **Transaction Monitoring Agent**: Detects suspicious patterns in escrow activity (unusual amounts, rapid creation, known risk indicators).
- **Reporting Agent**: Generates compliance reports and regulatory filings based on on-chain activity.

10.3 Execution Model

Agent execution follows the standard orchestration network flow with additional constraints:

1. **Registration**: An agent is registered by deploying a task handler contract that implements the ITaskHandler interface and registering it with the TaskExecutor for the TASK_AGENT_CALL task type.
2. **Scheduling**: Agent tasks are submitted to the orchestration network either by external callers or by other agents (enabling agent chaining). Each task includes the agent-specific payload describing the action to be performed.
3. **Authorization**: The TaskExecutor verifies that the submitting operator is authorized to execute agent tasks (following the same tiered authorization model as CCTP relay tasks).
4. **Execution**: The handler contract parses the agent-specific payload and executes the corresponding action. For on-chain actions, this involves direct contract calls. For off-chain actions, the operator executes the action off-chain and submits a proof of execution.

5. **Verification**: Agent execution results are recorded on-chain through the markExecuted function, and fees are distributed according to the fee model.
6. **Monitoring**: Agent activity is monitored by the coordinator service, which tracks execution success rates, latency, and error rates for each agent type.

The agent execution model is designed to be deterministic and verifiable. For on-chain actions, the result of agent execution is fully observable on the blockchain. For off-chain actions, agents produce execution receipts that are signed by the executing operator and can be verified by other network participants.

Agent Security Model. Agents operate within the same security model as the orchestration network:

- Agents cannot access FHE-encrypted values directly; they can only trigger operations on contracts that perform FHE computation.
- Agent execution is bounded by gas limits specified at task submission time.
- Agent actions that involve value transfer are subject to the same reentrancy protections as direct contract calls.
- Misbehaving agents (those that produce incorrect results or fail to execute) expose their operators to slashing through the dispute resolution mechanism.

Agent Composability. Agents can be composed into workflows through task chaining:

1. Agent A executes and produces a result.
2. The result triggers Agent B through an event-driven mechanism.
3. Agent B executes using the result from Agent A as input.
4. The chain continues until all agents in the workflow have executed.

This composability enables complex automated workflows such as "monitor for new escrow creation, verify the counterparty via zkTLS, auto-fund the escrow if verification passes, and notify the seller upon funding completion."

---

11. Plugin Ecosystem

11.1 Architecture

The ReineiraOS plugin ecosystem enables the core protocol to be extended with application-specific functionality without modifying the core contracts. Plugins interact with the protocol through well-defined interfaces and are deployed as independent smart contracts.

The plugin architecture follows two patterns:

Condition Resolver Plugins. These implement the IConditionResolver interface and are attached to individual escrows at creation time. They provide custom gating logic for escrow redemption. The condition resolver pattern is the primary extension point for the escrow engine.

Handler Plugins. These implement the ITaskHandler interface and are registered with the TaskExecutor to support new task types. They enable the orchestration network to handle arbitrary operations beyond CCTP relay.

Both patterns share a common design philosophy: plugins receive callbacks from the core protocol at well-defined points in the execution flow, and their return values influence protocol behavior. Plugins cannot modify core protocol state directly; they can only provide information that the core protocol uses to make decisions.

11.2 Core Plugins

ReineiraOS ships with several core plugins:

Time-Lock Resolver. A condition resolver that gates escrow redemption on a minimum time delay. The resolver stores a release timestamp for each escrow and returns true from isConditionMet only after the specified time has passed.

Multi-Signature Resolver. A condition resolver that requires a configurable number of approvals from a predefined set of signers before allowing escrow redemption. Signers submit approval transactions that are recorded in the resolver's state.

Oracle Resolver. A condition resolver that checks an external oracle (Chainlink, UMA, custom) for a specific value or condition. This enables escrows to be gated on real-world events such as asset prices, weather conditions, or sports outcomes.

CCTP Handler. The primary task handler plugin that processes CCTP V2 relay tasks. This handler is deployed as part of the core protocol and serves as the reference implementation for the handler plugin pattern.

Automation Handler. A task handler plugin for the TASK_AUTOMATION task type that executes pre-configured automation scripts. Automation scripts are registered by protocol administrators and executed by operators through the standard task flow.

11.3 Plugin Interface

The core plugin interfaces are minimal by design:

The IConditionResolver interface consists of a single view function, making it gas-efficient and side-effect-free. The resolver is called during the redemption flow and its return value determines whether the redemption can proceed.

The ITaskHandler interface provides four functions:

- executeTask: Performs the task's primary action and returns a result.
- validateTask: Checks whether a payload is well-formed without executing it.
- getTaskHash: Computes a deterministic hash for the task (used for claiming and deduplication).
- taskType: Returns the handler's task type constant (used for handler registration verification).

Plugin authors must ensure that their implementations are deterministic, gas-bounded, and do not introduce reentrancy vulnerabilities. The core protocol applies nonReentrant guards at the escrow and task executor level, but plugin contracts should also implement their own reentrancy protection.

Plugin Lifecycle. Plugins follow a defined lifecycle:

1. **Development**: The plugin author implements the required interface (IConditionResolver or ITaskHandler).
2. **Deployment**: The plugin contract is deployed independently to the target chain.
3. **Registration**: For task handlers, the plugin is registered with the TaskExecutor via registerHandler(taskType, handler). For condition resolvers, the plugin address is specified at escrow creation time.
4. **Activation**: The plugin begins receiving callbacks from the core protocol.
5. **Deactivation**: Task handlers can be removed via removeHandler(taskType). Condition resolvers cannot be removed from individual escrows after creation (by design, to prevent manipulation).
6. **Upgrade**: Plugin contracts can be upgraded independently if they follow an upgradeable proxy pattern. Upgrades do not require changes to the core protocol.

Plugin Security Considerations. Plugin interactions introduce additional attack surface:

- **Reentrancy via plugins**: A malicious condition resolver could attempt reentrancy during the isConditionMet call. The core protocol mitigates this through the nonReentrant modifier on all escrow operations.
- **Gas griefing**: A condition resolver could consume excessive gas, causing redemption transactions to fail. The protocol addresses this through gas-bounded external calls and the ability for the escrow owner to bypass conditions via governance (in extreme cases).
- **State manipulation**: A condition resolver could return different values for the same escrow depending on the caller or block context, creating non-deterministic behavior. This is mitigated through the view function requirement, which prevents state modifications during condition checks.

---

12. Security Analysis

12.1 Threat Model

The ReineiraOS threat model considers the following adversary classes:

A1: Passive Observer. An adversary who monitors on-chain transactions and state to extract information about escrow contents, payment amounts, and participant identities. This adversary does not submit transactions but has full visibility into the blockchain state.

A2: Active Participant. An adversary who submits transactions to the ReineiraOS contracts to probe escrow state, attempt unauthorized redemptions, or manipulate protocol behavior. This adversary controls one or more Ethereum addresses.

A3: Malicious Operator. An adversary who registers as an operator with the minimum stake and attempts to extract value through task front-running, selective non-execution, or collusion with other operators.

A4: Network-Level Adversary. An adversary who controls the communication layer between the coordinator, operators, and the blockchain. This adversary can delay, reorder, or drop messages.

A5: Compromised Notary. An adversary who controls one or more notary nodes in the zkTLS network and attempts to forge proofs or collude with clients to produce false attestations.

A6: Front-Running Adversary. An adversary who monitors the mempool for pending transactions and attempts to extract value by front-running wrapping operations, escrow creations, or redemptions. This adversary may operate as a validator/sequencer or use MEV infrastructure.

A7: Compromised Owner. An adversary who gains control of the contract owner key and attempts to abuse administrative functions (pausing, upgrading, parameter changes). This represents the most severe adversary class, mitigated through future governance decentralization.

For each adversary class, the following table summarizes the attack surface and mitigations:

Adversary	Primary Attack	Mitigation
A1 (Passive Observer)	Balance inference, transaction graph analysis	FHE encryption, silent failures
A2 (Active Participant)	Unauthorized redemption, state probing	Homomorphic authorization, silent failure
A3 (Malicious Operator)	Task front-running, selective non-execution	Exclusive window, slashing, permissionless fallback
A4 (Network Adversary)	Message delay/drop, operator isolation	SSE reconnection, task timeout, permissionless fallback
A5 (Compromised Notary)	Forged zkTLS proofs	Multi-notary threshold, proof verification on-chain
A6 (Front-Runner)	MEV extraction at wrapping boundary	L2 sequencer ordering, batch wrapping
A7 (Compromised Owner)	Malicious upgrades, parameter manipulation	Timelock, multisig, future DAO governance

12.2 Confidentiality Guarantees

ReineiraOS provides the following confidentiality guarantees:

G1: Balance Confidentiality. Individual cUSDC balances are stored as euint64 ciphertexts and cannot be decrypted by any party other than the balance holder (who has been granted FHE access permission). This guarantee is derived from the semantic security of the TFHE scheme under the Learning With Errors (LWE) assumption.

G2: Transfer Amount Confidentiality. Confidential transfer amounts are encrypted and never appear in plaintext in contract storage or events. The guarantee extends to the FHE evaluation: homomorphic operations on encrypted amounts produce encrypted results without revealing the operands.

G3: Escrow Amount Confidentiality. The expected payment amount, cumulative paid amount, and redemption status of each escrow are stored as FHE ciphertexts. These values are accessible only to parties explicitly granted permission through the FHE access control system.

G4: Escrow Owner Confidentiality. The owner address of each escrow is stored as an eaddress ciphertext. The identity of the escrow beneficiary is hidden from all observers, including operators, protocol administrators, and other escrow participants.

G5: Redemption Outcome Confidentiality. The silent failure pattern (Section 5.6) ensures that the outcome of a redemption attempt is not observable on-chain. Both successful and unsuccessful redemptions produce indistinguishable transaction traces.

12.3 Observable Information

Despite the confidentiality guarantees above, certain information is necessarily observable:

Observable	Information Revealed	Mitigation
Escrow creation event	An escrow was created; escrow ID	Sequential IDs are non-identifying
Escrow funding event	An escrow was funded; escrow ID; funder address	Funder may use intermediaries
Escrow redemption event	A redemption was attempted; escrow ID	Silent failure hides outcome
cUSDC wrapping	Amount of USDC wrapped	Wrapping amounts are public by design
cUSDC unwrapping	Amount of USDC unwrapped (after decryption)	Unwrapping requires owner permission
CCTP transfer amount	Amount of USDC transferred cross-chain	CCTP messages are public
Operator task claims	Which operator claimed a task	Operator identity is public
Gas consumption	Approximate computation complexity	FHE ops have uniform gas cost
Total escrow count	Number of escrows created	Counter is monotonic, non-sensitive
Transaction timing	When operations occurred	Standard blockchain property

The primary information leakage point is the CCTP cross-chain transfer, where the USDC amount is visible in the source chain burn message. This is an inherent limitation of the CCTP protocol, which operates on plaintext USDC. The wrapping boundary at ConfidentialUSDC is the earliest point at which amount confidentiality is established.

12.4 Access Control

ReineiraOS implements defense-in-depth access control across multiple layers:

Contract-Level Access Control:

Function	Access Control	Enforced By
registerOperator	Public (self-registration)	Stake requirement, sanctions check
addStake	Active operators only	Registry state check
requestUnbond	Active operators only	Registry state check
withdrawStake	Unbonded operators only	Unbonding period check
claimTask	Active operators only	Registry state check
executeTask	Authorized operators	Tiered authorization model
markExecuted	Monitor or owner only	Address check
slash	SlashingManager or owner	Address check
proposeSlash	Public	Bond requirement
setConfig	Owner only	onlyOwner modifier
create (escrow)	Public	No restriction (permissionless creation)
fund (escrow)	Public	Escrow must exist
redeem (escrow)	Public (silent failure)	Homomorphic authorization
pause / unpause	Owner only	onlyOwner modifier
upgradeToAndCall	Owner only	UUPS _authorizeUpgrade

FHE Access Control:

Encrypted values are accessible only to addresses that have been explicitly granted permission. The permission model is enforced by the Fhenix CoFHE framework and cannot be bypassed by smart contract interactions. Granting access to an encrypted value requires the current holder to explicitly call FHE.allow(value, address).

Sanctions Compliance:

The OperatorRegistry integrates an optional sanctions oracle that checks operator addresses against sanctions lists during registration. If a sanctions oracle is configured, sanctioned addresses are prevented from registering as operators.

12.5 Cross-Chain Security

Cross-chain operations introduce additional security considerations:

CCTP Attestation Security. The security of cross-chain settlement depends on the integrity of Circle's attestation service. An attestation is a cryptographic signature from Circle confirming that

a specific burn message was observed on the source chain. The trust assumption is that Circle's attestation service is honest—it does not sign attestations for burns that did not occur. This is a centralized trust assumption inherent to the CCTP protocol.

Message Replay Protection. The CCTP V2 MessageTransmitter implements nonce-based replay protection. Each burn message contains a unique nonce that is marked as used upon receipt, preventing the same message from being processed twice.

Hook Data Integrity. The hook data (escrow ID) is part of the attested message and cannot be modified without invalidating the attestation. An adversary cannot redirect a cross-chain transfer to a different escrow without producing a valid attestation for the modified message.

Operator Collusion. The worst case for operator collusion is task non-execution (liveness failure), not asset theft. Operators cannot steal funds because they do not have access to the FHE-encrypted escrow state. The permissionless fallback (600s timeout) ensures that operator collusion cannot permanently halt the system.

12.6 Trust Assumptions

ReineiraOS relies on the following trust assumptions:

Assumption	Description	Failure Mode
T1: FHE Security	The TFHE scheme is semantically secure under the LWE assumption	Encrypted values could be decrypted; all confidentiality lost
T2: Fhenix CoFHE	The CoFHE co-processor correctly evaluates FHE operations	Incorrect computation results; potential fund loss
T3: Arbitrum L2	Arbitrum correctly executes EVM transactions and provides data availability	Transaction censorship or reordering
T4: Circle CCTP	Circle's attestation service is honest and available	Cross-chain settlement halted or fraudulent
T5: Circle USDC	USDC maintains its 1:1 peg and is not paused or blacklisted	Wrapped cUSDC loses value or becomes unredeemable
T6: zkTLS Notary	zkTLS notaries do not collude with clients to forge proofs	False verification of Web2 conditions
T7: Operator Liveness	At least one operator (or any participant after 600s) is willing to execute tasks	Task execution delayed but not permanently blocked

The most critical trust assumption is T1 (FHE security). If the TFHE scheme is broken, all encrypted values in the system become readable, and the confidentiality guarantees are voided. However, the protocol's integrity and availability are unaffected—funds remain safe even if

confidentiality is lost, because the authorization logic (ownership checks, payment verification) is enforced through FHE operations that produce correct results regardless of whether an observer can decrypt intermediate values.

Trust Assumption Layering. The trust assumptions are structured in layers of decreasing criticality:

- **Critical (system-breaking if violated)**: T1 (FHE security), T2 (CoFHE correctness). Violation compromises confidentiality or fund safety.
- **Important (functionality-degrading if violated)**: T3 (Arbitrum L2), T4 (Circle CCTP), T5 (Circle USDC). Violation halts operations or degrades settlement capability.
- **Moderate (feature-degrading if violated)**: T6 (zkTLS notary), T7 (operator liveness). Violation disables specific features but does not affect core fund safety.

The protocol is designed to degrade gracefully as trust assumptions are violated. For example, if CCTP becomes unavailable (T4 violated), existing escrows continue to function—only new cross-chain funding is affected. If operator liveness is lost (T7 violated), the permissionless fallback ensures tasks can still be executed by any participant after the 600-second delay.

Formal Security Properties. ReineiraOS satisfies the following formal properties:

- **Safety**: No party can redeem an escrow they do not own, regardless of the number of redemption attempts or the ordering of transactions. This follows from the homomorphic ownership check: FHE.eq(escrow.owner, callerEncrypted) produces Enc(false) for any caller whose address does not match the encrypted owner.
- **Liveness**: Every fully-funded escrow can be redeemed by its owner within a bounded time, assuming at least one transaction can be submitted to the Arbitrum chain. This follows from the fact that redemption does not depend on operator cooperation or external services.
- **Confidentiality**: The probability that an adversary can distinguish between two escrows with different amounts is negligible under the LWE assumption. This follows from the semantic security of TFHE: $Enc(a)$ and $Enc(b)$ are computationally indistinguishable for any $a \neq b$.
- **Non-repudiation**: Once an escrow is created and the EscrowCreated event is emitted, the creation cannot be denied. The event serves as an immutable record of escrow instantiation on the Arbitrum chain.

---

13. Implementation

13.1 Contract Architecture

The ReineiraOS smart contract architecture follows a modular, upgradeable design organized into three functional modules:

Confidential Tokens Module. A single non-upgradeable contract (ConfidentialUSDC) that extends the FHERC20Wrapper base from the Fhenix confidential contracts library. The immutable design eliminates upgrade-related attack vectors for the token contract, which holds all deposited USDC.

Confidential Escrow Module. A set of upgradeable contracts implementing the escrow engine. The core ConfidentialEscrow contract and the CCTPV2EscrowReceiver follow the UUPS proxy pattern, enabling protocol evolution without state migration. Abstract extensions for unwrapping and condition resolution use ERC-7201 namespaced storage to avoid storage collision risks in the upgradeable proxy context.

Orchestration Module. A set of upgradeable contracts managing operator registration, task execution, fee collection, slashing, and CCTP message handling. All contracts follow the UUPS proxy pattern with owner-restricted upgrade authorization.

All upgradeable contracts inherit from a common base that combines OpenZeppelin's Initializable, UUPSUpgradeable, Ownable, ReentrancyGuard, and ERC2771Context modules. This base provides a consistent security foundation across the protocol while supporting gasless meta-transactions through a trusted forwarder.

The off-chain infrastructure consists of a coordinator service for task distribution and operator nodes for task execution, communicating via server-sent events. An operator CLI provides registration, staking, and manual relay capabilities.

13.2 Deployment Configuration

ReineiraOS is deployed on Arbitrum Sepolia as the destination chain, with Ethereum Sepolia serving as the source chain for CCTP burn operations. This two-chain testnet configuration validates the full cross-chain settlement flow while operating in a controlled environment.

The deployment leverages existing Circle infrastructure: USDC on Ethereum Sepolia and the CCTP V2 MessageTransmitter on Arbitrum Sepolia. All ReineiraOS contracts are deployed through OpenZeppelin's proxy deployment tooling with upgrade safety validation.

13.3 Protocol Parameters

The protocol defines the following configurable parameters, which are set at deployment and modifiable through governance:

Parameter	Value	Rationale
Minimum Operator Stake	5,000 GOV tokens	Economic barrier against Sybil attacks
Exclusive Window	60 seconds	Reward for timely execution
Permissionless Delay	600 seconds	Liveness guarantee fallback
Unbonding Period	7 days	Protection against stake-and-slash attacks
Protocol Fee	30 basis points	Sustainable protocol revenue
Operator Fee	50 basis points	Economic incentive for operators
Challenge Period	3 days	Sufficient time for dispute evidence
Voting Period	4 days	Operator participation in dispute resolution
Slasher Reward	10% of slashed amount	Incentive for monitoring operator behavior

These parameters represent initial testnet values. Mainnet deployment will involve formal parameter analysis to balance security, liveness, and economic sustainability.

13.4 Gas Consumption

FHE operations impose additional gas costs compared to plaintext arithmetic. The fundamental FHE primitives — encryption, homomorphic addition, comparison, and conditional selection — each consume between 50,000 and 200,000 gas. Composite protocol operations build on these primitives:

- **Escrow creation** requires approximately 500,000 gas, dominated by the initial encryption of owner address, amounts, and redemption status, plus permission grants.
- **Escrow funding** consumes approximately 600,000 gas for encrypted balance verification, homomorphic addition of the payment amount, and confidential token transfer.
- **Escrow redemption** is the most expensive operation at approximately 900,000 gas, requiring homomorphic authorization checks (address equality, payment sufficiency, redemption status) and conditional token transfer.
- **Batch redemption** amortizes the initial address encryption across N escrows, costing approximately 200,000 + N x 700,000 gas.

These costs are significantly higher than equivalent plaintext operations but remain within practical bounds for the Arbitrum L2 environment, where base gas costs are substantially lower than Ethereum L1. At current Arbitrum gas prices, the additional FHE overhead adds approximately \$0.05–\$0.50 per escrow operation depending on complexity.

13.5 Dependencies

The protocol builds on established open-source foundations:

- **OpenZeppelin Contracts**: ERC-20 token standards, access control, reentrancy protection, and UUPS proxy infrastructure.
- **Fhenix CoFHE**: FHE type system, encrypted arithmetic operations, and the FHERC20 confidential token base contracts.
- **Circle CCTP V2**: Cross-chain USDC transfer infrastructure providing burn-and-mint mechanics with attestation verification.

The off-chain infrastructure uses NestJS for the coordinator and operator services, with ethers.js for blockchain interaction. The coordinator implements a domain-driven design for task lifecycle management, while the operator node follows a hexagonal architecture that enables testing through mock adapters.

---

14. Related Work

The design of ReineiraOS draws upon and distinguishes itself from several categories of prior work:

Privacy-Preserving Tokens. Tornado Cash [11] provides transaction-level privacy through zero-knowledge proofs and mixing pools but does not support programmable escrow or confidential computation over balances. Zcash [12] implements shielded transactions using zk-SNARKs but operates as a standalone blockchain rather than an infrastructure layer for existing stablecoins. Aztec Protocol [13] brings zero-knowledge privacy to Ethereum through a rollup architecture with its own programming model. ReineiraOS differs from all of these by using FHE rather than zero-knowledge proofs, enabling ongoing computation over encrypted values rather than merely proving statements about them.

FHE on Blockchain. Fhenix [14] provides the FHE infrastructure (CoFHE co-processor, FHERC20 standard) that ReineiraOS builds upon. Zama [15] develops the TFHE library and fhEVM, an FHE-enabled virtual machine. Inco Network [16] provides a confidential computing chain using FHE. ReineiraOS is distinguished by its focus on stablecoin infrastructure rather than general-purpose FHE computing, providing purpose-built escrow, cross-chain settlement, and operator coordination primitives.

Cross-Chain Bridges. LayerZero [17], Axelar [18], and Wormhole [19] provide general-purpose cross-chain messaging. Circle's CCTP [9] provides native USDC bridging. ReineiraOS does not compete with these protocols but rather integrates CCTP V2 as a transport layer, adding confidential settlement on the destination chain as a value-added layer.

Escrow Protocols. OpenZeppelin's Escrow and ConditionalEscrow contracts provide basic plaintext escrow functionality. Kleros [20] provides decentralized dispute resolution for escrow. ReineiraOS extends the escrow concept with FHE encryption, eliminating the information exposure inherent in plaintext escrow contracts.

Operator Networks. EigenLayer [21] provides a restaking framework for shared security. Chainlink [22] operates a decentralized oracle network with staking. The Keeper Network (Keep3r) [23] coordinates off-chain task execution. ReineiraOS's orchestration network is purpose-built for CCTP relay coordination and does not require restaking or external oracle dependencies.

zkTLS Protocols. TLSNotary [24] provides the foundational zkTLS protocol for notarized TLS sessions. Reclaim Protocol [25] builds on zkTLS for identity and credential verification. ReineiraOS integrates zkTLS as a condition resolver for escrow operations, creating a bridge between Web2 verification and on-chain settlement.

14.1 Feature Comparison

The following table provides a direct comparison across the most relevant dimensions. "Native" means the capability is a first-class protocol feature; "Possible" means it can be built on top but is not provided; "No" means it is architecturally precluded.

Capability	Tornado Cash	Aztec	Fhenix (raw)	OZ Escrow	Kleros	ReineiraOS
Encrypted balances	No	ZK-based	FHE	No	No	FHE (native)
Computation on encrypted state	No	Limited (circuits)	Yes	No	No	Yes (native)
Programmable escrow	No	Possible	Possible	Plaintext	No	Encrypted (native)
Cross-chain settlement	No	No	No	No	No	CCTP V2 (native)
Operator network	No	Sequencer	No	No	Jurors	Staked operators
Dispute resolution	No	No	No	No	Yes	zkTLS + insurance
Web2 verification	No	No	No	No	Manual	zkTLS (native)
Stablecoin-specific	No	No	No	No	No	Yes
ERC-20 composability	No (mixing)	Own token model	Yes	Yes	N/A	Yes (ERC-7984)

The key differentiator is the combination: ReineiraOS is the only system that provides encrypted escrow with cross-chain settlement and Web2 verification as integrated primitives. Each individual capability exists elsewhere; the integration does not.

15. Future Work

The ReineiraOS protocol is designed for iterative extension. We identify the following areas for future development:

Mainnet Deployment. The current deployment on Arbitrum Sepolia serves as a testnet implementation. Mainnet deployment will require: (1) removal of testnet-specific contracts and migration to production-grade base contracts; (2) formal security audits by independent auditing firms; (3) progressive decentralization of contract ownership from a single owner to a governance multisig and eventually to on-chain governance; (4) integration with mainnet CCTP V2 and mainnet USDC contracts.

Multi-Chain Expansion. The protocol architecture supports expansion to additional destination chains beyond Arbitrum. Target chains include Base, Optimism, Polygon, and Avalanche. Multi-chain deployment requires: (1) FHE support on the destination chain (currently limited to Fhenix-compatible chains); (2) deployment of escrow and receiver contracts on each destination chain; (3) coordinator updates to support multi-chain task distribution.

Governance Token and DAO. The current GOV token serves as a staking token for the operator network. Future work will formalize governance: (1) token distribution and vesting schedules; (2) on-chain governance for protocol parameter changes; (3) treasury management; (4) fee parameter adjustment through governance proposals.

Enhanced Privacy. Future versions will address remaining information leakage points: (1) private mempools or encrypted transaction submission to prevent MEV and front-running at the wrapping boundary; (2) encrypted event parameters to hide escrow IDs in events; (3) FHE-encrypted CCTP messages (pending CCTP protocol support) to achieve full end-to-end confidentiality.

Advanced Escrow Types. Planned extensions include: (1) multi-party escrows with encrypted participant lists; (2) streaming escrows with encrypted rate parameters; (3) auction escrows with sealed-bid mechanics using FHE; (4) hierarchical escrows with nested conditions and cascading releases.

Insurance Protocol Formalization. The payment protection insurance pool (Section 9.3) will be formalized into a standalone protocol with: (1) actuarial pricing models; (2) reinsurance integration; (3) governance-controlled claim adjudication; (4) capital efficiency optimization through structured tranches.

Zero-Knowledge Proof Integration. Future versions will explore hybrid FHE-ZKP architectures where zero-knowledge proofs are used to attest to properties of FHE ciphertexts without decrypting them. This enables: (1) compliance proofs that demonstrate escrow amounts fall within regulatory thresholds without revealing exact amounts; (2) audit proofs that demonstrate total

protocol exposure without revealing individual escrow details; (3) solvency proofs for the insurance pool.

Decentralized Coordinator. The current coordinator service is a centralized component that distributes tasks to operators. Future work will decentralize the coordinator through: (1) a peer-to-peer gossip protocol for task distribution; (2) consensus among operators for task ordering; (3) elimination of the single point of failure in the coordinator service.

Cross-Chain Escrow. Extending the escrow engine to support escrows that span multiple chains, where creation occurs on one chain and redemption on another. This requires: (1) encrypted state synchronization across chains; (2) cross-chain FHE key management; (3) atomic cross-chain redemption protocols.

Agent Marketplace. An open marketplace for agentic automation, where developers can publish, discover, and monetize agent templates. The marketplace will include: (1) agent registry with reputation scoring; (2) standardized agent interfaces for interoperability; (3) revenue sharing between agent developers and operators.

Performance Optimization. Ongoing work to reduce FHE operation gas costs through: (1) batched FHE operations that amortize fixed costs; (2) optimized ciphertext serialization; (3) pre-compiled contracts for common FHE operation patterns; (4) integration with hardware acceleration for FHE co-processor nodes.

---

16. Conclusion

The starting observation is simple: stablecoins handle trillions in volume but cannot natively hold funds conditionally, keep balances private, verify real-world events, or protect participants from counterparty risk. ReineiraOS addresses this gap.

The protocol's core contributions are:

1. **A stablecoin-agnostic confidential token wrapper** that converts any ERC-20 stablecoin into FHE-encrypted balances with full homomorphic arithmetic – addition, comparison, conditional selection – over ciphertext. The wrapping boundary is the only point where plaintext values exist. The reference implementation wraps USDC into cUSDC.
2. **An encrypted escrow engine** where owner addresses, amounts, payment status, and redemption flags are all TFHE ciphertexts. The silent failure pattern (Definition 1) is, to our knowledge, the first application of homomorphic conditional selection as an anti-oracle mechanism. It eliminates an entire class of information leakage attacks that affect all existing plaintext escrow implementations.

3. **A decentralized operator network** with stake-weighted authorization, three-tiered execution windows (exclusive, open, permissionless), and optimistic dispute resolution. The permissionless fallback at 600 seconds provides an unconditional liveness guarantee.
4. **Bridge-agnostic cross-chain settlement** through a pluggable handler interface, currently implemented with CCTP V2 and hook data encoding. Any cross-chain bridge can be integrated without modifying core contracts, enabling a user on any supported chain to fund an encrypted escrow through a single transaction.
5. **A verification-agnostic condition resolution framework** and **plugin ecosystem** that connect any proof system — zkTLS, oracle attestations, multisig votes, time-locks — to on-chain escrow conditions, and enable jurisdiction-aware compliance without modifying core contracts.

The protocol is deployed on Arbitrum Sepolia. The gas overhead of FHE operations – approximately $0.05 to $0.50 per escrow operation at current L2 prices – is manageable for commercial use cases and will decrease as FHE hardware acceleration matures.

What remains is execution: mainnet deployment after audits, zkTLS and insurance pool implementation, multi-chain expansion, and progressive decentralization of governance. The architecture is designed for this – each component is independently upgradeable, and the plugin interface ensures that extensions do not require core protocol changes.

The infrastructure pieces are in place. The question is no longer whether confidential stablecoin operations are technically feasible. It is what applications will be built on top of them.

References

[1] DeFilLama, "Stablecoin Dashboard," 2025. Available: https://defillama.com/stablecoins

[2] C. Gentry, "Fully Homomorphic Encryption Using Ideal Lattices," in *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC)*, 2009, pp. 169–178.

[3] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) Fully Homomorphic Encryption without Bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS)*, 2012, pp. 309–325.

[4] J. Fan and F. Vercauteren, "Somewhat Practical Fully Homomorphic Encryption," *Cryptology ePrint Archive*, Report 2012/144, 2012.

[5] C. Gentry, A. Sahai, and B. Waters, "Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based," in *Advances in Cryptology – CRYPTO 2013*, Springer, 2013, pp. 75–92.

[6] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "TFHE: Fast Fully Homomorphic Encryption over the Torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[7] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic Encryption for Arithmetic of Approximate Numbers," in *Advances in Cryptology – ASIACRYPT 2017*, Springer, 2017, pp. 409–437.

[8] Ethereum Improvement Proposals, "ERC-7984: Confidential ERC-20 Token Standard," 2024. Available: https://eips.ethereum.org/EIPS/eip-7984

[9] Circle, "Cross-Chain Transfer Protocol V2 Technical Documentation," 2024. Available: https://developers.circle.com/stablecoins/cctp-getting-started

[10] TLSNotary Project, "TLSNotary Protocol Specification," 2024. Available: https://docs.tlsnotary.org/protocol/

[11] Tornado Cash, "Tornado Cash Whitepaper," 2019. Available: https://tornado.cash/Tornado.cash_whitepaper_v1.4.pdf

[12] E. B. Sasson et al., "Zerocash: Decentralized Anonymous Payments from Bitcoin," in *IEEE Symposium on Security and Privacy*, 2014, pp. 459–474.

[13] Aztec Protocol, "Aztec Connect: Privacy-First DeFi," 2023. Available: https://aztec.network/

[14] Fhenix, "Fhenix: Fully Homomorphic Encryption for Ethereum," 2024. Available: https://www.fhenix.io/

[15] Zama, "fhEVM: Confidential Smart Contracts on Ethereum Using Fully Homomorphic Encryption," 2024. Available: https://www.zama.ai/fhevm

[16] Inco Network, "Inco: The Confidential Computing Chain," 2024. Available: https://www.inco.org/

[17] LayerZero Labs, "LayerZero: An Omnichain Interoperability Protocol," 2022. Available: https://layerzero.network/

[18] Axelar Network, "Axelar: Secure Cross-Chain Communication," 2022. Available: https://axelar.network/

[19] Wormhole, "Wormhole: A Generic Cross-Chain Message Passing Protocol," 2022. Available: https://wormhole.com/

[20] F. Ast and C. Lesaege, "Kleros: Short Paper v1.0.7," 2019. Available: https://kleros.io/

[21] EigenLayer, "EigenLayer: The Restaking Collective," 2023. Available: https://eigenlayer.xyz/

[22] Chainlink, "Chainlink 2.0: Next Steps in the Evolution of Decentralized Oracle Networks," 2021. Available: https://chain.link/whitepaper

[23] Andre Cronje, "Keep3r Network: Decentralized Keeper Ecosystem," 2020. Available: https://keep3r.network/

[24] TLSNotary, "TLSNotary: Provable Data from Any Website," 2024. Available: https://tlsnotary.org/

[25] Reclaim Protocol, “Reclaim: zkTLS Proofs of Web Data,” 2024. Available: https://www.reclaimprotocol.org/

Appendix A: Notation

Symbol	Description
$Enc_{pk}(m)$	Encryption of message m under public key pk
$Dec_{sk}(c)$	Decryption of ciphertext c under secret key sk
$Eval(f, c_1, ..., c_n)$	Homomorphic evaluation of function f over ciphertexts
Enc(uint64)	FHE-encrypted 64-bit unsigned integer
Enc(address)	FHE-encrypted 160-bit Ethereum address
Enc(bool)	FHE-encrypted boolean value
BPS	Basis points (1 BPS = 0.01%)
$\mathbb{T}$	Real torus $\mathbb{R}/\mathbb{Z}$
LWE	Learning With Errors problem
