A zkTLS-based Decentralized Credential Verification Mechanism

Page • zkTLS Architecture +1

Version: 1.0

Date: October 31, 2025

Abstract

The Lighter.IM Protocol is a pioneering decentralized fiat On/Off-Ramp solution. This protocol leverages Zero-Knowledge Proof (ZKP) technology in conjunction with zkTLS (a trusted-witness-based TLS proxy mechanism) to provide a trustless, secure, efficient, and privacy-preserving environment for users to enter and exit the Web3 world.

This whitepaper details the core architecture, cryptographic principles, and operational workflow of Lighter.IM. The protocol's key innovation lies in combining the non-custodial security of Decentralized Finance (DeFi) with off-chain credential verification based on ZKP. Users can generate ZK proofs of their transactions or data from any Web2 website (such as an online bank) via HTTPS sessions, all from their local device (e.g., a browser extension).

This mechanism enables on-chain smart contracts (Relying Parties) to reliably verify the authenticity of a user's off-chain fiat operations without the user ever exposing sensitive information, such as account credentials, specific balances, or transaction histories. Lighter.IM acts as a faithful trustee for the counterparty's intent, automatically executing the release of escrowed assets upon successful proof verification, thereby creating a secure and automated Web3 on/off-ramp bridge.

1. Introduction

1.1 Background: The Core Challenge of Web3 On/Off-Ramps

Currently, the fiat on/off-ramp channels connecting Traditional Finance (TradFi) with the Web3 world face significant bottlenecks in trust, efficiency, and privacy. Centralized Exchanges (CEXs) require users to custody their assets, introducing risks of single points of failure and privacy leaks. Meanwhile, Over-the-Counter (OTC) or P2P markets rely on fragile social trust and manual arbitration, which is inefficient and fraught with fraud risk.

1.2 The Lighter.IM Solution

The Lighter.IM protocol aims to solve these problems at their root. We propose a non-custodial solution where the foundation of trust is not a centralized entity or social reputation, but rather verifiable cryptography and code.

This protocol allows users to use their operations on any bank or payment application (Data Holder), such as a "completed transfer," as "evidence." Through the zkTLS proxy model and a local ZK circuit, they generate a concise, encrypted "credential" (a ZK proof). The on-chain smart contract (Relying Party) only needs to verify this credential to be convinced that the user has completed the agreed-upon off-chain action, thereby automatically releasing the locked digital assets.

1.3 Core Technical Challenges

To achieve this goal, the Lighter.IM protocol must solve three core challenges:

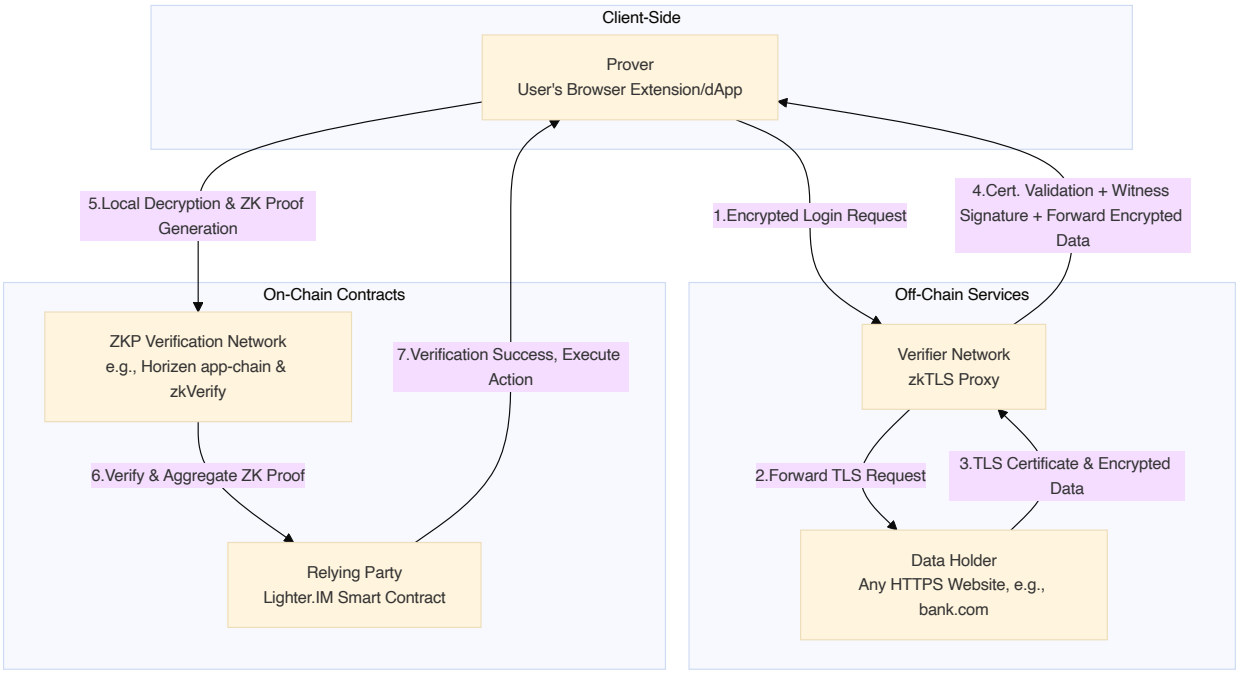
1. **Authenticity vs. Forgery:** How to ensure the credential provided by the user (e.g., a bank transfer record) genuinely originates from the target server (e.g., `bank.com`) and is not locally forged by the user?
2. **Privacy vs. Verification:** How to prove to a contract that "a $100 transfer to account 123456 was completed at 10:00 AM on Jan 1, 2025" without revealing any additional information, such as the specific account balance, account holder's identity, full transaction history, or login credentials?
3. **High Computation vs. On-Chain Efficiency:** ZKP generation (Proving) is computationally intensive, and its verification on-chain (especially on EVM-compatible chains) is expensive. How can we build a system that is both user-friendly and economically viable?

2. Protocol Architecture and Core Components

Lighter.IM adopts a zkTLS architecture based on the Proxy Model. This architecture decouples the four key stages: data acquisition, witnessing, proof generation, and proof verification.

2.1 Architecture Overview

The protocol's ecosystem consists of five core components that work in concert to achieve a secure and private flow of credentials.



2.2 Core Component Definitions

1. **Prover (User):**
    - **Actor:** The user's client software, such as the Lighter.IM browser extension or mobile app.
    - **Responsibilities:**
        - Holds the login credentials for the target website (Data Holder);
        - Initiates the proof request;
        - Collaborates with the Verifier to complete the proxied TLS session;
        - Locally decrypts the session data to extract key information (the secret plaintext); (e) Runs the ZK circuit locally to generate the final ZK proof.
2. **Data Holder:**
    - **Actor:** Any website providing services via HTTPS, e.g., `bank.com`, `wise.com`, etc.
    - **Responsibilities:**
        - Provides data services as normal.
        - A major advantage of the Lighter.IM protocol is that it **requires absolutely no modification from the Data Holder**. The server is not even aware it is being "proven."
3. **Verifier Network (zkTLS Proxy):**
    - **Actor:** A (decentralized) group of nodes acting as a "semi-trusted" proxy.
    - **Responsibilities:**

        - **TLS Witness:** Forwards the Prover's network traffic and **rigorously validates** the Data Holder's TLS certificate to ensure its identity is authentic;
        - **Key Collaborator:** Holds a share (e.g., K2) of the TLS session key, making it **unable to independently decrypt** the session content;
        - **Witness Signature:** Computes a hash of the **Encrypted Data Stream (EncryptedData)**—which it cannot decrypt—and signs it, guaranteeing the data's authentic origin.
4. **Relying Party (dApp):**
    - **Actor:** Any third party that needs to consume the proof, specifically referring to an **on-chain smart contract** in the Lighter.IM scenario.
    - **Responsibilities:**
        - Defines the required proof format (i.e., the ZK circuit logic);
        - Receives the verified proof, relayed by the ZKP Verification Network;
        - Automatically executes business logic (e.g., releasing escrowed USDC) upon confirming the proof's validity.
5. **ZKP Verification Network:**
    - **Actor:** A specialized (application) chain for ZKP verification, such as Horizen app-chain and zkVerify.
    - **Responsibilities:** Solves the third challenge from section 1.3.
        - Provides an efficient, low-cost environment to execute the ZK proof verification algorithm;
        - Acts as an aggregator and relayer, securely passing the verification result to the Relying Party on the target chain (e.g., Ethereum).

3. Core Mechanisms and Security Model

Lighter.IM's security and privacy guarantees are built upon the clever splitting of the TLS protocol and the cryptographic commitments of ZK proofs.

3.1 Authenticity Guarantee: TLS Session Witnessing

To prevent the Prover from connecting to a forged server (e.g., `fake-bank.com`) to generate a false proof, **all TLS handshakes must be proxied through the Verifier.**

- The Prover initiates a session request to the Verifier.
- The Verifier initiates a TLS connection to the Data Holder (`bank.com`) on the Prover's behalf.
- `bank.com` returns its TLS certificate.

- The Verifier **immediately validates this certificate** on the proxy side, checking that it is signed by a trusted root Certificate Authority (CA) and that the domain matches.
- The session only proceeds if the validation is successful. This mechanism guarantees the authenticity of the communicating party at the source.

3.2 Privacy Protection: Split Session Keys

To prevent the Verifier from snooping on user data (like bank passwords or balances), the session key negotiation is split.

- The TLS session's Master Secret is generated collaboratively by the Prover and the Verifier (e.g., via MPC or key exchange).
- Ultimately, the Prover holds key share K1, and the Verifier holds key share K2.
- **Neither party can decrypt the session data with their share alone.**
- When `bank.com` sends encrypted data (`EncryptedData`) back, the Verifier cannot decrypt it and can only forward it untouched to the Prover.
- The Prover, in their local client, can only decrypt the data using a combination of K1 and K2, revealing the plaintext HTML (`PlaintextData`).
- **Result:** The Verifier, acting as a proxy, only ever sees encrypted gibberish, learning nothing about the user's sensitive information.

3.3 Reliable Attestation: Encrypted Stream Signature

This is the key mechanism that ensures the Prover does not "swap out" the data locally.

1. When the Verifier receives `EncryptedData` from `bank.com` , before forwarding it, it computes a hash of the encrypted packet: `H(EncryptedData)` .
2. The Verifier signs this hash with its private key, generating a **witness signature**: `Sig(H(EncryptedData))` .
3. The Verifier sends both `EncryptedData` and `Sig(H(EncryptedData))` to the Prover.

This signature means: "I (Verifier) guarantee that an encrypted data packet with the fingerprint H(...) truly originated from the `bank.com` server I authenticated."

3.4 Final Privacy & Binding: The ZK-Proof

After the Prover decrypts the data and extracts the necessary information (e.g., "transfer of $100"), they execute the ZK circuit. This ZK proof mathematically binds all of the following assertions:

- **Private Inputs:**

1. The complete plaintext data `PlaintextData` .

2. The session key (or its shares) used for decryption.

- **Public Inputs:**

1. The Data Holder's domain name (e.g., `bank.com`).
2. The Verifier's witness signature `Sig(H(EncryptedData))`.
3. The hash of the encrypted data `H(EncryptedData)`.
4. The Prover's public claim (e.g., "A $100 transfer to account 123456 was made after 10:00 AM on Jan 1, 2025").

The ZK circuit verifies:

- (a) That decrypting `EncryptedData` (whose hash must equal the public input `H(EncryptedData)`) using the session key indeed results in `PlaintextData`.
- (b) That the information extracted from `PlaintextData` correctly satisfies the public claim.

When the Relying Party (on-chain contract) verifies this ZK proof, it:

1. Checks the validity of the Verifier's signature `Sig(H(EncryptedData))`.
2. Verifies the ZK proof itself.

Why cheating is ineffective: If the Prover forges plaintext data `FakePlaintext`, it must correspond to some `FakeEncryptedData`. However, the Prover cannot obtain the Verifier's signature for `H(FakeEncryptedData)`. The ZK circuit would fail because the public inputs (the signature and the hash) would not match the private data. The Prover cannot forge data without breaking this hash-and-signature bond.

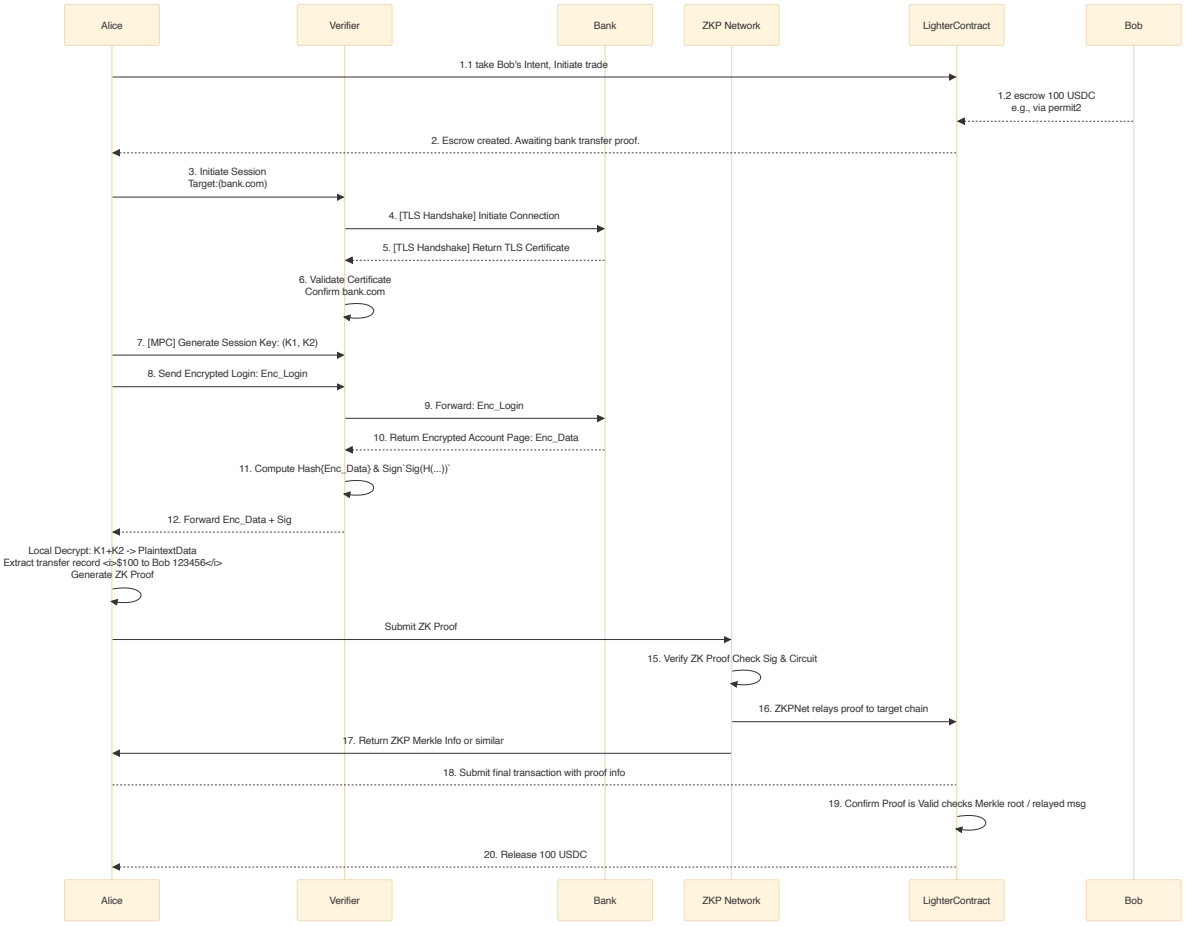
4. Detailed Workflow

4.1 Scenario Definition

Scenario: User Alice (Prover) wants to buy 100 USDC from Bob on Lighter.IM (Relying Party) for $100. Bob requires Alice to prove she has completed a $100 fiat transfer to his account via `bank.com` (Data Holder).

4.2 Flowchart: Lighter.IM Credential Generation and Verification

The following sequence diagram illustrates the complete interaction steps:



(Note: Step 16 or steps 17/18 can be used depending on the chain aggregation design.)

4.3 Interaction Steps Explained

1. **Step 1: Proof Request**
    - Alice initiates an escrow trade on the Lighter.IM contract (Relying Party). The contract requires Alice to provide a proof from `bank.com` for "a $100 transfer to Bob's account 123456 after 10:00 AM, Jan 1, 2025."
2. **Step 2: Session Initiation**
    - Alice's Lighter.IM extension (Prover) contacts the Verifier Network.
3. **Step 3: Proxied TLS Handshake & Certificate Validation**
    - The Verifier and Prover collaboratively initiate a TLS handshake with `bank.com` (Data Holder).
    - The Verifier validates the `bank.com` TLS certificate, ensuring its authenticity.
4. **Step 4: Key Split & Login**
    - Prover and Verifier collaboratively compute the session key (Prover gets K1, Verifier gets K2).

- Prover submits their encrypted login request through the Verifier.

5. **Step 5: Encrypted Stream & Witness Signature**
    - `bank.com` validates the credentials and sends back the **encrypted data (Enc_Data)** containing the transfer history.
    - The Verifier cannot decrypt this data but computes `H(Enc_Data)` and generates `Sig(H(Enc_Data))`.
    - The Verifier forwards `Enc_Data` and its signature to the Prover.
6. **Step 6: Local Decryption & Proof Generation**
    - The Prover's extension locally decrypts the data stream using K1 and K2, yielding the plaintext HTML.
    - The extension parses the HTML to find the record matching the claim ("$100 to Bob 123456...").
    - The extension executes the ZK circuit, feeding in the (private) plaintext and the (public) signature, hash, and claim. This outputs the ZK Proof (Proof.data).
7. **Step 7: Proof Verification & Execution**
    - The Prover submits the ZK Proof to the ZKP Verification Network (ZKPNet).
    - ZKPNet efficiently verifies the proof (checking both the Verifier's signature and the ZK circuit logic).
    - ZKPNet aggregates the verification result and makes it available to the target chain (e.g., via a Merkle root or a cross-chain message).
    - Alice submits the final transaction to the Lighter.IM contract, referencing the valid proof.
    - The contract confirms the proof's validity and automatically releases the escrowed 100 USDC to Bob. The trade is complete.

5. Conclusion and Future Outlook

The Lighter.IM protocol, by combining the zkTLS proxy model with Zero-Knowledge Proofs, successfully creates a trustless and privacy-preserving decentralized credential system. This architecture not only solves the core trust problem of fiat on/off-ramps but also offers a general-purpose solution that can be extended to any Web3 scenario requiring Web2 data credentials, such as credit lending, digital identity, and on-chain reputation.

By outsourcing the heavy ZKP verification workload to a specialized network, this protocol achieves economic viability and efficient on-chain interaction while maintaining the highest level of security. We believe the "Off-Chain Witnessing + Local Proving + On-Chain Verification" paradigm demonstrated by Lighter.IM will

become a standard for trusted data interoperability between the Web2 and Web3 worlds.