The PEACE Protocol[1]

A protocol for transferable encryption rights.

Logical Mechanism LLC[2]

December 16, 2025

[1]This project was funded in Fund 14 of Project Catalyst.
[2]Contact: support@logicalmechanism.io

Contents

1 Abstract **1**

2 Introduction **1**

3 Background And Preliminaries **2**

4 Cryptographic Primitives Overview **3**
4.1 Register-based 3
4.2 ECIES + AES-GCM 4
4.3 Re-Encryption 5

5 Protocol Overview **6**
5.1 Design Goals And Requirements 6
5.2 On-Chain And Off-Chain Architecture 7
5.3 Key Management And Identity 9
5.4 Protocol Specification 10

6 Security Model **15**
6.1 Trust Model 15

7 Threat Analysis **15**
7.1 Metadata Leakage 15

8 Limitations And Risks **15**
8.1 Performance And On-Chain Cost 15

9 Conclusion **15**

A Appendix A - Proofs **16**

Bibliography **17**

1 Abstract

In this report, we introduce the PEACE protocol, an ECIES-based, multi-hop, unidirectional proxy re-encryption scheme for the Cardano blockchain. PEACE solves the encrypted-NFT problem by providing a decentralized, open-source protocol for transferable encryption rights, enabling creators, collectors, and developers to manage encrypted NFTs without relying on centralized decryption services. This work fills a significant gap in secure, private access to NFTs on Cardano. Project Catalyst[1] funded the PEACE protocol in round 14.

2 Introduction

The encrypted NFT problem is one of the most significant issues with current NFT standards on the Cardano blockchain. Either the data is not encrypted, available to everyone who views the nft, or the data encryption requires some form of centralization, with some company doing the encryption on behalf of users. Current solutions [1] claim to offer decentralized encrypted assets (DEA), but lack a publicly available, verifiable cryptographic protocol or an open-source implementation. Most, if not all, of the mechanics behind current DEA solutions remain undisclosed. This report aims to fill that knowledge gap by providing an open-source implementation of a decentralized re-encryption protocol for encrypted assets on the Cardano blockchain.

Several mandatory requirements must be satisfied for the protocol to function as intended. The encryption protocol must allow tradability of both the NFT itself and the right to decrypt the NFT data, implying that the solution must involve smart contracts and a form of encryption that allows data access to be re-encrypted for another user without revealing the encrypted content in the process. The contract side of the protocol should be reasonably straightforward. It needs a way to trade a token that holds the encrypted data and allows other users to receive it. To ensure decryptability, the tokens will need to be soulbound. On the encryption side of the protocol is some form of encryption that enables the re-encryption process to function correctly. Luckily, this type of encryption has been in cryptography research for quite some time [2] [3] [4]. There are even patented cloud-based solutions already in existence [5]. There is no open-source, fully on-chain, decentralized re-encryption protocol for encrypting NFT data on the Cardano blockchain. The PEACE protocol aims to provide a proof-of-concept solution to this problem.

The PEACE protocol will implement an ambitious yet well-defined, unidirectional, multi-hop proxy re-encryption scheme that utilizes ECIES [6] and AES [7]. Unidirectionality means that Alice can re-encrypt for Bob, and Bob can then re-encrypt it back to Alice, using different encryption keys. Unidirectionality is important for tradability, as it defines the one-way flow of data and removes any restriction on who can purchase the NFT. Multi-hop means that the flow of encrypted data from Alice to Bob to Carol, and so on, does not end, in the sense that it cannot be re-encrypted for a new user. Multi-hopping is important for tradability, as a finitely tradable asset does not fit many use cases. Typically, an asset should always be tradable if the user wants to trade it. The encryption primitives used in the protocol are considered industry standards at the time of this report.

The remainder of this report is as follows. Section 4 discusses the preliminaries and background required for this project. Section 5 will be a brief overview of the required cryptographic primitives. Section 6 will be a detailed description of the protocol. Sections 7, 8, and 9 will delve into security and threat analysis, the limitations of the protocol, and related topics, respectively. The goal of this report is to serve as a comprehensive reference and description of the PEACE protocol.

[1] https://projectcatalyst.io/funds/14/cardano-use-cases-concepts/decentralized-on-chain-data-encryption

3 Background And Preliminaries

Understanding the protocol will require some technical knowledge of modern cryptographic methods, a basic understanding of elliptic curve arithmetic, and a general understanding of smart contracts on the Cardano blockchain. Anyone comfortable with these topics will find this report very useful and easy to follow. The report will attempt to use research standards for terminology and notation. The elliptic curve used in this protocol will be BLS12-381 [8]. All smart contracts required for the protocol are written in Aiken.

Table 1: Symbol Description [9]

Symbol	Description
p	A prime number
$\mathbb{F}_p$	The finite field with characteristic p
$E(\mathbb{F}_p)$	An elliptic curve E defined over $\mathbb{F}_p$
E'	A twisted elliptic curve
$\#E(\mathbb{F}_p)$	The order of $E(\mathbb{F}_p)$ (also denoted n)
r	A prime number dividing $\#E(\mathbb{F}_p)$
δ	A non-zero integer in $\mathbb{Z}_n$
$\mathcal{O}$	The point at infinity of an elliptic curve E
$\mathbb{G}_1$	A subgroup of order r of $E(\mathbb{F}_p)$
$\mathbb{G}_2$	A subgroup of order r of the twist $E'(\mathbb{F}_{p^2})$
$\mathbb{G}_T$	The multiplicative target group of the pairing: $\mu_r \subset \mathbb{F}_{p^{12}}$
$e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$	A type-3 bilinear pairing
g	A fixed generator in $\mathbb{G}_1$
q	A fixed generator in $\mathbb{G}_2$
R	The Fiat-Shamir transformer
H_κ	A hash to group function for $\mathbb{G}_\kappa$
m	The order of Ed25519
γ	A non-zero integer in $\mathbb{Z}_m$

The protocol, including both on-chain and off-chain components, will heavily utilize the `Register` type. The `Register` stores a generator, $g \in \mathbb{G}_\kappa$ and the corresponding public value $u = [\delta]g$ where $\delta \in \mathbb{Z}_n$ is a secret. We shall assume that the hardness of ECDLP and CDH in $\mathbb{G}_1$ and $\mathbb{G}_2$ will result in the inability to recover the secret $\delta \in \mathbb{Z}_n$. When using a pairing, we additionally rely on the standard bilinear Diffie-Hellman assumptions over the subgroups ($\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$). We will represent the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ with additive notation and $\mathbb{G}_T$ with multiplicative notation.

The `Register` type in Aiken:

Where required, we will verify Ed25519 signatures [10] as a cost-minimization approach; relying solely on pure BLS12-381 for simple signatures becomes costly on-chain. There will be instances where the Fiat-Shamir transform [11] will be applied to a Σ-protocol for non-interactive purposes. In these cases, the hash function will be Blake2b-224 [12].

```
pub type Register {
  // the generator, #<Bls12_381, G1> or #<Bls12_381, G2>
  generator: ByteArray,
  // the public value, #<Bls12_381, G1> or #<Bls12_381, G2>
  public_value: ByteArray,
}
```

Listing 1: The Register type

4 Cryptographic Primitives Overview

This section provides brief explanations of the cryptographic primitives required by the protocol. If a primitive has an algorithmic description, then it should be included in the respective section. The `Register` type will be a tuple, $(\ g, u\ )$, for simplicity inside the algorithms. We shall assume that the decompression of the elliptic curve points are a given. Correctness proofs for many algorithms are in Appendix A. In any algorithm, $\mathbb{G}_1$ may be switched with $\mathbb{G}_2$ without any required changes.

4.1 Register-based

The protocol requires proving knowledge of a user's secret using a Schnorr Σ-protocol [13] [14]. This algorithm is both complete and zero-knowledge. We can use simple Ed25519 signatures for spendability, and then utilize the Schnorr Σ-protocol for knowledge proofs related to encryption. We will make the protocol non-interacting via the Fiat-Shamir transform.

Algorithm 1: Non-interactive Schnorr's Σ-protocol for the discrete logarithm relation

Input:
$(\ g, u\ )$ where $g \in \mathbb{G}_\kappa$, $u = [\delta]g \in \mathbb{G}_\kappa$
Output: bool

1 select a random $\delta' \in \mathbb{Z}_n$
2 compute $a = [\delta']g$
3 calculate $c = R(g, u, a)$
4 compute $z = \delta * c + \delta'$
5 output $[z]g = a + [c]u$

The protocol requires proving a binding relationship between a user's public value and other known encryption related elliptic curve elements. The binding proof is multiple Schnorr proofs for various informaiton.

Algorithm 2: Non-interactive Binding Σ-protocol

Input:
(g, u) where $g \in \mathbb{G}_1$, $u = [\delta]g \in \mathbb{G}_1$
(r_1, χ) where $r_1 \in \mathbb{G}_1$, $\chi \in \mathbb{G}_1$
(a, r) where $a \in \mathbb{Z}_n$ and $r \in \mathbb{Z}_n$
Output: bool

select a random $\rho \in \mathbb{Z}_n$ and $\alpha \in \mathbb{Z}_n$
compute $t_1 = [\rho]g$
compute $t_2 = [\alpha]g + [\rho]u$
calculate $c = R(g, u, t_1, t_2)$
compute $z_a = a * c + \alpha$
compute $z_r = r * c + \rho$
output $[z_a]g + [z_r]u = t_2 + [c]\chi$ AND $[z_r]g = t_1 + [c]r_1$

There will be times when the protocol requires proving some equality using a pairing. In these cases, we can use something akin to the BLS signature, allowing only someone with the knowledge of the secret to prove the pairing equivalence. BLS signatures are a straightforward yet important signature scheme for the protocol, as they enable public confirmation of knowledge of a complex relationship beyond the limitations of Schnorr's Σ-protocol. BLS signatures work because of the bilinearity of the pairing [15].

Algorithm 3: Boneh-Lynn-Shacham (BLS) signature method

Input:
(g, u, c, w, m) where $g \in \mathbb{G}_1$, $u = [\delta]g \in \mathbb{G}_1$,
$c = H_2(m) \in \mathbb{G}_2$, $w = [\delta]c \in \mathbb{G}_2$, and $m \in \{0, 1\}^*$
Output: bool

$e(u, c) = e(g, w)$
$e(q^\delta, c) = e(q, c^\delta) = e(q, c)^\delta$

4.2 ECIES + AES-GCM

The Elliptic Curve Integrated Encryption Scheme (ECIES) is a hybrid protocol involving asymmetric cryptography with symmetric ciphers. The encryption used in ECIES is the Advanced Encryption Standard (AES). ECIES and AES, combined with a key derivation function (KDF) such as HKDF [16], form a complete encryption system.

Algorithm 4: Encryption using ECIES + AES

Input:
(g, u) where $g \in \mathbb{G}_\kappa$, $u = [\delta]g \in \mathbb{G}_\kappa$, m as the message
Output: (r, c, h)

select a random $\delta' \in \mathbb{Z}_n$
compute $r = [\delta']g$
compute $s = [\delta']u$
generate $k = KDF(s|r)$
encrypt $c = AES(m, k)$
compute $h = BLAKE2B(m)$
output (r, c, h)

Decrypting the ciphertext requires rebuilding the data encryption key (DEK), k, from the KDF. The DEK is rebuildable because r is public and the user knows the secret δ, allowing them to decrypt the data.

Algorithm 5: Decryption using ECIES + AES

Input:
(g, u) where $g \in \mathbb{G}_1$, $u = [\delta]g \in \mathbb{G}_1$, (r, c, h) as the cypher text
Output: ($\{0,1\}^*$,bool)

compute $s' = [\delta]r$
generate $k' = KDF(s'|r)$
compute $m' = AES(c, k')$
compute $h' = BLAKE2B(m')$
output (m', h' = h)

Algorithm 4 describes the case where a `Register` is used to generate the DEK, k, from the KDF function. Anyone with knowledge of k may decrypt the ciphertext. The algorithm shown differs slightly from the PEACE protocol, as the protocol allows transferring k to another `Register`; however, the general flow remains the same. The key takeaway here is that encrypting a message and decrypting the ciphertext requires a KDF to generate the DEK. Both algorithms 4 and 5 use a simple hash function for authentication. In the PEACE protocol, we will use AES-GCM with authenticated encryption with associated data (AEAD) for authentication.

4.3 Re-Encryption

There are various types of re-encryption schemes, ranging from classical proxy re-encryption to hybrid methods. These re-encryption schemes involve a proxy, an entity that performs the re-encryption and verification processes. The PRE used in the PEACE protocol is modeled as an interactive flow between the current owner and a prospective buyer, utilizing a smart contract as part of the proxy. We need an interactive scheme because in many real-world use cases, there are numerous off-chain checks, such as KYC/AML and various legal requirements, that must occur before transferring the decryption rights to the new owner. The PEACE protocol obtains interactivity via a bidding system and having the owner agreeing to the exchange.

The method described below is a hybrid approach. The current owner's wallet performs the re-encryption process for the buyer. At the same time, the Cardano smart contract acts as a proxy, verifying various cryptographic proofs, enforcing the correct bindings, handling payments, and updating the on-chain owner field. This design explicitly supports off-chain processes, such as KYC or contractual agreements, before delegation: the owner only submits the re-encryption transaction once these off-chain conditions are satisfied. This method will allow for the most use cases for real-world assets. This type of method is unidirectional, meaning the re-encryption flow is one-way: from the current owner to the next owner. If Alice delegates to Bob, Bob does not automatically gain the ability to 'go backwards' and create ciphertexts for Alice using the same re-encryption material. This flow differs from a bidirectional method, where the PRE is symmetric, enabling a two-way encryption relationship between the parties. So, Alice can transform a ciphertext into one for Bob, and Bob can transform a ciphertext into one for Alice, without either Alice or Bob having to re-run the entire re-encryption flow. That is not what we want for this implementation. Each direction is a separate, explicit delegation with its own re-encryption material, matching the tradability requirements.

Note that in the original Catalyst proposal, the protocol defines itself as a bidirectional, multi-hop PRE. However, during the design phase, it became clear that the actual Cardano use case requires a unidirectional, multi-hop PRE. This change is fully compatible with the original proposal's PRE goals (transfer of decryption rights without exposing plaintext or private keys), but reflects the reality of trading tokens via Cardano smart contracts within the PRE landscape.

Algorithm 6: Owner-mediated re-encryption from Alice to Bob

Input:
(g, u) where $g \in \mathbb{G}_1$, $u = [\delta_a]g \in \mathbb{G}_1$ (Alice's public key),
(g, v) where $v = [\delta_b]g \in \mathbb{G}_1$ (Bob's public keys),
Alice's secret key $\delta_a \in \mathbb{Z}_n$
$(r_{1,a}, r_{2,a}, r_{3,a})$, where $r_1 \in \mathbb{G}_1$, $r_2 \in \mathbb{G}_T$, and $r_3 \in \mathbb{G}_2$
(h_0, h_1, h_2, h_3), where $h_i \in \mathbb{G}_2$ are public points.
Output: $(r_{1,b}, r_{2,b}, r_{3,b})$ and $(r'_{1,a}, r'_{2,a}, r'_{3,a})$

1 select a random $\kappa \in \mathbb{G}_T$, $\kappa = e(q^a, h_0)$
2 select a random $r \in \mathbb{Z}_n$
3 compute $r_{1,b} = [r]g$
4 compute $r_{2,b} = e(q^a, h_0) * e(v^r, h_0) = e(q^a v^r, h_0)$
5 compute $c = [BLAKE2b(r_{1,b})]h_1 + [BLAKE2b(r_{1,b}||r_{2,b})]h_2$
6 compute $r_{4,b} = [r]c$
7 compute $r_{5,b} = [BLAKE2b(\kappa)]p + [\delta_a]h_0$
8 update $r'_{2,a} = r_{2,a} * e(r_{1,a}, r_{5,b})$
9 output $(r_{1,b}, r_{2,b}, r_{4,b})$ and $(r_{1,a}, r'_{2,a}, r_{3,a})$

Algorithm 6 describes the actual re-encryption process for Alice, giving the decryption rights to Bob. Bob can then use this information to recursive calculate the secret κ and eventually the original secret used in the encryption process.

5 Protocol Overview

The PEACE protocol is an ECIES-based, multi-hop, unidirectional proxy re-encryption scheme for the Cardano blockchain, allowing creators, collectors, and developers to trade encrypted NFTs without relying on centralized decryption services. The protocol should be viewed as a proof of concept, as the data storage layer for the protocol is the Cardano blockchain; thus, ultimately, the storage limit, the maximum size of the encrypted data and required decryption data, is bound by the current parameters of the Cardano blockchain. In a production setting, the data storage layer should allow for arbitrary file sizes.

5.1 Design Goals And Requirements

Two equally important areas, the on-chain and off-chain, define the protocol design. The on-chain design is everything related to smart contracts written in Aiken for the Cardano blockchain. The off-chain design includes transaction building, cryptographic proof generation, and the happy path flow. The design on both sides will focus on a two-party system: Alice and Bob, who want to trade encrypted data. Alice will be the original owner, and Bob will be the new owner. As this is a proof of concept, the protocol will not include the general n-party system, as that is future work for a real-world production setting.

```
pub type BidDatum {
  owner_vkh: VerificationKeyHash,
  owner_g1: Register,
  pointer: AssetName,
  token: AssetName,
}
```

Listing 2: The Bid datum type

The protocol must allow continuous trading via a multi-hop pre, meaning that Alice will trade with Bob, who could then trade with Carol. In this setting, Alice will trade to Bob then Bob will trade back to Alice rather than to Carol without any loss of generality. Each hop will generate new owner and decryption data. The storage of previous hop data should grow at most linearly. Users will use a basic bid system for token trading. A user may choose to not trade their token by simply not selecting a bid if one exists.

The encryption direction needs to flow in one direction per hop. Alice trades with Bob, and that is the end of their transaction. Bob does not gain any ability to re-encrypt the data back to Alice without a new bid made by Alice, restarting the re-encryption process. Any bidirectionality here implies symmetry between Alice and Bob, thereby circumventing the re-encryption requirement via token trading. The unidirectional requirement forces tradability to follow the typical trading interactions currently found on the Cardano blockchain.

Each UTxO in this system must be uniquely identified via an NFT. The uniqueness requirement works well for the encryption side because the NFT could be a tokenized representation of the encrypted data, something akin to a CIP68 [17] contract, but using a single token. The bid side does work, but the token becomes a pointer rather than having any real data attached, essentially a unique, one-time-use token. Together, they provide the correct uniqueness requirement. UTxOs may be removed from the contract at any time by the owner. After the trade, the owner of the encrypted data may do whatever they want with that data. The protocol does not require the re-encryption contract to store the encrypted data permanently.

The protocol will use an owner-mediated re-encryption flow (a hybrid PRE), which is UX-equivalent to a classical proxy re-encryption scheme in this setting, since smart contracts on Cardano are passive validators and do not initiate actions. Ultimately, a user must act as the proxy, the one doing the re-encryption, because the contract cannot do it on its own. The smart contract must act as the proxy's validator, not solely as the proxy itself. To simplify this proof of concept implementation, the owner will act as their own proxy in the protocol.

5.2 On-Chain And Off-Chain Architecture

There will be two user-focused smart contracts: one for re-encryption and the other for bid management. Any UtxO inside the re-encryption contract is for sale via the bidding system. A user may place a bid into the bid contract, and the current owner of the encrypted data may select it as payment for re-encrypting the data to the new owner. To ensure functionality, a reference data contract must exist, as it resolves circular dependencies. The reference datum will contain the contract script hashes for the re-encryption and bid contracts.

The bid contract datum structure is defined in Listing 2. The bid datum contains all of the required information for re-encryption. The owner of a bid UTxO will be type `Register` in $\mathbb{G}_1$. The `pointer`

```
pub type BidMintRedeemer {
  EntryBidMint(SchnorrProof)
  LeaveBidBurn(AssetName)
}
pub type BidSpendRedeemer {
  RemoveBid
  UseBid
}
```

Listing 3: The Bid redeemer types

```
pub type EncryptionDatum {
  owner_vkh: VerificationKeyHash,
  owner_g1: Register,
  token: AssetName,
  levels: List<EncryptionLevel>,
  capsule: Capsule,
}
pub type Capsule {
  nonce: ByteArray,
  aad: ByteArray,
  ct: ByteArray,
}
pub type EncryptionLevel {
  r1b: ByteArray,
  r2: EmbeddedGt,
  r4b: ByteArray,
}
pub type EmbeddedGt {
  g1b: ByteArray,
  g2b: Option<ByteArray>,
}
```

Listing 4: The Encryption datum type

is the NFT name on the bid UTxO, and `token` is the NFT name on the re-encryption UTxO. The `token` forces the bid to apply only to a specific encrypted data sale.

The bid contract redeemer structures are defined in Listing 3. Entering into the bid contract uses the `EntryBidMint` redeemer, triggering a `pointer` mint validation, a `token` existence check, a Ed25519 signature with `owner_vkh`, and a Schnorr Σ-protocol using `owner_g1`. Leaving the bid contract requires using `RemoveBid` and `LeaveBidBurn` redeemers together, triggering a `pointer` burn validation and Ed25519 signature with `owner_vkh`. When a user selects a bid, they will use `UseBid` and `LeaveBidBurn` together, triggering a `pointer` burn validation and the proxy re-encryption validation.

The re-encryption contract datum structure is defined in Listing 4. The ciphertext and related data are held in the `capsule` sub-type and each hop generates a new encryption level sub-type. We can't store or do full arithmetic on $\mathbb{G}_T$ elements on-chain, and storing extra group elements is expensive. So `EmbeddedGt` stores only the minimal factors needed to reconstruct the $\mathbb{G}_T$ elements

```
pub type EncryptionMintRedeemer {
  EntryEncryptionMint(SchnorrProof, BindingProof)
  LeaveEncryptionBurn(AssetName)
}
pub type EncryptionSpendRedeemer {
  RemoveEncryption
  UseEncryption(ByteArray, ByteArray, AssetName, BindingProof)
}
pub type SchnorrProof {
  z_b: ByteArray,
  g_r_b: ByteArray,
}
pub type BindingProof {
  z_a_b: ByteArray,
  z_r_b: ByteArray,
  t_1_b: ByteArray,
  t_2_b: ByteArray,
}
```

Listing 5: The Encryption redeemer types

during validation, while everything else is treated as an implied constant or a value that can be referenced elsewhere.

The re-encryption datum contains all of the required information for decryption. The owner of an encrypted data UTxO will be type `Register` in $\mathbb{G}_1$. The `token` is the NFT name on the re-encryption UTxO. The `capsule` contains the encryption information and `levels` contains the decryption information. Inside the `capsule` is the `nonce`, `aad`, and `ct`.

The re-encryption contract redeemer structures are defined in Listing 5. Entering into the re-encryption contract uses the `EntryEncryptionMint` redeemer, triggering a `token` mint validation, a Ed25519 signature with `owner_vkh`, a binding proof using `owner_g1` and a Schnorr Σ-protocol using `owner_g1`. Leaving the re-encryption contract requires using `RemoveEncryption` and `LeaveEncryptionBurn` redeemers together, triggering a `token` burn validation and a Ed25519 signature with `owner_vkh`. When a user selects a bid, they will use `UseEncryption`, triggering the proxy re-encryption validation.

The redeemers `UseEncryption`, `UseBid`, and `LeaveBidBurn` must be used together during re-encryption.

5.3 Key Management And Identity

Each user in the protocol has the ability to deterministically generate BLS12-381 keypairs represented by `Register` value in $\mathbb{G}_1$. The $\mathbb{G}_1$ points are used as the user's on-chain identity for encryption and signature verification. The corresponding secret scalar $\delta \in \mathbb{Z}_n$ is held off-chain by the user's wallet or client software and is never published on-chain.

The BLS12-381 keys used for re-encryption are logically separate from the Ed25519 keys used to sign Cardano transactions. A wallet must manage both: Ed25519 keys to authorize UTxO spending, and BLS12-381 scalars to obtain and delegate decryption rights. Losing or compromising the BLS12-381 secret key means losing the ability to decrypt any items associated with that identity, even if the Cardano spending keys are still available.

```
pub fn generate_token_name(inputs: List<Input>) -> AssetName {
  let input: Input = builtin.head_list(inputs)
  let id: TransactionId = input.output_reference.transaction_id
  let idx: Int = input.output_reference.output_index
  id |> bytearray.push(idx) |> bytearray.slice(0, 31)
}
```

Listing 6: Token name generation

The proof-of-concept does not implement a full key rotation or revocation mechanism. If a user's BLS12-381 secret key is compromised, an attacker can decrypt all current and future capsules addressed to that key, but cannot retroactively remove or alter on-chain history. Handling key rotation, partial recovery, and revocation across many encrypted positions is left as future work for a real-world production deployment.

For each encrypted item, the protocol generates a fresh symmetric key for AES-GCM encryption of the actual payload. This key is not stored on-chain. The on-chain capsule contains the AES-GCM nonce, associated data, and ciphertext.

5.4 Protocol Specification

The protocol flow starts with Alice selecting a secret $[\gamma] \in \mathbb{Z}_m$ and $[\delta] \in \mathbb{Z}_n$. The secret γ will generate a Ed25519 keypair that will in turn generate the `VerificationKeyHash`, `vkh`, used on-chain in the Ed25519 signatures. The secret δ will generate the `Register` in $\mathbb{G}_1$ using the fixed generator, g. Alice will fund the address associated with `vkh` with enough Lovelace to pay for the minimum required Lovelace for both the change and contract UTxO, and the transaction fee. Alice may then build the entry to the re-encryption contract transaction.

The re-encryption entry transaction will contain a single input and two outputs. The transaction will mint a `token` using the `EntryEncryptionMint` redeemer. The `token` name is generated by the concatentation of the input's output index and transaction id as shown in the Listing 6. The specification for the protocol assumes a single input but in general many inputs may be used in this transaction. If more than one input exists then the first lexicographical sorted input will be used for the name generation.

The example below shows the token name generation output.

```
input = "1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef#24"
token_name = "181234567890abcdef1234567890abcdef1234567890abcdef1234567890abcd"
```

Alice may now finish building the `EncryptionDatum` by constructing the `levels` and `capsule`. Since Alice is the first owner, she will encrypt to herself. Alice will encrypt the original data by generating a root κ in $\mathbb{G}_T$. The root secret will be used in the KDF to produce a valid AES key. The message can now be encrypted using AES-GCM. The resulting information is stored in the `Capsule` type. Below is a pythonic psuedocode for generating the original encrypted data and the first encryption level.

The sub-types can be populated as shown in Listing 8. The contract will validate the first encryption level using the assertion from Listing 9. Alice can prove to herself that the encryption level is valid by verifying the assertion in Listing 10. Alice may now construct the full `EncryptionDatum` as shown in Listing 11.

```
message = "This is a secret message."

# generate random data for the first encryption level
a0 = rng()
r0 = rng()
k0 = random_fq12(a0)

# alice as a register, sk is the secret key
alice = Register(sk)

# generate the r terms
r1b = scale(g, r0)
r2_g1b = scale(g, a0 + r0*sk)

a = to_int(blake2b(r1b))
b = to_int(blake2b(r1b + r2_g1b))

c = combine(combine(scale(H1, a), scale(H2, b)), H3)
r4b = scale(c, r0)

# encrypt the message
nonce, aad, ct = encrypt(r1, k0, message)
```

Listing 7: Creating the Encrypted data and first encryption level

```
pub type EncryptionLevel {
  r1b,
  r2: EmbeddedGt {
    g1b: r2_g1b,
    g2b: None,
  },
  r4b,
}
pub type Capsule {
  nonce,
  aad,
  ct: ciphertext,
}
```

Listing 8: Encryption data format

```
assert pair(g, r4b) = pair(r1b, c)
```

Listing 9: First level validation

```
expected_k0 = pair(r2_g1b, H0) / pair(r1b, scale(H0, sk))
assert k0 == expected_k0
```

Listing 10: Alice can decrypt the key

```
pub type EncryptionDatum {
  owner_vkh,
  owner_g1,
  token: generate_token_name(inputs),
  levels: [
    EncryptionLevel {
      r1b,
      r2: EmbeddedGt {
        g1b: r2_g1b,
        g2b: None,
      },
      r4b,
    }
  ],
  capsule: Capsule {
    nonce,
    aad,
    ct: ciphertext,
  },
}
```

Listing 11: Full first level datum

The entry redeemer verifies that Alice's verification key (Ed25519 key hash) is valid via a simple Ed25519 signature as Alice needs a valid `vkh` to be able to remove the entry. The entry redeemer also verifies a valid `Register` via a Schnorr Σ-protocol 1 as Alice needs this to decrypt her own data and it verifies that Alice binds her public value to the first encryption level via a binding proof 2. After successfully creating a valid entry transaction and submitting it to the Cardano blockchain the encrypted data is ready to be traded.

Bob may now place a bid into the bid contract in an attempt to purchase the encrypted data from Alice. First, Bob selects a secret $[\gamma] \in \mathbb{Z}_m$ and $[\delta] \in \mathbb{Z}_n$. Similarily to Alice, the secret γ will generate a Ed25519 keypair that will in turn generate the `VerificationKeyHash`, `vkh`, used on-chain in the Ed25519 signatures. The secret δ will generate the `Register` in $\mathbb{G}_1$ using the fixed generator, g. Bob will fund the address associated with `vkh` with enough Lovelace to pay for the change and the contract UTxO, and the transaction fee. Bob may then build the entry to the bid contract transaction. Note that the protocol grows linearly thus the required Lovelace for some given encrypted message will increase over time, meaning Bob should contribute to the minimum required Lovelace for the encrypted data though this is not required on-chain.

The structure of the bid entry transaction is similar to the re-encryption entry transaction but using `EntryBidMint` instead of `EntryEncryptionMint`. The `pointer` token name is generated in the exact some way as the `token` name.

Now, Bob can create the `BidDatum`. The `token` name may be referenced on-chain from the re-encryption contract and the `pointer` is derived from the inputs as shown in Listing 12.

Similar to the re-encryption contract, the entry redeemer will verify Bob's `vkh` and the `Register` values in $\mathbb{G}_1$. This is important as the validity of these points will determine if Bob can decrypt the data after the re-encryption process. The value of the UTxO is price Bob is willing to pay for Alice to re-encrypt the data to his `Register`. There may be many bids but only one can be selected by

```
pub type BidDatum {
  owner_vkh,
  owner_g1,
  pointer: generate_token_name(inputs),
  token,
}
```

Listing 12: Full Bid datum

```
a1 = rng()
r1 = rng()
k1 = random_fq12(a1)

hk = to_int(k1)

r1b = scale(g, r1)
r2_g1b = combine(scale(g, a1), scale(bob_public_value, r1))

a = to_int(generate(r1b))
b = to_int(generate(r1b + r2_g1b))
c = combine(scale(H1, a), scale(H2, b))
r4b = scale(c, r1)

r5b = combine(scale(q, hk), scale(invert(H0), sk))
```

Listing 13: Generate the next level

Alice for the re-encryption transaction. For simplicity of the protocol, Bob will need to remove their old bids and recreate the bids for any nessecary price adjustments. Bob may remove his bid at any time.

Alice will select a bid UTxO from the bid contract and will do the re-encryption process using Bob's `Register` data. This step requires Alice to burn Bob's bid token, update the on-chain data to Bob's data, and create re-encryption proofs. This is the most important step as this is the tradability of both the token and the encrypted data. The re-encryption redeemer will provide all of the required proxy validation proofs. The PRE proofs are pairings between the original owner's `Register` values in $\mathbb{G}_1$, proving that the new owner's `Register` was used during the re-encryption process, resulting in a transfer of ownership and decryption rights. Listing 13 is a pythonic psuedocode for generating the next encryption level.

Bob's and Alice's encryption levels are shown in Listing 14. The complete next encryption datum is shown in Listing 15.

The contract will validate the re-encryption using a binding proof and two pairing proofs as shown in Listing 16. The first assertion follows the Alice's validation, ensuring that the encryption level terms are consistent. The second assertion hows that Alice create the r_5 term correctly. Adding a SNARK for valid witness creation is left for later work.

Bob can now decrypt the root key by recursiving computing all the random $\mathbb{G}_T$ points as shown in Listing 17.

```
pub type EncryptionLevel {
  r1b,
  r2: EmbeddedGt {
    g1b: r2_g1b,
    g2b: None,
  },
  r4b,
}
pub type EncryptionLevel {
  r1b: alice.r1B,
  r2: EmbeddedGt {
    g1b: alice.r2_g1b,
    g2b: Some(r5b),
  },
  r4b: alice.r4b,
}
```

Listing 14: Bob's and Alice's encryption levels

```
pub type EncryptionDatum {
  owner_vkh: bob.owner_vkh,
  owner_g1: bob.owner_g1,
  token,
  levels: [
    EncryptionLevel {
      r1b,
      r2: EmbeddedGt {
        g1b: r2_g1b,
        g2b: None,
      },
      r4b,
    },
    EncryptionLevel {
      r1b: alice.r1B,
      r2: EmbeddedGt {
        g1b: alice.r2_g1b,
        g2b: Some(r5b),
      },
      r4b: alice.r4b,
    }
  ],
  capsule: Capsule {
    nonce,
    aad,
    ct: ciphertext,
  },
}
```

Listing 15: Bob's encryption datum

```
assert pair(g, bob.r4b) = pair(bob.r1b, c)
assert pair(g, alice.r5b) * pair(alice.u, H0) = pair(alice.witness, p)
```

Listing 16: Validate the re-encryption process

```

h0x = scale(H0, sk)
shared = h0x

for entry in encryption_levels:
    r1 = entry.r1

    if is_half_level(entry.r2):
        r2 = pair(entry.r2.g1, H0)
    else:
        r2 = pair(entry.r2.g1, H0) * pair(r1, entry.r2.g2)

    b = pair(r1, shared)
    key = fq12_encoding(r2 / b, F12_DOMAIN_TAG)
    k = to_int(key)
    shared = scale(q, k)

message = decrypt(r1, key, capsule.nonce, capsule.ct, capsule.aad)
```

Listing 17: Decrypting the secret message

6 Security Model

6.1 Trust Model

6.1.1 Assumptions

7 Threat Analysis

7.1 Metadata Leakage

8 Limitations And Risks

8.1 Performance And On-Chain Cost

9 Conclusion

A Appendix A - Proofs

Lemma A.1. *Correctness for Algorithm 1, a non-interactive Schnorr's Σ-protocol for the discrete logarithm relation.*

Proof. We start with (g, u, a, z) where $g \in \mathbb{G}_1$, $u = [\delta]g \in \mathbb{G}_1$, $a \in \mathbb{G}_1$, and $z \in \mathbb{Z}_n$. Let us assume that $z = r + c * \delta$ and $a = [r]g$.

Use the Fiat-Shamir transform to generate a challenge value $c = R(g, u, a)$.

$[z]g = [r + c * x]g$

$[z]g = [r]g + [c][x]g$

$[z]g = a + [c]u$

An honest `Register` can produce an a and z that will satisfy $[z]g = a + [c]u$ proving knowledge of the secret */delta*.

□

Bibliography

[1] J. Stone, "Stuff.io whitepaper 1.0." Accessed: Oct. 25, 2025. [Online]. Available: https://book-io.medium.com/stuff-io-whitepaper-1-0-9529db7cdeaf

[2] M. Mambo and E. Okamoto, "Proxy cryptosystems: Delegation of the power to decrypt ciphertexts," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E80–A, no. 1, pp. 54–63, 1997, Available: https://search.ieice.org/bin/summary.php?id=e80-a_1_54

[3] M. Blaze, G. Bleumer, and M. Strauss, "Divertible protocols and atomic proxy cryptography," in *EUROCRYPT 1998*, Springer, 1998. doi: 10.1007/BFb0054122.

[4] G. Ateniese, K. Fu, M. Green, and S. Hohenberger, "Improved proxy re-encryption schemes with applications to secure distributed storage," in *NDSS 2005*, 2005. Available: https://www.ndss-symposium.org/ndss2005/improved-proxy-re-encryption-schemes-applications-secure-distributed-storage/

[5] IronCore Labs, *Recrypt (rust): Transform/proxy re-encryption library*. Accessed: Oct. 25, 2025. [Online]. Available: https://github.com/IronCoreLabs/recrypt-rs

[6] *IEEE standard specifications for public-key cryptography—amendment 1: Additional techniques*. IEEE, 2004.

[7] *Advanced encryption standard (AES)*. NIST, 2001.

[8] S. Bowe, "BLS12-381: New zk-SNARK elliptic curve construction." Accessed: Oct. 25, 2025. [Online]. Available: https://electriccoin.co/blog/new-snark-curve/

[9] N. El Mrabet and M. Joye, Eds., *Guide to pairing-based cryptography*. in Chapman & hall/CRC cryptography and network security. New York: Chapman; Hall/CRC, 2017, p. 420. doi: 10.1201/9781315370170.

[10] S. Josefsson and I. Liusvaara, *Edwards-curve digital signature algorithm (EdDSA)*. IETF, 2017. Available: https://www.rfc-editor.org/rfc/rfc8032

[11] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Advances in cryptology – CRYPTO '86*, in Lecture notes in computer science, vol. 263. Springer, 1986, pp. 186–194. doi: 10.1007/3-540-47721-7_12.

[12] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein, *The BLAKE2 cryptographic hash and message authentication code (MAC)*. IETF, 2015. Available: https://www.rfc-editor.org/rfc/rfc7693

[13] J. Thaler, "Proofs, arguments, and zero-knowledge," *Foundations and Trends in Privacy and Security*, vol. 4, no. 2–4, pp. 117–660, 2022, doi: 10.1561/3300000030.

[14] C.-P. Schnorr, "Efficient signature generation by smart cards," *Journal of Cryptology*, vol. 4, pp. 161–174, 1991, doi: 10.1007/BF00196725.

[15] A. J. Menezes, *Elliptic curve public key cryptosystems*, vol. 234. in The springer international series in engineering and computer science, vol. 234. Boston, MA: Kluwer Academic Publishers, 1993. doi: 10.1007/978-1-4615-3198-2.

[16] H. Krawczyk, "Cryptographic extraction and key derivation: The HKDF scheme." Cryptology ePrint Archive, Paper 2010/264, 2010. Available: https://eprint.iacr.org/2010/264

[17] A. Konrad and T. Vellekoop, "CIP-68: Datum metadata standard." https://cips.cardano.org/cip/CIP-68, 2022.