Contents

my_coin WhitePaper (Draft)

This document is the **technical description / draft specification** of the my_coin protocol.
It is intended for readers who already have a rough idea of what zero-knowledge proofs and Zcash-style anonymity systems can do (but are **not** required to understand low-level ZK algorithms or curve details).

Roadmap / Document Structure

This document is divided into 6 parts, corresponding to the following reading path:

1. **Background and Status Quo**
   - What is the structural tension today between "privacy" and "anti–money laundering" in payment systems?
2. **The Problem I Want to Solve**
   - The properties / goals my_coin aims to achieve, and what it explicitly does *not* try to solve.
3. **Overview of the Protocol Mechanism: How my_coin Works**
   - From the perspective of the state machine and transaction types, a high-level description of my_coin's core mechanisms (account coloring, anonymous pool / non-anonymous layer, anonymous payments, accountability, etc.).
4. **Why This Design: Attack–Defense and Incentives**
   - Explaining one by one those rules that may look "strange" at first sight (KYC + user agreement, blind_change, S1 & S2, masterSeed derivation, dynamic custody fee, etc.), and what attack paths would appear if my_coin *didn't* do it this way.
   - This part explains how the "strange" rules make my_coin anti-money-laundering friendly.
5. **Prototype Coverage: What Has Been Implemented / Not Implemented**
   - Mapping the protocol mechanisms in Part 3 to the current Python + Rust single-node prototype, and explaining what is already realized and what is still only in the design phase.
6. **How to Run the Prototype**

1. Background and Status Quo: What's Wrong with Today's Payment Systems?

This section first lays out "the main kinds of payment / settlement systems that already exist today", and then explains, for each of them, the extremes and gaps in terms of privacy and accountability.

1.1 Traditional Financial System: Strong AML, but Privacy Relies on Institutions "Keeping Secrets for You"

- A user's so-called "privacy" there essentially means:
  - Relying on whether banks, clearing houses, and third-party service providers are **willing and able to behave properly**;
- At the level of **cryptography and system design**, ordinary users have no "decentralized privacy guarantee" at all:
  - Once there is abuse, data leakage, or forced data sharing inside these institutions,
  - Users have almost no technical means to defend or audit; they can only rely on external regulation and ex-post accountability.

1.2 Public Blockchains: Transparent Ledger + Weak Anonymity

Take Bitcoin / Ethereum-style public blockchains as examples (ignoring privacy contracts for now):

> The payment path is completely transparent;
>
> At the address level, things *appear* to be "anonymous strings", but in real-world usage:

- Most people do **not** obtain BTC / ETH via anonymous mining. Instead, they usually:
  - Buy in via centralized exchanges or fiat on-ramps;
  - These on-ramps almost always enforce strict KYC.
- Once one of your addresses is linked to your real-world identity at some point (for example):
  - Exchange in/out records;
  - Addresses disclosed in legal letters / tax filings;
- From that point on:
  - The history and future behavior of that address can be linked to your identity;
  - All related addresses and fund flows can gradually be de-anonymized via on-chain graph analysis.

So:

> Unless you have extremely strict operational discipline and fully anonymous in-flow and usage patterns,
> once identity is exposed at a single point, it is very easy to track and analyze your later fund usage in the long term.

1.3 Privacy Coins / Mixers: Strong Anonymity, but They Greatly Reduce the Difficulty and Cost of Money Laundering, and Are Inconvenient or Risky for Normal Users Under Regulation

Systems like Zcash, Monero, Tornado Cash and other privacy tools attempt to provide very strong anonymity using cryptography:

- From the perspective of external observers:
  - They cannot see who is paying whom;
  - In some designs, even the transfer amount can be hidden;
  - All they know is that "the total amount in the system is conserved" and some minimal necessary information.

From a technical perspective, they indeed do quite well in the dimension of "privacy of individual transactions".

But in reality:

- For normal users:
  - They are **costly and inconvenient to use**:
    - Many centralized exchanges, banks, and payment institutions either refuse to support privacy coins / mixer-related assets at all;
    - Or once they discover that some funds have passed through a mixer or certain privacy pools, they directly refuse to receive them or require complicated source-of-funds documentation;
- For hackers / scammers / extortion groups and money-laundering intermediaries:
  - Once they successfully send stolen funds into privacy coins / mixing pools:
    - In many cases they can **completely disappear from public view**;
    - Afterwards they can slowly cash out the funds via various off-chain channels and OTC trades;
    - Tracking becomes much harder compared to transparent ledgers.

To summarize: - From the perspective of "cryptographic anonymity", privacy coins / mixers do very well; - But from the perspective of "system-level behavior", they look more like: - Extremely favorable tools for attackers (the friction for laundering and moving stolen funds is greatly reduced); - While for normal users they combine two major drawbacks: **inconvenient to use + high regulatory risk**.

1.4 Existing Compromise Solutions: Backdoored Privacy Protocols and "Only Accepting Clean Money"

In order to find trade-offs between "privacy" and "regulation / AML", the industry has already proposed some approaches, which roughly fall into two categories:

1.4.1 Privacy Designs with "Viewing Keys / Super Keys"

- The basic idea is:
  Let the system or some "regulator" hold an extra viewing key / super key:
  - Day-to-day transactions remain anonymous to the public;

  - But when an investigation is needed, this key can be used to open up the anonymous history of individual accounts or even the entire system.
- The problem is:
  - This key itself is a **highly centralized form of power**:
    * Who holds it? Is it single-holder or multi-sig? How do we audit that it is not abused?
    * Once it leaks or is abused by insiders, everyone's history is exposed at once;
  - From the user's point of view, this essentially falls back to the traditional model of "trusting some central party to properly use this key", rather than providing truly backdoor-free cryptographic privacy.

1.4.2 "Only Accept Clean Money" – Compliance-Oriented Mixers / Privacy Pools

1.4.2.1 "Compliant CoinJoin" as in Wasabi Wallet A relatively typical approach today is to wrap traditional CoinJoin with a layer of "only accept clean money" risk control. Wasabi Wallet is a representative example.

Very roughly:

- Under the hood it is still standard **Bitcoin CoinJoin**: multiple users aggregate their UTXOs into a single multi-input multi-output transaction, shuffling the mapping between inputs and outputs.

- When the coordinator receives each UTXO, it queries an on-chain analytics service for a **risk score**, checking whether the funds are associated with sanctioned addresses, known hacks, ransomware, and so on.

- UTXOs with too high a risk score are simply rejected from that CoinJoin round; publicly it is advertised as "our CoinJoin only serves clean bitcoin".

There is a key point here:

- **All participants of the same CoinJoin round are effectively treated as being "in the same pot"**.
  - From the on-chain point of view, a CoinJoin is just a bunch of inputs and outputs;

  - If later one input in the transaction is re-labeled as "actually hack-related", then all other outputs in that CoinJoin can, to varying degrees, be "tainted" as well.

A typical attack / collateral-damage path could look like this:

- An attacker first steals Alice's private key and gains control of her wallet;

- They take a UTXO of Alice's that is **still considered an "excellent customer" in the risk-control system** and use it to join a Wasabi CoinJoin;

- Because its prior history looks entirely "clean", neither the coordinator nor the analytics service will filter it out;

- After several rounds of CoinJoin, this UTXO is now mixed with a large number of innocent users' inputs;

- Some time later, Alice realizes the theft and reports it; only then do the risk-control providers flag that original outgoing transaction as "hack-related".

At this point, if we try to "crack down on this laundering path", the only way is to go along the graph and assign higher risk scores to the entire CoinJoin and possibly many downstream UTXOs:

- For the attacker, this is of course undesirable;

- But for **the other innocent users who joined the same CoinJoin just to improve privacy**, their coins are also labeled as "high risk / suspicious origin", leading to extra trouble at exchanges / banks, or even investigations and freezes. It is also hard for them to prove their innocence purely through cryptographic means.

1.4.2.2 "Compliant ZK Privacy Pools" Represented by Proof-of-Innocence / Privacy Pools

This line of work (a typical example being **Privacy Pools** proposed by Vitalik et al., and Railgun's **Private Proofs of Innocence** series) tries to strike a compromise between "strong privacy" and "regulatory acceptability":

- Users deposit funds into a ZK privacy pool;
- The protocol or third parties maintain a collection of "good deposits" and a list of "known bad funds";
- When users withdraw or interact with counterparties, they generate a ZK proof to show:
  - The withdrawn funds can be explained as coming from some good set;

  - And do **not** come from certain blacklisted deposits;

  - Without revealing exactly which one.

To see the structural issues of this route, we first fix an example timeline and define "conservative" and "aggressive" strategies on top of it.

Example Timeline: The Window T = 0 → T = 5

Assume the following timeline:

- T = 0: Ransom / scam proceeds are deposited into the privacy pool, forming a deposit commitment $\texttt{commit}_1$;

- $T \in (0, 5)$: $\texttt{commit}_1$ is split, merged, and passed around inside the pool multiple times; some of the funds may already have been withdrawn to various public addresses;

- T = 5: Law enforcement and analytics providers finally confirm that $\texttt{commit}_1$ corresponds to stolen funds and add it to a blacklist.

Then the interval $T \in [0, 5]$ is the "dangerous window" for this money. On this timeline, there are two extreme strategies:

- **Conservative strategy**:
  - Only at $T \geq 5$ do we add $\texttt{commit}_1$ itself to the blacklist;
  - Future attempts to deposit from the same source into the pool are rejected;
  - But we do **not** automatically blacklist any newly created commitments or withdrawals that occurred during $T \in (0, 5)$.
- **Aggressive strategy**:
  - Once $\texttt{commit}_1$ is flagged as stolen at T = 5, we also blacklist all pool states, newly created commitments, and withdrawals that could be related to it during that window;
  - We can also propagate further along the graph: public accounts that received such withdrawn funds during the window, and a long chain of accounts connected to them, can also be automatically included in high-risk sets.

Next, let's look at how these two strategies behave in practice.

---

(1) PoI / Privacy Pools in Reality: Conservative "Entrance-Only" Strategy = Maginot Line

Existing PoI / Privacy Pools essentially implement the **conservative strategy** described above:

- Railgun's PoI checks whether the tokens being shielded on "entry" come from known bad sources:
  - If they are on the blacklist → reject this shield operation;
  - If not → allow them into the privacy pool;
- After entering the pool, PoI itself cannot see the splits, merges, or withdrawals, and does not retroactively intervene;
- Similarly, Privacy Pools-style proposals provide "set + ZK proof" mechanisms at the contract level, but which deposits are deemed bad and how to structure the sets are mainly determined by users and higher-level applications. The contract does not hard-code "window-period catch-all" logic.

Going back to the timeline:

- As long as $\texttt{commit}_1$ has **not** yet been blacklisted in the period T < 5, all derivatives and withdrawals flowing from it inside the pool are treated as normal;
- When $\texttt{commit}_1$ is added to the blacklist at T = 5, it may have already been fully spent within the pool and no longer exist as a directly actionable deposit;

- The result: the blacklist only blocks **future attempts to reenter the pool from the same source**, and is basically powerless against the laundering that has already been completed during the earlier window.

From the attacker's perspective, this conservative strategy is very friendly:

> At T = 0, they receive stolen funds and deposit them as $\texttt{commit}_1$;
> By T = 3, they have already finished splitting and withdrawing inside the privacy pool; the stolen funds have left their original shape;
> At T = 5, when $\texttt{commit}_1$ is finally blacklisted, the useful part of the value has long since gone.

At this point, what remains in the blacklist is essentially an **already-spent deposit record**, which no longer imposes real constraints on the attacker. This is very similar to the **Maginot Line** effect:

The defensive line looks impressive, but in actual combat, the critical path has already bypassed it. By the time you react and close the gate, there is no one left behind it.

---

(2) If We Switch to an Aggressive Strategy: AML Is Strong, but the System Collapses

Now suppose the protocol does **not** adopt the conservative approach but instead uses the **aggressive strategy** defined above:

> Once $\texttt{commit}_1$ is flagged at T = 5,
> The strategy blacklist (or at least graylist) all commitments created during $T \in [0, 5]$ and all withdrawn funds in that period.

In the real world, scam / ransomware / card-fraud money flows into financial systems **every day**, not just occasionally:

- Each day, new "$\texttt{commit}_i$ deposited at T = 0, and only confirmed as stolen at T = some hours/days later" timelines appear;
- For each $\texttt{commit}_i$, the "dangerous window" is from the moment it is deposited to the moment it is blacklisted;
- These windows accumulate on the time axis, and the combined effect is: **all year round, there are multiple open windows at any given time**. It is rare—and may almost never happen—that a commitment is guaranteed not to fall into *some* window.

Under the aggressive strategy, the system faces two extreme options:

(2a) No "Whitening" Exit at All: Almost All Commitments Get Stuck Forever

Given a reality of "stolen funds flowing in continuously + overlapping windows over time", this leads to:

- Almost every commitment that interacts with the privacy pool will, at some point, get pulled into a black/gray list because of some stolen $\texttt{commit}_i$'s window;

- After a while, almost all commitments in the pool become "suspect", and commitments that have *never* fallen into any window instead become rare exceptions;

- If black/graylisted commitments are not given any cryptographic way to clear their name, they basically get stuck forever—and this is likely to be the **majority** of the pool. Under the aggressive strategy, providing a cryptographic "whitening" mechanism for commitments is almost mandatory.

(2b) Allow "Whitening" from Black/Gray Lists: It Becomes Almost Impossible to Prove Innocence in Practice

To avoid destroying the entire system, the protocol is then forced to introduce some kind of "whitening exit":

> "Yes, this commitment falls into some dangerous window,
> but if you can produce a ZK proof showing that
> the nullifier you're using did **not** spend any deposit from the blacklist (like $\texttt{commit}_1$),
> then you can be removed from the black/gray list."

In a typical commitment model, the spent old commitment looks like:

```
old_commit = H(amount_old, other_fields, secret_randomness
r_old)
```

At the step when a new commitment is created, a nullifier is produced to "spend" this `old_commit`, and the new commitment is recorded. To prove that "this nullifier did **not** spend any blacklisted commitment (for example $\texttt{commit}_1$)," you essentially need to prove that:

- You know the private randomness `r_old` (and other private inputs) of the consumed `old_commit`;

- The `old_commit` derived from these private inputs is not in the bad set.

The key issue is: **once the step involves a transfer of ownership, this private information typically resides with the payer, not the payee.**

More concretely:

- When initiating the transfer, the payer holds the full set of private inputs of the `old_commit` being nullified, including the secret randomness `r_old`;

- The payee only learns the new `new_commit` and the private inputs related to themselves, but has no knowledge of the internal data of the old commitment.

If we want the **payee** to complete the whitening, there are two conceptual options, both highly problematic:

1. **The payer actively hands over the old commitment's secret randomness to the payee**
    - Then the payee can construct a ZK proof themselves to show that this nullifier did *not* spend any blacklisted commitment;
    - But the cost is: the payer has to reveal the full deposit-time privacy of their own funds to the payee—losing their own privacy.
    - This is hard to accept even for honest users; for real hackers/scammers it is even more unrealistic—they usually disappear once they get the money.
2. **The payee traces every previous hop and asks all upstream parties to jointly participate in a giant ZK proof**
    - In theory, one can design a multi-party proof protocol where every hop in the chain contributes some private input, so that the entire ancestry tree proves "originated from a clean commitment";
    - But in the real world there is no such social process:
        - Many payers use disposable addresses and one-off transactions, disappearing after they get paid;
        - This chain of payments may cross multiple countries and jurisdictions, making coordination nearly impossible;
        - Even if you can locate a payer, they have their own problem: the payer's commitment may be one of those that **was not blacklisted under the conservative strategy but gets caught under the aggressive strategy**, so they in turn must chase *their* upstream, and so on;
        - This means the payer now has to go further up one hop at a time, then that hop goes further up again, and so on—you don't know where this ends.

The result is:

Although the protocol textually defines a "whitening" exit, in typical ownership-transfer scenarios it is practically too costly to use.

---

2. Problems I Aim to Solve: Target Profile of my_coin

2.1 Protocol Goals (What I want to achieve)

- **G1: Decentralized, backdoor-free cryptographic anonymity**

  - Anonymity should rely entirely on public algorithms and the cryptographic structure itself, and not on any centralized "viewing key", regulatory backdoor, or super decryption key.

  - There must be no single entity in the protocol that can unilaterally decrypt and reconstruct the entire network's transaction history.
- **G2: Strong anonymity set — mixed with a large pool of funds, not a small circle**
  - Anonymous payments should be mixed together with a large, high-volume pool of funds and many transactions, rather than only with a small number of inputs or a small ring of signature members.
- **G3: Provide anonymous payments without lowering the barrier to money laundering**
  - my_coin is intended to provide "anonymous payments usable by rule-abiding users", not to make money laundering easier than it is today.

  - In the ideal case, the difficulty for an attacker to launder money via my_coin **should not be significantly lower than** the difficulty of laundering money using a combination of traditional banking plus non-mixing Bitcoin.
- **G4: Payments must be sufficiently convenient and fast**
  - When using anonymous payments, the complexity and confirmation time should be as close as possible to PayPal / credit-card payments.
- **G5: Provide a technical basis for ex-post tracing and "recovery of stolen funds", assuming a backup of the private key still exists**
  - The goal is that, when funds are stolen or misused, there is a strong enough on-chain evidential basis for victims and judicial authorities so that "recovery / accountability" has real technical support in actual legal and procedural flows.

2.2 What is explicitly *not* guaranteed

- **NG1: No guarantee of anonymity after private key leakage**
  - my_coin provides decentralized, backdoor-free cryptographic anonymity **under the assumption that the private key is properly kept and not leaked**.

  - Once a user's private key is leaked, stolen, or forcibly surrendered, the protocol no longer makes any promises about the anonymity of past transactions associated with that private key.

---

3. Protocol Mechanism

This chapter mainly explains what the protocol *actually specifies* at a high level, without going too much into *why* it is designed this way.

For the reasoning behind the design, please see Chapter 4.

my_coin uses ECC over the Pallas–Vesta curve. For Merkle trees and other places where hashes are computed, it uses the Poseidon hash function, in order to be friendly to zero-knowledge proofs (this project uses Halo2 ZK). In the future, I may consider upgrading to ZK-STARKs to achieve quantum resistance.

Due to time constraints, I have not yet written a fully complete and rigorous protocol specification. This chapter mainly introduces the core mechanism framework and may lack some specific details.

In addition, this chapter only describes the protocol as I envision it.
The prototype implementation has a few minor differences from the protocol in certain details and is slightly simplified.
For a detailed introduction of the prototype, please refer to Chapter 5.

3.1 Account Model in my_coin: What Extra Identity States Exist

my_coin also uses an account-based model, but each account has four extra fields hard-coded into its on-chain state, like additional "system fields":

- `color`: integer, default 0;
- `ID`: real-world identity identifier;
- `master_seed_hash`: derived from a password remembered by the user via a Poseidon hash chain;
- `token`: 32-byte integer, used together with `master_seed` to derive the private key of this account.

Initialization rules:

- For a new account (an account whose on-chain state has not been modified before), the defaults are:
  - `color = 0`;
  - `ID` is considered unset;
  - `master_seed_hash` is considered unset;
  - `token` defaults to a value of "32 bytes of all zeros".
- As long as `color ≠ 0`, an `ID` must be set; this is a hard constraint at the chain level;
- When an account is upgraded to `color = 1`, the node must permanently record the binding between this address and `(ID, master_seed_hash, token)` in its state, for later compliance, audit, and accountability.

3.2 How a Normal User Upgrades `color = 0` to `color = 1`

This part is mostly implemented in the prototype, with some minor simplifications. Here it describes the **first** blue-account upgrade for a given real-world identity (the first

`color = 1` account with `token = 0`). Subsequent blue addresses for the same identity are described in Section 3.3.

Upgrading an account from `color = 0` to `color = 1` consists of four steps:

1. Locally derive `master_seed_hash`, the first scalar `SK` and public key `PK` from a password and generate a ZK proof;
2. The user signs the my_coin terms of use;
3. The Clerk performs offline KYC and confirms the identity;
4. A blue-address upgrade request, endorsed by the Clerk's signature, is submitted on-chain; a cooling-off period elapses, and then the account officially becomes `color = 1`.

3.2.1 Local Cryptographic Steps: password → master_seed → master_seed_hash, SK, PK + ZK Proof On the user's own device, for the **first blue account**, the local cryptographic steps are:

1. Choose a password that exists only in their own memory;
2. Apply Poseidon once to this password to obtain `master_seed`;
3. Apply Poseidon once more to `master_seed` to obtain `master_seed_hash`;
4. Fix `token = 0` for this first blue address;
5. Concatenate `master_seed` with this `token` in the way specified by the protocol, and apply Poseidon hash once to the concatenated input to obtain a private scalar `SK`;
6. Let `G` be the fixed generator point on the elliptic curve used by my_coin; compute the corresponding public key point

   $$PK = SK \cdot G.$$

7. Construct a zero-knowledge proof for the following statement (informally):

   There exists a password that I know,
   such that applying Poseidon once to this password yields some `master_seed`,
   applying Poseidon once to this `master_seed` yields the currently public `master_seed_hash`,
   and, with `token = 0`, concatenating this `master_seed` with `token` in the prescribed way
   and hashing once more with Poseidon yields a private scalar `SK` such that
   $SK \cdot G$ equals the currently public `PK`.

In this first blue-address upgrade request, the public input values to the ZK proof are:

- `master_seed_hash`;
- `token = 0`;
- the elliptic-curve public key `PK`;
- and the ZK proof itself.

The password and `master_seed` are never revealed; `SK` and all other intermediates remain internal to the circuit.

Implementation navigation (ZK details):

- Rust-side ZK proof generation and verification:
  - `zkcrypto/src/zkproof_for_ii_blue_apply_gen.rs`
  - `zkcrypto/src/zkproof_for_ii_blue_apply_verifier.rs`
- Protocol documentation (logic and circuit constraint explanations):
  - See the documents under the `docs/blue_apply/` directory
- Tests calling the Rust ZK APIs (to check that generation and verification behave as expected):
  - `tests/rust_api_tests/zkproof_ii_blue_apply_tests.py`

3.2.2 Terms of Use: What the User Must Agree to Before Upgrading Before the Clerk helps the user perform the on-chain upgrade, the user must sign the my_coin terms of use. The terms should clearly specify in one place:

3.2.2.1 Commitments About the Password and ZK Proof The user declares and agrees that:

- They do in fact know the password behind the `master_seed_hash` they submitted;
- Any zero-knowledge proof they submit as part of a blue-address application is generated based on this password that they themselves control,
  not a proof text given by someone else which they simply forwarded without understanding;
- They understand and accept that if they later forget this password, it will not only make "asset recovery" difficult,
  but will also directly affect their ability to present evidence and allocate responsibility in cases involving stolen or suspicious funds.

3.2.2.2 When an Incoming Payment Is Deemed "Stolen / Suspicious Funds" The terms specify that a payment received by a `color = 1` account will be treated as stolen or suspicious funds by my_coin when both of the following hold:

1. The account holder cannot provide a reasonable, verifiable explanation of the source of funds (for example, cannot provide evidence of a corresponding contract, order, labor, gift, etc.);
2. The specific transfer is reported by the originating party of the funds (the exact meaning of "report" is defined below).

In the protocol semantics and the terms of use, such an incoming payment is considered stolen / suspicious funds and triggers the subsequent liability mechanism.

3.2.2.3 The Precise Definition of "Report" in This Context "Report" here means an action that satisfies all of the following:

- The reporter is the originating party of the funds, i.e., the controller of the account from which the funds were originally sent;
- The reporter can provide a concrete on-chain fingerprint of this transfer: which transaction it is, the source account, destination account, and the amount;

- The reporter must use the private key of the source account, or an equivalent zero-knowledge proof, to prove:

  "I really control this source account, and these funds were indeed sent out from an account under my control."

More concretely, there are two cases:

- If the transfer was a non-anonymous transparent payment:
  - The source and destination addresses are directly visible on-chain;
  - The reporter signs with the source address's SK and includes the transaction information.
- If the anonymous payment feature was used at the time:
  - The transfer already hides the identity of the source address;
  - When reporting, the originating party must additionally use their keys together with a zero-knowledge proof to show that
    "a certain anonymous output is in fact derived from a key system they control";
  - The concrete structure of anonymous payments and the exact proving method will be described separately later.
    my_coin only requires that a valid report, via cryptography, link "this anonymous payment" to "the reporter's account system".

Only reports satisfying the above conditions count as "reports" under the terms of use, and can be used to classify funds as stolen / suspicious.

3.2.2.4 Obligations to Return Stolen / Suspicious Funds Once the conditions in 3.2.2.2 hold (i.e., an incoming payment is classified as stolen / suspicious funds), the terms specify:

- Within the "maximum liability amount" that the user has declared in advance, the account holder has a civil obligation to return such stolen / suspicious funds;
- The user agrees and pre-authorizes that, if necessary, local judicial authorities may enforce this obligation against their off-chain assets
  (bank accounts, real estate, etc.) through compulsory execution.

3.2.2.5 Intermediary Launderers vs Upstream Perpetrators: Differences in Liability Once a payment is classified as stolen / suspicious funds:

- If the account holder claims to be merely an intermediary in the laundering chain, the terms allow them to reduce their liability by:
  - Using their SK to clearly identify the downstream receiving account(s);
  - Providing on-chain evidence that the stolen funds have indeed been forwarded from accounts under their control to those downstream accounts,
    and that they only retained a small portion as a fee / commission;
  - In that case, they are liable for returning the portion they retained as commission, and pay penalties according to the rules;
    primary liability for the bulk of the stolen funds is then pursued downstream along the chain.
- If the account holder:

  - Cannot identify downstream recipients or refuses to provide any verifiable evidence; or
  - Is in fact the top-level perpetrator of theft / fraud / extortion;

Then, under the terms of use, they bear primary liability for the entire amount of stolen funds, up to their own agreed liability cap.

3.2.2.6 Forgetting the Password and the Liability Cap Since many ways to "reduce liability" depend on the account holder controlling their key system (and that key system is driven by password / master_seed), once the user forgets their password, the following may happen:

- They might never have actually received the funds (someone else used their identity to launder money);
- But from the chain's accounting perspective, those funds are still recorded under their address;
- When a report is filed, they cannot use keys and evidence to prove that the funds have already gone downstream.

To prevent this risk from growing without bound, the terms require:

- When a user first upgrades an account to `color = 1`, they must specify a "maximum liability amount" they can personally bear;
- The protocol logic uses this cap to limit, within a given time window, the cumulative amount of "anonymous payments from others" that this account may receive;
- In the worst case (the user forgets the password and someone else uses their account to launder), their obligation to return funds will not exceed the cap they agreed to in the terms.

3.2.3 Clerk, KYC, Cooling-Off Period, and the First Blue Account (`token = 0`) Let's move on to the Clerk and on-chain part.

Rules for the first upgrade to `color = 1` (the first blue account, `token = 0`):

- For the first blue account of a given real-world identity, the derivation rule described in Section 3.2.1 is applied with `token = 0`,
  and the resulting public key `PK` is used to define the account address as `address = Poseidon(PK)` on chain;
- When this first account is successfully upgraded from `color = 0` to `color = 1` (after the cooling-off period), its on-chain state for that `address` is updated as follows:
  - `color`: set from 0 to 1;
  - `ID`: the real-world identity identifier in the offline identity system;
  - `master_seed_hash`: the value published by the user and validated via ZK proof;
  - `token`: 0 (32 bytes of all zeros).
- This first `color = 1` account's `token` must be "32 bytes of all zeros".

Process summary:

1. The user generates `master_seed_hash`, chooses `token = 0`, derives the corresponding public key `PK` and the ZK proof described in Section 3.2.1, and confirms that they have signed the terms of use and completed offline KYC with the Clerk;

2. The Clerk, after finishing offline KYC and checking that the real-world identity matches, uses their official key to sign this first blue-address upgrade request (which contains `master_seed_hash`, `token = 0`, `PK`, and the ZK proof);
3. This Clerk-endorsed upgrade request is submitted to the blockchain node as an on-chain transaction (it may be broadcast by the user, by the Clerk, or by any relay node; what matters is that it carries a valid Clerk signature);
4. The chain verifies the ZK proof against these public values and verifies the Clerk's signature;
5. The upgrade request enters a cooling-off period:
    - During the cooling-off period, the Clerk monitors all upgrade requests signed with their key;
    - If any request is found to be unauthorized (for example, if the Clerk's signing key has been stolen), it can be revoked during this period;
6. If the cooling-off period ends without revocation, the upgrade request is accepted:
    - the node computes `address = Poseidon(PK)` using the derivation rule,
    - the account's state for this `address` is updated as above,
    - and the binding `(address, ID, master_seed_hash, token = 0)` becomes an on-chain fact.

3.3 Subsequent blue addresses: token choice, PASTA-curve derivation, and the new ZK statement

When a single real-world identity wants to open multiple `color = 1` addresses, they must use the "subsequent blue addresses" procedure. Here it is not just "recommending" that you derive addresses in a certain way: the protocol **requires** that every subsequent blue-address upgrade request must include a ZK proof showing that the SK / PK was indeed derived from the same `master_seed` according to the specified rule.

3.3.1 Token choice and derivation rule (Poseidon + PASTA) For a given user with an already established `master_seed` (obtained from a password via Poseidon):

- The user may choose a 32-byte `token` value for each subsequent blue address (the token must be nonzero, and each address must use a different token);
- The derivation rule for the private key SK is:
    - Concatenate `master_seed` with this `token` in the way specified by the protocol;
    - Apply Poseidon hash once to this concatenated input to obtain the private key SK;
- On the elliptic-curve side, my_coin uses the PASTA curve, rather than traditional secp-series curves:
    - Use SK on the PASTA curve to compute the public key PK;
    - The address is defined as:
        * Apply Poseidon hash once more to the public key PK to obtain `address`;
        * This `address` is the account key that is publicly visible on chain once the upgrade to `color = 1` is finalized.

Thus, starting from the same `master_seed`, a given `token` uniquely determines a chain:

> master_seed → (concatenate with token) → Poseidon → SK → compute PK on the PASTA curve → Poseidon(PK) → address.

3.3.2 What the ZK proof must show when applying for subsequent blue addresses For each subsequent blue address, when the user applies to "open a new `color = 1` address with this particular `token`", the request must be accompanied by a new ZK proof. This proof is no longer just "I know a password"; instead it is closer to "I know a `master_seed`, and this new PASTA public key `PK` really is derived from that same `master_seed` and the chosen `token`".

It can be decomposed like this:

- Public inputs include:
  - The PASTA public key `PK` of the new blue address (the circuit does **not** need the `address` itself; the node computes `address = Poseidon(PK)` outside the circuit);
  - The corresponding `token` (32-byte integer, nonzero for subsequent blue addresses);
  - `master_seed_hash` (the same value as for this user's first blue address).
- Private inputs (witness) include:
  - `master_seed`;
  - Secret intermediates appearing in the derivation process (for example SK, which is sensitive and can remain internal to the circuit);
  - Any internal variables needed to compute PK from SK on the PASTA curve (handled internally by the circuit and not revealed).

The ZK proof must establish the following statement (informally):

> "There exists a `master_seed` known to me,
> such that applying Poseidon once to this `master_seed` yields the currently public `master_seed_hash`;
> under this same `master_seed`, if I concatenate it with the currently public `token` in the prescribed way
> and apply Poseidon hash to the concatenated result, I obtain some private key SK;
> using this SK on the PASTA curve, I can compute a public key PK that equals the currently public `PK`.
> In other words, this `PK` is indeed derived from this `master_seed` and this `token` under the protocol's derivation rule."

After the node verifies this ZK proof, it is mathematically convinced that:

- The new public key `PK` and the public `token` indeed belong to the same `master_seed`;
- This `master_seed` is, via `master_seed_hash`, bound to the user's cryptographic identity established earlier (the "password → master_seed → master_seed_hash" relation was already proven during the first blue-address application);

- Therefore, the address defined as `address = Poseidon(PK)` can safely be treated as "another blue address of the same `master_seed` / the same subject".

At the on-chain state level:

- When the upgrade transaction for this new blue address is processed, the transaction includes:
  - the public values `(master_seed_hash, token, PK)`,
  - the proposed `ID` (matching the user's previous blue address),
  - and the ZK proof described above;
- The contract verifies that:
  - The attached ZK proof satisfies the statement above for these public values;
  - The `master_seed_hash` used in the proof matches the one written in the transaction;
- Once this upgrade transaction is finalized on chain:
  - the node computes `address = Poseidon(PK)`,
  - the on-chain system fields for this `address` are set to:
    - `color`: 1 (upgraded from 0);
    - `ID`: identical to the user's previous blue address;
    - `master_seed_hash`: identical to the previous one;
    - `token`: the currently public `token`;
  - and this new `(address, ID, master_seed_hash, token)` binding is recorded on chain.

Implementation navigation (ZK details):

- The derivation constraints in this subsection reuse the same ZK circuit family as for blue-account applications:
  - `zkcrypto/src/zkproof_for_ii_blue_apply_gen.rs`
  - `zkcrypto/src/zkproof_for_ii_blue_apply_verifier.rs`
- Corresponding protocol and circuit descriptions:
  - Detailed specifications under `docs/blue_apply/`
- Python-side integration tests for these circuits:
  - `tests/rust_api_tests/zkproof_ii_blue_apply_tests.py`

3.4 Account nonce and transparent transfer rules

- Beyond the color and identity mechanism, each my_coin account carries an on-chain `nonce` field, which constrains the ordering of all operations that this account actively initiates. Transparent transfers operate on top of this, with overall rules very similar to Ethereum.

3.4.1 Nonce rules: essentially "++nonce" A my_coin account has an on-chain `nonce` field that constrains the order of all operations actively initiated by that account:

- In on-chain state, `nonce` starts at 0.

- Every time this account **actively initiates an operation that changes its own state** (such as changing color, making a transparent transfer, or depositing funds from the non-anonymous layer into the anonymous layer), the outer transaction envelope must carry a `nonce'`.
- The validation rule is:

To accept the transaction:
`nonce'` must equal the account's current on-chain `nonce + 1`.

After the transaction executes successfully:
the account's on-chain `nonce` is updated to `nonce'`.

In other words:

- For the first active operation: the state has `nonce = 0`, the envelope must carry `1`, and after success the state becomes `1`;
- Second operation: the state has `nonce = 1`, the envelope must carry `2`, and after success the state becomes `2`;
- And so on: each time the node checks `state_nonce + 1`, then directly jumps `state_nonce` to this new value.

This is what is meant by "essentially `++nonce`, not `nonce++`":

- The on-chain value represents "the most recent value that has already been successfully used";
- The transaction carries "the next value to be used".

Passive changes (such as someone else sending funds to you) do not modify your `nonce`, because those are not operations initiated by you.

3.4.2 Transparent transfers: basically the same as Ethereum The transparent-transfer layer behaves almost the same as Ethereum:

- The account signs with its own private key to prove "I control this address";
- It specifies the sender, the receiver, the amount, and the `nonce` that should be used according to the rules in the previous subsection;
- The node verifies the signature and the `nonce`, checks that the sender's balance is sufficient, and then performs a simple "subtract / add" on balances, ensuring that:
  - The sum of all input amounts equals the sum of all output amounts;
  - No account balance ever becomes negative.

There is nothing particularly novel here; this subsection is mainly to clarify that:
on the "non-anonymous, transparent layer", my_coin's accounting and ordering behavior matches common account-based chains, and this layer serves as a foundation for the anonymous-layer design that follows.

3.5 Anonymous commitment layer: AC and MerkleTreeCommit

The anonymous layer of my_coin is built around **anonymous commitments (ACs)**. An AC is a commitment hash, and its plaintext is called a `note`.

When a non-anonymous account deposits funds "into the anonymous layer", this operation is called an **account-to-anonymous deposit**.

3.5.1 MerkleTreeCommit and the structure of anonymous commitments The core object in the anonymous layer is the **anonymous commitment (AC)**, which is simply a commitment hash.

The preimage of an AC is a `note`, whose fields are fixed and ordered as four items:

1. `balance`: how much value is stored in this anonymous unit;
2. `sk`: the secret that controls this anonymous unit;
3. `nonce`: this anonymous unit's own counter;
4. `src`: a source label (for example, indicating whether it comes from an account, or from a previous note in a chain).

The AC is computed using the Poseidon hash, applied to these four fields in fixed order:

AC = Poseidon(balance, sk, nonce, src)

All ACs are inserted into a single commitment tree called **`MerkleTreeCommit`**:

- The structure of `MerkleTreeCommit` is very similar to Zcash Sapling's note commitment tree, and can be understood as "its Poseidon-based version";
- Its height is fixed at 32 levels, so it supports up to 2^{32} leaves;
- Each leaf is an AC (i.e., the result of `Poseidon(balance, sk, nonce, src)` as defined above).

The following are public on chain:

- The root of `MerkleTreeCommit`;
- The value of each AC;
- The insertion position of each AC within the tree.

The four fields of the note and `sk` itself are not public.

Implementation navigation (ZK details that will be used later in the anonymous-pay section):

- ZK circuits related to the anonymous layer (not account-to-anonymous deposits in this subsection, but anonymous pay and internal transfers):
  - `zkcrypto/src/zkproof_for_anon_pay_gen.rs`
  - `zkcrypto/src/zkproof_for_anon_pay_verify.rs`
- Protocol documentation (logical description of the anonymous-pay circuit):
  - `docs/anon_pay/`
- Rust API tests (ZK for anonymous pay):
  - `tests/rust_api_tests/zkproof_anon_pay_tests.py`

3.5.2 A basic principle: no ownership transfer when generating an AC In my_coin's design, there is a very important—but somewhat "counter-intuitive"—principle:

> Whenever a new anonymous commitment (AC) is being generated,
> the protocol does **not** allow ownership transfer to be completed in the same action.

In other words:

- The party who grants value to this AC; and
- The party who controls this AC;

must be the same entity.

Semantically, generating an AC is essentially:

> "Moving funds from your own public account into an anonymous unit controlled by yourself,"

not:

> "Sending funds anonymously to someone else."

This principle is hard-coded into the ZK constraints for the **account-to-anonymous deposit circuit**. The reasons for this will be explained in detail in Chapter 4. For now, this subsection just states the principle and describes the structure of the deposit under this rule.

3.5.3 Account-to-anonymous deposit: ZK statement for "non-anonymous account → single anonymous commitment" This subsection only considers the simplest case:

> In one operation, a non-anonymous account deposits into the anonymous layer,
> and only one AC (a single anonymous unit) is created.

Extensions such as multiple inputs/outputs and change will be introduced in the anonymous-pay section later. Here, the goal is to clarify the semantics of the basic "account → single AC" proof.

3.5.3.1 Typical deposit scenario

- There is a non-anonymous account, whose on-chain address is `ADDR`;
- The account holder wants to move some public balance `balance` from this address into the anonymous layer;
- The protocol will create a new AC and insert it into `MerkleTreeCommit`;
- The design requires that:
  - The funds are indeed deducted from `ADDR`'s balance;
  - When this AC is created, the secret `sk` controlling this AC is held by the same entity who controls `ADDR`;
  - That is: generating an AC does not perform ownership transfer.

3.5.3.2 Public inputs and secret inputs in the deposit circuit In the corresponding ZK proof, the key inputs can be divided as follows:

Public inputs:

- `balance`: the amount being deposited into the anonymous layer;
- `nonce`: the new note's own counter (note: this is not the account's `nonce`, but the note's field);
- `src`: the source label of the new note (encoded to reflect "which account it comes from");
- `ADDR`: the non-anonymous account address that initiates the deposit (the public paying address);
- `AC`: the value of the anonymous commitment inserted into `MerkleTreeCommit`.

Secret inputs (witness):

- `sk`: the secret that controls this anonymous unit, and also the secret used to generate `ADDR`'s private key;
- Intermediate values used to compute the PASTA-curve public key `PK` from `sk` (used inside the circuit and not made public).

3.5.3.3 Two statements the ZK circuit must prove The deposit circuit actually enforces **two** logical statements, and it enforces that **both** use the same `sk` in the proof.

First statement: `sk` → `PK` → `ADDR` (proving who is paying)

The circuit must prove:

> "There exists a secret `sk`,
> such that computing a public key `PK` from `sk` on the PASTA curve,
> and then applying Poseidon hash to `PK`,
> yields the currently public paying address `ADDR`."

The meaning of this is:

- The circuit confirms that:
  - `ADDR` is indeed derived from this `sk`;
- In other words, the entity who holds this `sk` controls `ADDR`, and is the source of the funds.

Second statement: construct the AC using the same `sk`

The circuit must further prove that, using the *same* `sk` together with public `(balance, nonce, src)`, the Poseidon hash computed matches the public `AC`:

> "Using the same `sk` as above,
> concatenate `(balance, sk, nonce, src)` in order,
> apply Poseidon hash to this four-tuple,
> and the result equals the currently public `AC`."

This means that:

- The anonymous commitment corresponds to a note with:
  - `note.balance` = public `balance`;
  - `note.sk` = private `sk`;
  - `note.nonce` = public `nonce`;
  - `note.src` = public `src`;

- The value of `AC` is indeed Poseidon of this note, not some arbitrary number.

Combining the two statements, the core semantics are:

> "There exists an `sk` that both controls `ADDR`
> and serves as the control key inside the note corresponding to the new anonymous commitment.
> And the note's `balance / nonce / src` match the current public values."

Therefore:

- Funds are debited from `ADDR`'s public balance;
- The `sk` controlling the new AC is the same as that of `ADDR`—the same controlling entity;
- In this "generate AC" step, it is impossible to complete "giving the money to another person" as an ownership transfer—because the circuit forces `sk` to be the same.

Why the protocol insists on this design (no ownership transfer when generating an AC), and how genuine hidden-identity transfers are implemented **inside** the anonymous layer, will be explained in later sections and in Chapter 4. This subsection focuses on explaining the structure and ZK statement of the account-to-anonymous deposit.

Implementation navigation (ZK details for the account-to-anonymous deposit circuit):

- Rust-side ZK proof generation and verification for deposits from an account into a single anonymous commitment:
  - `zkcrypto/src/zkproof_for_acct_to_anon_tx_gen.rs`
  - `zkcrypto/src/zkproof_for_acct_to_anon_tx_verifier.rs`
  - `zkcrypto/src/zkproof_for_acct_to_anon_tx_evil_gen.rs` (used for malicious paths / counterexamples)
- Protocol docs (circuits and semantics for "account → anonymous commitment"):
  - `docs/acct_to_anon_commit/`
- Single tests that call the Rust ZK API ("account → anonymous commitment"):
  - `tests/rust_api_tests/zkproof_acct_to_anon_tx_tests.py`

3.6 Anonymous Pay: single old AC → CommitChange & public address

This subsection describes the most typical form of anonymous payment:

- Consume one old AC;
- Create one new anonymous change AC (`CommitChange`);
- Pay a public amount `pay_balance` to a public address `ADDR_to`.

3.6.1 Quick recap of related objects: AC, MerkleTreeCommit, nullifier AC and MerkleTreeCommit were already defined in Section 3.5. Here, I only add points that are directly relevant to anonymous pay.

1. **Anonymous commitment (AC)**

- An AC is a Poseidon hash;
- Its preimage is a note, with the four fixed fields (defined in 3.5):
  - `balance`: how much value is in the anonymous unit;
  - `sk`: the secret controlling this anonymous unit;
  - `nonce`: the anonymous unit's own counter;
  - `src`: source label (account, previous AC in a lineage, etc.);
- The computation is:

  AC = Poseidon(balance, sk, nonce, src)

2. **MerkleTreeCommit**

- All ACs are inserted as leaves into a single Merkle tree called `MerkleTreeCommit`;
- Its structure is basically the same as Zcash Sapling's note commitment tree, but using Poseidon as the node hash;
- The tree height is fixed at 32, corresponding to up to 2^{32} leaves;
- Public on chain:
  - The root of `MerkleTreeCommit` (`CommitRoot`);
  - Newly inserted AC values and their positions in the tree.

3. **Definition of the nullifier**

The **nullifier** is used as a tombstone when a note is spent, preventing the same note from being spent twice.

- It is defined by taking the note and removing `balance`, leaving `(sk, nonce, src)` and hashing them with Poseidon in fixed order:

  nullifier = Poseidon(sk, nonce, src)

- Intuitively:
  - The nullifier is an "identity fingerprint" of the note that does not include the amount;
  - If the same nullifier appears on chain a second time, it signals an attempt to spend the same note twice, and the transaction can be rejected.

The old AC value `AC_old` is not a public input; it appears only inside the ZK circuit (recomputed from the note's internal fields).

3.6.2 Design principle: no ownership transfer when creating new ACs This subsection continues the hard rule from before:

> The action of creating a new anonymous commitment (AC)
> must not complete ownership transfer in the same step.

That is:

- The person who creates this AC; and
- The person who controls this AC;

must be the same entity.

In anonymous pay, this leads to the following:

- It is not allowed to "consume one old AC and at the same time produce two new ACs, one as your own change and one controlled by the payee";
- The true payee can only be a public address `ADDR_to` that receives a public amount `pay_balance`;
- The anonymous layer only does:
  - The old AC is spent;
  - The same controller receives a new `CommitChange` (anonymous change);
  - A portion of the old note's amount is peeled off and paid to `ADDR_to`.

3.6.3 Operation overview: consume old AC → create CommitChange → pay ADDR_to In the minimal Anonymous Pay scenario, one transaction does three things:

1. Consume one old AC (corresponding to an old note, whose commitment value is internally denoted `AC_old`);
2. Create one new anonymous change AC, denoted `CommitChange`;
3. Pay a public amount `pay_balance` to the public address `ADDR_to`.

From the on-chain observer's perspective, one can see:

- The current root `CommitRoot` of MerkleTreeCommit;
- A public `nullifier` (indicating that some old note has been spent);
- A new AC value `CommitChange` being inserted into MerkleTreeCommit;
- A transparent payment record: an anonymous source paying `pay_balance` to `ADDR_to`.

They **cannot** see:

- The internal fields of the spent old note: `balance_old / sk / nonce_old / src_old`;
- The change amount `balance_change` inside CommitChange;
- The exact leaf position of the old AC and its Merkle-path details.

3.6.4 Note structure and conventions for CommitChange The note corresponding to `CommitChange` still uses the four fields `(balance, sk, nonce, src)`, but with specific conventions for anonymous pay:

- `balance`: the change amount (secret, denote it as `balance_change`);
- `sk`: the same `sk` as in the spent old note (same controller);
- `nonce`: fixed to `0`;
- `src`: fixed to the public `nullifier` (i.e., the identity fingerprint of the old note).

So the note for CommitChange is:

- `balance = balance_change` (secret);
- `sk` (secret, identical to the old note's `sk`);
- `nonce = 0` (constant);
- `src = nullifier_public` (constant).

The value of `CommitChange` is then:

CommitChange = Poseidon(balance_change, sk, 0, nullifier)

This implies:

- CommitChange is explicitly "descended from" the old note identified by this `nullifier`;
- The controlling secret remains the same `sk`;
- On chain, it can be interpreted as "anonymous change for the same holder", rather than a new anonymous output for someone else.

3.6.5 ZK for Anonymous Pay: public inputs, secret inputs, and the statement Here I describe, in natural language, the structure of the ZK proof for a single Anonymous Pay transaction: the public inputs, secret inputs, and the statement to be proven.

3.6.5.1 Public inputs From the circuit's perspective, the public inputs for an Anonymous Pay include:

- `CommitRoot`: the current root of MerkleTreeCommit;
- `nullifier`: the nullifier of the old note being marked as spent;
- `CommitChange`: the value of the newly created anonymous change AC;
- `ADDR_to`: the non-anonymous address of the recipient (the payee is public);
- `pay_balance`: the public payment amount sent to `ADDR_to`.

Note that the change amount `balance_change` inside CommitChange is hidden and does not appear as a public input.

3.6.5.2 Secret inputs (witness) The prover (the anonymous holder) must supply the following secret data inside the circuit:

- The four fields of the old note:
  - `balance_old`;
  - `sk`;
  - `nonce_old`;
  - `src_old`;
- The balance of the new change note:
  - `balance_change`;
- The Merkle path for the old AC within MerkleTreeCommit:
  - Sibling hashes, left/right positions, etc., used to compute up to `CommitRoot` from `AC_old`.

The old AC value `AC_old = Poseidon(balance_old, sk, nonce_old, src_old)` is computed internally by the circuit and is not exposed as a separate public input.

3.6.5.3 Statements that the ZK proof must guarantee (in logical order) The circuit must prove the following logical statements:

Statement 1: the old note is indeed a real leaf in MerkleTreeCommit

Internally, the circuit:

- Uses the private fields `(balance_old, sk, nonce_old, src_old)` to compute:

  AC_old = Poseidon(balance_old, sk, nonce_old, src_old)

- Uses `AC_old` as the leaf, plus the private Merkle path, to compute the root from bottom to top according to the rules of MerkleTreeCommit;
- The resulting root must equal the public `CommitRoot`.

This ensures that:

- The spent item is truly one of the leaves of the tree;
- The note corresponding to `AC_old` is not external, fabricated data.

Statement 2: the public nullifier really comes from this old note

According to the definition:

nullifier = Poseidon(sk, nonce_old, src_old)

The circuit must prove:

- Using the same `sk, nonce_old, src_old` from Statement 1, computing `Poseidon(sk, nonce_old, src_old)` yields a result equal to the public `nullifier`.

In natural language:

> "The public `nullifier` is indeed computed from the `(sk, nonce_old, src_old)` of the same old note."

This lets the chain:

- Record only the `nullifier`; if the same value appears again in the future, it can detect a double-spend attempt.

Statement 3: CommitChange is correctly formed and controlled by the same sk

The circuit must prove:

- Take the same `sk` from Statement 1;
- Take a secret `balance_change`;

- Assemble the new note:
  - `balance = balance_change`;
  - `sk` = the same `sk`;
  - `nonce = 0`;
  - `src` = the public `nullifier`;
- Apply Poseidon hash to these four fields:

  Poseidon(balance_change, sk, 0, nullifier)

- The result must equal the public `CommitChange`.

This guarantees that:

- CommitChange is a structurally correct AC;
- Its `src` is explicitly bound to the nullifier of the old note;
- Its control key is the same `sk` as the old note;
- In other words, CommitChange is anonymous change for the same controller, not an anonymous output for some other party.

Statement 4: strict value conservation — old balance = payment + change

Finally, the circuit must prove:

balance_old = pay_balance + balance_change

where:

- `balance_old`: the old note's balance (private witness);
- `pay_balance`: the public payment amount (public input);
- `balance_change`: the private change amount.

This equation prevents:

- `balance_old > pay_balance + balance_change` (funds "disappearing"); and
- `balance_old < pay_balance + balance_change` (funds "appearing out of thin air").

3.6.6 Overall semantics of Anonymous Pay and implementation navigation Putting the four statements together, the semantics of this minimal Anonymous Pay are:

1. The prover demonstrates (in the zero-knowledge sense) that:
   - The old note is indeed a real leaf in MerkleTreeCommit;
   - The public `nullifier` is indeed computed from that note's `(sk, nonce_old, src_old)`.

2. At the same time, the prover shows that:
    - The new `CommitChange` is constructed using the same `sk`, with note structure `(balance_change, nonce=0, src=nullifier)`;
    - That is, CommitChange is anonymous change for the same controller, **not** an anonymous balance handed to someone else.
3. The prover also shows that:
    - The old balance `balance_old` is fully split into `pay_balance` (paid to the public address) and `balance_change` (change);
    - No value is created or destroyed in the process.

From the on-chain observer's viewpoint, one can only see:

- Some anonymous `nullifier` being used;
- A new AC (CommitChange) added to MerkleTreeCommit;
- A public address `ADDR_to` receiving a payment of `pay_balance`.

But mathematically, the system is assured that:

- The payer truly controls the old AC;
- The change AC CommitChange remains under the same controller;
- The recipient can only receive `pay_balance` via the public address, and cannot obtain an anonymous balance through any "new AC".

This is precisely how the principle
"no ownership transfer when creating new anonymous commitments"
is implemented in the Anonymous Pay mechanism.

Implementation navigation (ZK details for Anonymous Pay):

- Rust-side ZK proof generation and verification for anonymous pay:
  - `zkcrypto/src/zkproof_for_anon_pay_gen.rs`
  - `zkcrypto/src/zkproof_for_anon_pay_verify.rs`
- Protocol documents (circuits and logic for anonymous pay):
  - `docs/anon_pay/`
- Single tests calling the Rust ZK API (anonymous pay):
  - `tests/rust_api_tests/zkproof_anon_pay_tests.py`

3.7 Merging multiple anonymous commitments (2 → 1, self-merge)

> This part is not yet implemented in the prototype, but the principle is similar to Sections 3.6 and 3.5 and should not be difficult to add.

Consider the following typical "merge ACs" operation:

> In a single operation, consume 2 old anonymous commitments,
> create 1 new anonymous commitment,
> and the whole process is just the same holder reorganizing their own balance inside the

anonymous layer,
with no possibility of paying others or transferring ownership in the process.

In other words, this operation is essentially "internal consolidation bookkeeping", making it easier to later use a larger anonymous balance for payments.

3.7.1 Merge scenario and the structure of JointCommit Suppose the two commitments to be merged are $\texttt{AC}_1$ and $\texttt{AC}_2$, whose internal note structures in the circuit are:

- For $\texttt{AC}_1$, the corresponding $\texttt{note}_1$ fields are:
  - $\texttt{balance}_1$
  - $\texttt{sk}$
  - $\texttt{nonce}_1$
  - $\texttt{src}_1$
- For $\texttt{AC}_2$, the corresponding $\texttt{note}_2$ fields are:
  - $\texttt{balance}_2$
  - $\texttt{sk}$
  - $\texttt{nonce}_2$
  - $\texttt{src}_2$

Where:

- Both old notes use the same $\texttt{sk}$ (the circuit enforces this to ensure they belong to the same holder);
- Their nullifiers are defined as:
  - $\texttt{nullifier}_1 = \texttt{Poseidon(sk, nonce}_1\texttt{, src}_1\texttt{)}$
  - $\texttt{nullifier}_2 = \texttt{Poseidon(sk, nonce}_2\texttt{, src}_2\texttt{)}$

After merging, a new commitment is created, denoted $\texttt{JointCommit}$, whose new note fields are defined as:

- $\texttt{balance = balance}_1 \texttt{ + balance}_2$ (the merged total balance, as a secret input, i.e. the total after "consolidation");
- $\texttt{sk}$ = the same $\texttt{sk}$;
- $\texttt{nonce = 0}$;
- $\texttt{src = Poseidon(nullifier}_1\texttt{, nullifier}_2\texttt{)}$.

Thus the value of $\texttt{JointCommit}$ satisfies:

$$\text{JointCommit} = \text{Poseidon}(\text{balance}_1 + \text{balance}_2, \text{sk}, 0, \text{Poseidon}(\text{nullifier}_1, \text{nullifier}_2))$$

The meaning of this structure is:

- $\texttt{JointCommit}$ explicitly marks itself as "originating from the merge of the two old notes corresponding to $\texttt{nullifier}_1$ and $\texttt{nullifier}_2$";
- The controlling secret is still the same $\texttt{sk}$;
- Therefore, this new commitment is simply the same party merging two smaller anonymous balances into one larger anonymous balance, with no transfer of ownership to any third party.

3.7.2 What the ZK proof must guarantee (similar to Anonymous Pay) The ZK proof for the merge operation is conceptually very similar to the Anonymous Pay proof in Section 3.6 and can be described in natural language as the following statements.

3.7.2.1 The old AC_1 and AC_2 really exist in the current MerkleTreeCommit Inside the circuit:

- Using the private fields of $note_1$, $(balance_1, sk, nonce_1, src_1)$, compute:
  - $AC_1 = Poseidon(balance_1, sk, nonce_1, src_1)$;
- Using the private fields of $note_2$, $(balance_2, sk, nonce_2, src_2)$, compute:
  - $AC_2 = Poseidon(balance_2, sk, nonce_2, src_2)$;
- For each of AC_1 and AC_2, treat it as a leaf and, together with its private Merkle path, compute upwards in MerkleTreeCommit;
- Both paths must result in a root equal to the current `CommitRoot` in the public inputs.

This statement guarantees that:

- Both commitments being merged are real leaves in the current MerkleTreeCommit;
- They are not fabricated data outside the tree;
- Both belong to the same tree under the same current root.

3.7.2.2 Correctness of the public $nullifier_1$ and $nullifier_2$ The circuit must prove:

- Using $(sk, nonce_1, src_1)$ from $note_1$, compute:

  $Poseidon(sk, nonce_1, src_1) = nullifier_1\_public$

- Using $(sk, nonce_2, src_2)$ from $note_2$, compute:

  $Poseidon(sk, nonce_2, src_2) = nullifier_2\_public$

In other words:

> "The currently public $nullifier_1$ and $nullifier_2$
> do indeed come from $(sk, nonce, src)$ of the notes corresponding to AC_1 and AC_2 respectively,
> computed exactly as the protocol specifies."

This lets the chain:

- Record $nullifier_1$ and $nullifier_2$ as "spent", preventing AC_1 and AC_2 from being used again in the future.

3.7.2.3 JointCommit has the correct structure and is still controlled by the same sk The circuit must also prove:

- It uses the same sk as for AC_1 and AC_2;
- It introduces a private new balance $balance\_new$, and imposes the constraint:

balance_new = $\texttt{balance}_1$ + $\texttt{balance}_2$

- It then assembles the new note fields as:
  - `balance = balance_new`;
  - `sk` = the same `sk`;
  - `nonce = 0`;
  - `src = Poseidon(nullifier`$_1$`, nullifier`$_2$`)`;
- It applies Poseidon to these four fields and requires the result to equal the public `JointCommit`.

In natural language:

> "Using the same `sk` as for the two old notes,
> I add `balance`$_1$ and `balance`$_2$ to get a new balance `balance_new`,
> then form a new note with `balance = balance_new`, `nonce = 0`,
> and `src = Poseidon(nullifier`$_1$`, nullifier`$_2$`)`,
> and apply Poseidon to obtain the value that equals the currently public `JointCommit`."

This statement guarantees that:

- `JointCommit` is a structurally correct commitment;
- It binds to the two old notes via `nullifier`$_1$ and `nullifier`$_2$ as lineage;
- Its control key has not changed and is still the same `sk`.

3.7.2.4 Value conservation: the new balance equals `balance`$_1$ + `balance`$_2$ The circuit enforces internally that:

- `balance_new = balance`$_1$ + `balance`$_2$, and the `balance` used to construct JointCommit is exactly this `balance_new`;
- It disallows:
  - `balance_new > balance`$_1$ + `balance`$_2$ (creating value out of thin air);
  - `balance_new < balance`$_1$ + `balance`$_2$ (losing value for no reason).

This means:

> "The balances from the two old commitments
> are fully merged into the new commitment's balance.
> No value is lost, and no extra value is created."

3.7.3 Semantic summary of the merge operation Combining the above statements, the semantics of "merging two anonymous commitments into one" can be summarized as:

- The old `AC`$_1$ and `AC`$_2$ are real leaves in MerkleTreeCommit;
- Their `nullifier`$_1$ and `nullifier`$_2$ have been publicly revealed and correctly marked as spent;

- The new `JointCommit` corresponds to a note with:
  - balance $\texttt{balance}_1 + \texttt{balance}_2$;
  - control key `sk` identical to the old notes;
  - $\texttt{nonce} = 0$;
  - $\texttt{src} = \texttt{Poseidon}(\texttt{nullifier}_1, \texttt{nullifier}_2)$;
- The whole process does not involve any public address and does not generate an anonymous output for anyone else:

  It is purely the same holder consolidating two smaller anonymous balances
  into one larger anonymous balance inside the anonymous layer,
  to make subsequent larger-amount Anonymous Pay operations easier.

From the on-chain perspective, one can only see:

- Two nullifiers being consumed;
- A new `JointCommit` being inserted into MerkleTreeCommit;
- No new public-payment record appearing.

This matches the design goal that "merge operations for multiple anonymous commitments are restricted to self-merges and cannot be used to pay others."

3.8 Hidden-amount payments between blue addresses (commit-style payments with mandatory backup)

> This section describes a **planned feature** in the protocol design.
> It is not yet implemented in the current Python + Rust single-node prototype and exists only at the specification-draft level.

Scenario: a blue address Alice (payer) sends funds to a blue address Bob (payee):

- Account identities are public (who pays whom is visible on chain);
- The amount is hidden inside a commit, so external observers cannot see the precise value;
- The protocol **requires** that Alice must back up the change commit's amount and blind, and must prove in ZK that she has indeed done so.

Concretely, the constraint is:

> Every hidden-amount payment must include a ZK proof showing that:
> "When I generated this change commit `commitChange`,
> I used the current paying account's nonce `N_new` as the `counter` field,
> and using my `SK_A`,
> I encrypted the `balance` and `blind` of `commitChange` into two on-chain ciphertexts `S1` and `S2`."

The ZK circuit does **not** care about the structure of Bob's receiving commit, nor about the correctness of the receiving ciphertext `messageR`.
Those are handled by the payee and consensus logic outside of ZK.

3.8.1 Commit format and ownership model For the "hidden amount but public identity" setting, each blue account maintains a **List of Commits**.
Each commit is a Poseidon commitment whose preimage has three fields:

- `value`: the amount (private);
- `blind`: the blinding factor (private);
- `counter`: a counter (public);

defined as:

commit = Poseidon(value, blind, counter)

Ownership rule:

- The ledger explicitly records "which account a given commit belongs to";
- The preimage of a commit does not itself contain a public key field;
- As long as you can prove you control the private key `SK_A` corresponding to the account's public key `PK_A`, you can spend all commits attached to that account.

Additional fixed rule (specific to this section):

- In "blue-address hidden-amount payments", all newly created commits must use, as their `counter` field, the account nonce `N_new` used in the transaction (i.e. the `nonce` in the outer transaction envelope);
- The contract layer separately checks that `N_new` satisfies the `++nonce` rule defined in the previous section.

3.8.2 A typical blue-to-blue payment: consume old commit → pay Bob → get change This section focuses on the most typical case:

Alice consumes one old commit,
gives Bob a new receiving commit,
and obtains a new change commit `commitChange` for herself.

3.8.2.1 Initial state

- Alice's current account nonce is `N_old`;
- In Alice's List of Commits there is an old commit, denoted `commit_in`, whose preimage is:
  - `value_in` (old amount, private);
  - `blind_in` (old blind, private);
  - `counter_in` (old counter, public and recorded in the ledger);
- And they satisfy:

commit_in = Poseidon(value_in, blind_in, counter_in)

- Alice's private key is `SK_A`, with corresponding public key `PK_A`.

3.8.2.2 Choosing the payment amount and change amount (strict equality) Alice chooses:

- The amount to pay Bob, `value_pay`;
- The change amount, `value_change`.

The protocol enforces **strict equality** for value conservation:

value_in = value_pay + value_change

These two values only exist locally on Alice's side and in the ZK circuit; they are not public on chain.

3.8.2.3 Bob's receiving commit and `messageR` (parts that are outside ZK) Alice constructs a receiving commit for Bob (`commit_pay`) and a ciphertext `messageR` for Bob.
The high-level idea is:

- Derive a shared symmetric key from Alice's and Bob's key material;
- Encrypt `value_pay` (and any necessary auxiliary information) with that symmetric key to obtain `messageR`;
- `commit_pay` and `messageR` are validated by Bob and the consensus logic outside ZK using standard cryptography.

Important:

- In the ZK proof in this section:
  - The structure of `commit_pay` is **not** verified;
  - The correctness of `messageR` is **not** verified;
- They are handled purely outside this ZK circuit.

3.8.2.4 Constructing Alice's change commitChange and mandatory backups `S1 / S2` (the part enforced by ZK) This is the part the ZK circuit must strictly enforce.

1. **Determine the account nonce used in this transaction:**
   - The new account nonce is defined as `N_new = N_old + 1`;
   - The transaction envelope carries `N_new` as its `nonce`;
   - The consensus layer will separately check that `N_new` equals the on-chain `N_old + 1`.
2. **Construct the preimage of the change commit `commitChange`:**
   - `value_change` (private);
   - `blind_change` (new blind, private randomness);
   - `counter = N_new` (public);

   Compute the change commit:

   commitChange = Poseidon(value_change, blind_change, N_new)

This commitChange is then attached to Alice's List of Commits as a new anonymous balance.

3. **Generate mandatory backup ciphertexts `S1 / S2`:**
    - Use Alice's private key SK_A (or a symmetric key derived from SK_A via a public rule) as the key for symmetric encryption;
    - Choose a ZK-friendly symmetric primitive (see Section 3.8.5), denoted SymEnc;
    - Define:
        - `S1 = SymEnc_key(SK_A)(value_change)`: backup for the change amount;
        - `S2 = SymEnc_key(SK_A)(blind_change)`: backup for the change blind;
    - Both S1 and S2 are published on chain as public data.

Later, as long as Alice still controls SK_A, she can:

- Decrypt S1 with SK_A to recover value_change;
- Decrypt S2 with SK_A to recover blind_change;
- Combine them with the public `counter = N_new` to reconstruct the full preimage `(value_change, blind_change, N_new)` of commitChange.

3.8.3 ZK proof: public inputs / private inputs The ZK proof for this "hidden-amount payment" enforces the following:

- The preimage of the old commit_in really exists;
- Strict value conservation: `value_in = value_pay + value_change`;
- The counter used in the change commit commitChange is exactly the account nonce N_new for this transaction;
- S1 and S2 are indeed produced by encrypting value_change and blind_change with SK_A using the agreed ZK-friendly symmetric primitive.

3.8.3.1 Public inputs The circuit's public inputs can be summarized as:

- commit_in: the old commit being consumed;
- counter_in: the counter of the old commit (read from the account state);
- commitChange: the value of the new change commit;
- N_new: the paying account's nonce from the transaction envelope (the consensus layer will check its ++nonce relation with the account state);
- S1: the symmetric ciphertext backing up value_change;
- S2: the symmetric ciphertext backing up blind_change;
- PK_A: the payer account's public key (used to bind the private key SK_A in the circuit to this account identity).

Note:

- There is no commit_pay here;

- There is no `messageR` for Bob;
- Neither enters this ZK proof.

3.8.3.2 Private inputs (witness) The witness values in the circuit (known only to the prover) include:

- `value_in`: the old commit's amount;
- `blind_in`: the old commit's blind;
- `value_pay`: the payment amount for this transaction;
- `value_change`: the change amount for this transaction;
- `blind_change`: the blind for the change commit;
- `SK_A`: the payer's private key;
- Any intermediate values used by the symmetric primitive `SymEnc` (depending on the chosen ZK-friendly algorithm; these are implementation details internal to the circuit).

3.8.4 Statements the circuit must prove (logical breakdown) The circuit must jointly satisfy the following constraints:

1. **Correct structure of the old `commit_in`**
   - Inside the circuit, use witness `value_in`, `blind_in` and public `counter_in` to compute:

     Poseidon(value_in, blind_in, counter_in)

   - The result must equal the public `commit_in`.

   Meaning:

   > The prover truly knows the full preimage of the old commit being spent,
   > and is not arbitrarily claiming some unrelated commitment as theirs.

2. **Strict value conservation: `value_in = value_pay + value_change`**
   - The circuit checks that the three witness values satisfy:

     value_in = value_pay + value_change

   Meaning:

   > The amount in the old commit is completely split into:
   > — the payment amount `value_pay` to Bob, and
   > — the change amount `value_change` returning to Alice's own balance,
   > with neither extra value appearing nor any value disappearing.

3. **Correct structure of the change `commitChange`, with counter = `N_new`**
   - Using witness `value_change`, `blind_change` and public `N_new`, compute:

     Poseidon(value_change, blind_change, N_new)

- The result must equal the public `commitChange`.

Meaning:

- `commitChange` is indeed computed from `value_change`, `blind_change`, and the account nonce `N_new` for this transaction;
- The `counter` in the change commit matches the transaction envelope's `nonce`;
- The contract layer then checks that `N_new` equals on-chain `N_old + 1`, thereby aligning the account-level ordering control with the commit-level `counter`.

4. **`SK_A` really corresponds to the paying address's public key `PK_A`**

- In the circuit, use the witness `SK_A` to compute an expected public key `PK_A_expected` on the elliptic curve;
- Require `PK_A_expected` to equal the public `PK_A`.

Meaning:

> All behavior in the circuit is carried out under the identity
> of "the holder of `PK_A`".
> The prover is not simply an external attacker trying to use someone else's `commit_in`,
> nor an anonymous outsider with no relation to the account.

5. **`S1` is produced by ZK-friendly symmetric encryption of `value_change` with `SK_A`**

- In the circuit, use `SK_A` (or a symmetric key derived from it via a public rule) as the key for the symmetric primitive `SymEnc`;
- Apply `SymEnc` to `value_change` (possibly via one or several rounds as specified by `SymEnc`) to obtain `S1_expected`;
- Require `S1_expected` to equal the public `S1`.

Meaning:

- `S1` must be the result of "encrypting `value_change` with Alice's key using `SymEnc`";
- It is not arbitrary data nor an attacker's random string;
- As long as Alice keeps control of `SK_A`, she can recover `value_change` from `S1`.

6. **`S2` is produced by ZK-friendly symmetric encryption of `blind_change` with `SK_A`**

- Similarly, use the same `SK_A` (or its derived symmetric key) as the key for `SymEnc`;
- Apply `SymEnc` to `blind_change` to obtain `S2_expected`;
- Require `S2_expected` to equal the public `S2`.

Meaning:

- `S2` is likewise a strictly constrained ciphertext;
- It must be "encrypting the change blind `blind_change` with Alice's key using `SymEnc`";
- As long as Alice retains `SK_A`, she can recover `blind_change` from `S2`, and with the public `N_new` reconstruct the entire preimage of `commitChange`.

3.8.5 Choice of symmetric primitive and implementation status In this draft, `SymEnc` is **not** pinned down to one specific algorithm at the protocol level, for two main reasons:

1. **AES-like algorithms are typically not "ZK-friendly" in circuits**
   - They work on bytes, with complex S-boxes and arithmetic over GF(2^8);
   - In Halo2-style prime-field constraint systems, this usually requires heavy bit-level decomposition or large look-up tables, making proofs expensive.
2. **Within Halo2 and the current zkcrypto architecture, a more natural approach is "field-friendly" symmetric primitives**
   - For example, ciphers built from SNARK-friendly permutations such as Poseidon, Rescue, MiMC, etc.;
   - These primitives work directly over the scalar field and can reuse existing Poseidon circuits (like this project's `poseidon_chip`), significantly reducing implementation complexity and proof cost.

Therefore, in this WhitePaper Draft, the requirements for `SymEnc` are:

- It must be a symmetric encryption scheme that is easy to implement in Halo2 and ZK-friendly;
- A recommended implementation path is:
  - Build a simple PRP or sponge mode from the existing Poseidon circuit over the same field;
  - Let the same `poseidon_chip` support both hashing and encryption to reduce circuit diversity;
- Detailed parameter choices and security analysis for the concrete algorithm are left to later engineering stages and will be added to `docs/anon_pay/` and the related circuit documentation.

At present, this whole mechanism of
"hidden-amount blue-address payments + mandatory S1/S2 backups + ZK constraints"
is **not** implemented in the prototype:

- There is no corresponding `zkproof_for_blue_hidden_pay_*` source file on the Rust side;
- There is no corresponding wrapper, client demo, or tests on the Python side;
- This section should be viewed as a conceptual spec for a future "blue-account private payment module".

3.9 Future extensions: fast payment channels and custody fees in the non-anonymous layer

The functionality in this subsection is not implemented in the prototype yet.

In the long run, my_coin plans to support a class of **fast payment channels** similar to the Bitcoin Lightning Network, used to implement low-latency, small-value, high-frequency

payments in an off-chain or semi-off-chain manner. Like Lightning, this layer is better suited to everyday spending and micropayments, while the on-chain anonymous layer remains responsible for larger amounts and privacy-preserving payments.

In designing these fast-payment features, my_coin makes a deliberate trade-off:

- The **non-anonymous layer** where fast payment channels live does **not** support hidden payment amounts;
- That is, on this "fast path":
  - Who pays whom is visible;
  - The payment amount itself is also transparent;
  - Real "amount privacy" only happens in the anonymous layer.

This is not because it is technically impossible to hide amounts there, but rather due to a combined consideration of system-wide fund distribution and regulatory acceptability.

3.9.1 Balance between anonymous / non-anonymous layers and dynamic custody fees If fast payment channels are too convenient and too cheap, a realistic extreme could occur:

> More and more users keep most of their funds long-term in the non-anonymous layer and in payment channels,
> causing the total funds in the anonymous layer to shrink, the anonymity set to shrink, and overall anonymity to degrade.

To avoid the anonymous layer gradually "hollowing out", my_coin introduces a **time-based custody fee** on funds in the non-anonymous layer (you can think of it as a "storage fee" for funds parked outside the anonymous layer), and this fee rate is **dynamic**:

- Let `A_anon` be the total assets in the anonymous layer, and `A_non_anon` be the total assets in the non-anonymous layer (including fast payment channels);
- Roughly speaking:
  - When `A_anon` is relatively large, the per-unit-time custody fee in the non-anonymous layer can stay low;
  - When `A_anon` shrinks and fewer funds remain in the anonymous layer, the protocol will **automatically increase the custody fee** in the non-anonymous layer;
- The intuitive effect is:
  - Users are encouraged to keep their **main holdings in the anonymous layer**;
  - The non-anonymous layer behaves more like a "short-term working account" and "fast payment buffer zone";
  - Users who keep large balances in the non-anonymous layer for long periods will face increasingly unfavorable storage costs.

Under this mechanism, a "non-anonymous fast payment channel without hidden amounts" becomes a controlled tool:
the protocol can always roughly observe the ratio between anonymous-layer and non-anonymous-layer funds and use the dynamic custody fee as a lever to pull funds back into the anonymous layer,

maintaining a sufficiently large anonymity set and a good privacy baseline.

3.9.2 Alternative route: let anonymous pay also support hidden amounts (and why this is not adopted yet) From a purely technical and UX perspective, there is an appealing alternative path:

- Let the anonymous pay functionality itself support **hidden-amount payments** (i.e. both "anonymous identity + hidden amount" together);
- Charge only a **constant** per-unit-time custody fee in the non-anonymous layer, instead of dynamically adjusting it based on the anonymous-layer fund ratio;
- This would lead to two parallel usage patterns:
  - Non-anonymous layer + fast payment channels: for high-frequency, small-value, regulation-heavy scenarios;
  - Anonymous layer + hidden-amount payments: for higher-privacy, larger-amount scenarios.

In such a design, as long as sufficient funds continue to participate in the anonymous layer, **anti–money-laundering effects can still be strong**, and the system would have very powerful privacy capabilities and flexibility. However:

- From a regulator's viewpoint, a channel that combines "anonymous identity + hidden amounts" may be more easily used for various forms of tax evasion and similar behavior;
- Even if the protocol incorporates complex mechanisms for reporting, responsibility sharing, and accountability, the real-world level of acceptance remains uncertain.

After weighing these factors, my_coin's current roadmap **prioritizes the version more likely to be acceptable to regulators**:

- Fast payment channels and the non-anonymous layer do not hide amounts;
- A dynamic custody fee steers users toward keeping main holdings in the anonymous layer and using anonymous pay more;
- The more aggressive "anonymous identity + hidden amount" design is retained as a **theoretically feasible, future extension**, rather than being activated in the first version of the protocol.

This means:

> The current my_coin protocol draft leans toward
> "finding a practically workable balance between privacy capability and regulatory acceptability,"
> rather than immediately pushing anonymity to its theoretical maximum, which would also be hardest to get accepted in mainstream regulatory environments.

3.10 Receipt confirmation mechanism: defending against "account-pollution-style" attacks

> This subsection describes a protocol-level design idea that is not yet implemented in the current Python + Rust single-node prototype.

In earlier design parts, the focus was mostly on "who is paying" and "how funds can be traced afterward".
There is another real-world attack mode that needs special defense:

> An attacker obtains a batch of illicit funds, not to launder it for themselves,
> but to deliberately spray that money into many unrelated blue addresses,
> dragging innocent users into lawsuits and investigations —
> a DoS-like "account pollution" attack.

To mitigate such "account pollution" risks, the protocol includes a **"confirm-to-receive"** mechanism to distinguish between:

- **"I was just spammed with money; I never intended to receive it"**, and
- **"I willingly and deliberately accepted this payment"**.

3.10.1 Basic semantics: unconfirmed incoming funds can be directly returned In a version that supports "receipt confirmation", a payment to a blue address goes through two stages:

1. **Arrival stage**:
   - The payment transaction is already on chain;
   - The ledger records "X paid amount A to Y" (transparent or via anonymous entry as per previous rules);
   - But this amount is in an "unconfirmed incoming" state on Y's account.
2. **Receipt-confirmation stage**:
   - If the recipient believes this payment is **funds they were supposed to receive** (salary, compensation, refunds, etc.),
   - they must use their address's private key `SK_Y` to sign a "fingerprint" of this payment (e.g. transaction hash + amount + their own address),
   - and broadcast a "receipt confirmation" message:
     > "I (address Y) confirm that this payment A from X is funds I willingly accept."
   - This signature is recorded on chain or in equivalent contract state.

Under this mechanism:

- **If a payment has never been confirmed by the recipient** and is later reported/recognized as illicit:
  - The chain can simply "freeze" that payment,
  - It does not enter the complex responsibility-sharing process from Chapter 4 ("who is the first hop, who must point to next hop", etc.);

  - The recipient is treated, in protocol terms, as having "been spammed with funds and never accepted them".
- **Only payments that have been confirmed by the recipient** will enter the full responsibility game:
  - Who is the first-level holder, who needs to identify downstream recipients with their SK,
  - Who only took a commission, who is liable for the whole illicit amount,
  - All are handled under the rules in Chapter 4.

As a result, simply "spraying dirty money into random blue addresses" is much less effective at dragging innocent users into convoluted disputes — because as long as they never sign a "receipt confirmation",
both the protocol and courts can treat the funds as:

> Refunded back to the origin, and no pollution has actually been established.

3.10.2 Abuse and accountability Of course, this mechanism itself leaves a full trace to prevent abuse:

- If someone **receives normal payments for genuine services**, but then tries to exploit the mechanism:
  - Pay first, then quickly request a "wrong transfer" refund before the other party can confirm receipt;
  - Such behavior leaves a public on-chain history of "repeated paying and repeatedly retracting";
  - Whether this constitutes breach of contract or fraud is left to the courts, using the on-chain evidence.
- If a party **has already confirmed receipt** (signed to accept) and later claims "I was just polluted":
  - They will immediately fall under the responsibility-allocation rules in Chapter 4;
  - They can no longer hide behind "unconfirmed" status to avoid liability.

Overall:

- **The receipt-confirmation signature = the recipient's subjective consent to the funds**;
- For unconfirmed incoming funds that are recognized as illicit, the protocol defaults to "returning them to the source" and shields the recipient from forced pollution;
- For payments that have been confirmed, the process follows Chapter 4's rules for explaining sources and assigning responsibility.

This mechanism is currently only a protocol-level idea.

If implemented later, a more detailed state machine and transaction format will be specified in the `docs/` directory.

4. Why this design for AML: attack–defense and motivation

After reading Chapter 3, you may have several intuitive questions:

- Why do we need a whole **masterSeed + token** scheme, instead of simply "digital ID + arbitrary address signatures"? Why does the protocol *force* all blue addresses to be derived from the masterSeed?
- Why is it that **acct→anon is not allowed to change `sk`** when generating an anonymous commitment, and AnonPay also forbids "anonymous → anonymous", only allowing "anonymous → address + self change"?
- Why must hidden-amount payments between blue addresses *mandatorily* produce those two seemingly "verbose" backup ciphertexts S1 / S2?
- How are all these strange rules actually related to "anti–money laundering"? How exactly do they make money laundering "unsafe"?

The goal of this chapter is: step by step explain **why my_coin can help with AML**, and at the same time answer the design motivations behind these "odd-looking" rules.

4.1 Start with the simplest question: can a lone hacker clean money using my_coin?

First, assume a hacker **does not use any intermediaries** and just wants to rely on my_coin alone to launder Alice's money. Let's see what paths are available, and why none of them work well.

4.1.1 Using only red addresses: essentially no better than "Bitcoin without mixers" If the hacker only routes funds through red addresses (fully transparent layer):

- The entire payment path is transparent to everyone;
- This is on the same level as "using non-mixed Bitcoin to launder money":
  - Entry, exit, and path are all drawn on-chain in front of you;
  - It is simply a different chain.

From an AML perspective, **my_coin does not help the hacker become harder to trace; it even adds extra KYC information**. This path offers the hacker no particular advantage.

4.1.2 Anonymous payment directly from Alice's blue address to the hacker's blue address: hidden from third parties, not hidden from Alice A slightly "smarter" hacker would try to use Alice's anonymous payment capability:

> Take the stolen funds from Alice and anonymously pay them into the hacker's own blue address B_evil.

Third-party observers see:

- "Some anonymous source paid an amount X (amount may or may not be hidden externally, identity is definitely hidden) to B_evil."

At first glance, this looks perfect: nobody can see that the money came from Alice.
But the problem is — **Alice herself is not "everybody else"**; she has her own SK.

In my_coin, the anonymous layer is deliberately designed so that:

- As long as Alice still holds her SK;
- She has the ability to **trace all anonymous payments originating from her own accounts**:
  - Which acct→anon operations deposited how much into the anonymous pool;
  - Which AnonPay operations spent which portion of that balance to which addresses (red / blue);
  - All intermediate change commits / merges / re-spends can be reconstructed locally step by step using her SK.

This is exactly the effect of those "stubborn" rules from earlier:

- In acct→anon, the `sk` used in the anonymous commitment is enforced by the ZK circuit to be the address's `sk`;
- AnonPay only allows "anonymous → public address + self change", and all internal evolution in the anonymous layer is rigidly tied to `sk / nullifier / src`;
- All of these equalities are wired into the circuit — the prover cannot "silently swap in another SK" at some intermediate step.

What is the result?

- For third parties: they cannot see that "this was Alice paying B_evil";

- For Alice herself:
  - As long as she still remembers her password / SK;
  - She can use her own keys to reconstruct this anonymous flow all the way to the blue address B_evil;

- Once Alice reports the theft and hands her SK and local proofs to law enforcement, **the first hop B_evil becomes extremely clear in the evidential trail**.

And blue addresses are:

- KYC'ed real-world identities;
- Bound to a single real-world ID via masterSeed + token;
- There is no situation where "the chain says it's Bob, but the SK is actually in a hacker's hands" (this will be discussed separately below).

So for a "solo" hacker:

- If they want to use my_coin's anonymous features;
- Either they avoid blue addresses entirely, in which case they do not have anonymous payment capability and fall back to red addresses (= transparent chain);

- Or they use their own blue address, in which case, as long as the victim is willing to pursue the case, **the first hop lands directly on the hacker's real-world identity**.

Conclusion: **For a lone actor, my_coin does not offer better laundering conditions than "non-mixing Bitcoin + KYC exchanges".**

To avoid being the first hop, the hacker must pull in intermediaries to "buffer" a few layers of identity. That brings us to the next section.

4.2 Professional launderers: why are they "structurally unsafe" in my_coin?

In the real world, money laundering almost inevitably relies on a chain of intermediaries ("mules", "jump addresses", "water rooms").
my_coin obviously cannot prevent people from recruiting intermediaries, but it makes **each layer of intermediaries extremely unsafe**, to the point that they are reluctant to participate.

We'll break this into two parts:

1. Why are intermediaries easy to lock down to a specific hop?
2. Once locked down, why can they almost never "stonewall and say nothing"?

4.2.1 Why are intermediaries easy to lock down to specific hops? There are three key structural features:

1. **Blue accounts must be KYC'ed + bound by usage terms**
2. **A single real-world ID can have only one masterSeed + many derived addresses**
3. **All evolution in the anonymous layer is bound by SK + public fields**

We'll go through them quickly.

(1) Blue accounts are KYC'ed + governed by usage terms

- Blue addresses must go through Clerk-based offline KYC;
- The protocol requires users to sign usage terms, in which they:
  - Agree that, within their "responsibility cap", they bear civil liability for stolen / suspicious funds received by their account;
  - If they claim "I am only an intermediary", they must use their own SK to identify the next hop.

So behind each blue account lies a concrete person/entity, together with a very explicit civil liability model.

(2) Each real-world ID has only one masterSeed

- Each real-world ID can bind only one `master_seed_hash`;
- All blue addresses under that ID have SKs derived from this masterSeed + different tokens;
- This implies:
  - As long as this person remembers their password, they can enumerate all of their blue-address SKs;

- The protocol **forbids** any blue address where "on-chain the real-name is Bob, but only a hacker holds the private key".

This directly shoots down the "digital ID + arbitrary address signatures" design: that approach encourages a world where "ID belongs to Alpha, SK belongs to Beta". In my_coin, that is structurally disallowed.

(3) All evolution in the anonymous layer is structurally constrained by SK

- acct→anon: the ZK circuit requires that the anonymous commitment's `sk` equals the account's `sk`;
- Anonymous merges: both commitments must use the same `sk`, and the merged commitment must also use that same `sk`;
- Anonymous payments (AnonPay):
  - Can only pay to public addresses;
  - The anonymous side can only generate a single change commit_change;
  - The ZK circuit enforces that commit_change's `sk` is the same as the `sk` of the spent commitment, and `src` is strictly derived from the nullifier and other public fields.

The combined effect of these three features is:

- Once a victim comes forward and uses her SK to reconstruct "all anonymous flows originating from her", up to the first blue recipient Blue_1;
- From the protocol's point of view:
  - Blue_1 must be a KYC'ed, real-name subject;
  - All blue addresses under Blue_1 must be derived from a single masterSeed;
  - All of Blue_1's anonymous asset evolution paths are mathematically determined by "Blue_1's SK + a bunch of public fields".

Therefore, **"who is the first-hop intermediary" can be pinned down to a specific real-world account very easily.**
This is very different from Zcash, where you only know "funds went into the pool"; who exactly is inside the pool and how they move is entirely unanchored in identity.

4.2.2 Once an intermediary is identified, why can't they just "stonewall"? Once locked down, the intermediary faces only two options:

1. Stonewall: say nothing;
2. Confess: use their SK to identify downstream participants.

This is where the "annoying" protocol constraints we introduced earlier start to matter.

(1) Cost of stonewalling: in principle, they may be liable for the entire stolen amount (up to their responsibility cap)

- The usage terms are very explicit:
  - Once a particular incoming payment is identified as stolen / suspicious;
  - Under the terms, within the "liability cap" they agreed to, they owe a civil refund;

- If they insist "I forgot my password / I don't know my SK / I don't acknowledge this money is controlled by me":
  - The protocol does *not* automatically let them off the hook because they "technically cannot decrypt the chain";
  - On the contrary, the fact they cannot explain their own anonymous assets makes them look worse — in the worst case, the system may treat them as the first accountable party for that stolen amount, within their agreed liability cap.

In other words:
"I forgot my password" is not an excuse in my_coin. It actually makes it harder for you to demonstrate your innocence.

(2) To reduce civil liability, they must "name the next hop"

If someone claims "I am only a laundering intermediary and only kept a 5% fee", they can theoretically reduce their liability, but at the cost of:

- Using their SK to:
  - Reconstruct the flow of the stolen funds inside their anonymous holdings;
  - Prove to the court that:
    - They indeed received amount X;
    - They kept Y% as their own fee/commission;
    - The remainder was transferred to specific downstream addresses;
- They must identify these downstream blue addresses (i.e., "name the next hop").

This step cannot be done without SK:
Because the evolution of anonymous commitments and nullifiers is hard-wired to `sk / src / nonce` inside the circuit, someone without SK cannot fabricate a plausible narrative of anonymous flows that stands up to cryptographic scrutiny.

Thus the intermediary's game becomes:

- If they stonewall:
  - Within their liability cap, they are likely responsible for the whole stolen amount;
- If they want to be responsible only for the small commission they actually pocketed:
  - They must use their SK to push the chain forward and reveal the downstream recipients.

This contrasts sharply with traditional systems, where statements like "I have no idea; the money just passed through me" are a common way to shift blame. In my_coin, that route is structurally very difficult.

4.2.3 Sting operations / entrapment: why professional launderers are even *more* at risk in this system So far we have discussed situations where a real victim comes forward.
We can go a step further: in my_coin, it is very easy for law enforcement to conduct sting operations.

A simplified scenario:

1. The police themselves control a blue account Victim_0 and send a "pseudo-stolen" amount into the anonymous pool;

2. They use social engineering / black-market channels to find a suspected professional launderer Blue_1 and ask them to "clean" this money;
3. Blue_1 uses my_coin to route the funds through some anonymous paths, taking a small fee;
4. Later, the police, now acting as "Victim_0", file a complaint:
    - Using Victim_0's SK to precisely reconstruct the anonymous path;
    - Showing the court that the first-hop intermediary is Blue_1;
5. Blue_1's choices are exactly those described above:
    - Either admit being a laundering intermediary and name downstream participants;
    - Or, within their liability cap, take responsibility for the entire "pseudo-stolen" amount.

tldr: **Every time you attempt to earn money from laundering inside my_coin, you are writing an anonymous path onto the chain that "can be precisely reconstructed at any time by an SK".**

In the long run:

- The more frequently you participate, the higher the probability that:
    - You will be exposed by a real upstream victim tracing the flow;
    - Or you will be named by downstream actors when they are caught;
    - Or you will be caught in a sting operation by law enforcement.
- Each time you only earn a tiny commission, but if something goes wrong, you face an extreme event of "high liability up to the cap + potential criminal exposure".

This is the real meaning of "professional launderers are structurally unsafe in my_coin".

4.3 Re-examining the "weird rules": why none of them can be removed

Now we can go back and answer those initial "this is so weird" design choices:

4.3.1 masterSeed + token: forbidding "real-name belongs to one person, SK belongs to someone else"

- Goal: ensure that "if a blue address is in someone's real name, then that same person holds the SK";
- Once that is guaranteed:
    - If any blue address is identified as a hop on a laundering path;
    - We can assume that this real-world person **must** hold the SK for that family of blue addresses;
    - Therefore, they **must** be able to reconstruct their own segment of the anonymous path — if they want to exonerate themselves, they must present their SK and evidence, not just say "I don't know".

4.3.2 Forbidding `sk` changes in acct→anon: making sure the anonymous entry never escapes the original owner's "field of view"

- Goal: ensure that the victim can use her own SK to reconstruct "how the money entered the anonymous pool";

- If acct→anon were allowed to arbitrarily switch `sk`, a hacker could:
  - Deduct money from victim address A;
  - When depositing into the anonymous pool, immediately use his own `sk'`;
- From victim A's perspective, the resulting anon_commit would have nothing to do with her SK and would **disappear from her view from the very first step**.

By enforcing anon_commit.sk = from_addr.sk, my_coin blocks this attack.

4.3.3 AnonPay only allows "anonymous → public address + self change": forbidding "ownership changes" inside the anonymous layer

- Goal: ensure that all "shapes" inside the anonymous layer are still predictable to the previous owner;
- If AnonPay were allowed to output an anonymous commit controlled by the recipient, that commit would necessarily swap to a new SK;
- Once "free SK swapping outputs" are allowed in the anonymous layer:
  - A hacker only needs to swap SK in this step to his own SK;
  - From the old owner's perspective, the path is broken;
  - Subsequent splitting / merging / routing inside the pool is completely invisible to the old owner.

my_coin simply forbids this at the protocol level:

- Anonymous spending can only pay public addresses;
- The anonymous side may only produce a self-owned change commit_change, with SK unchanged;
- In this way, anyone who ever held a particular SK inherently has the ability to see the full evolution of anonymous flows under that SK.

4.3.4 Hidden-amount payments between blue addresses + mandatory S1 / S2 backups: preventing "playing dumb afterwards"

- Hidden-amount payments between blue addresses aim to: **hide the amount, but not the identity**;
- The role of S1 / S2 is:
  - Use a ZK-friendly symmetric primitive (SymEnc) with SK_A as the key;
  - Encrypt `value_change` and `blind_change` for the change commit;
  - Prove inside the circuit that "the on-chain S1/S2 were indeed obtained in this way".

The purpose of this design is:

- In the future, if Alice wants to explain in court "exactly how much I paid and how much change I got in that transaction", she has:
  - Her SK_A;
  - On-chain S1 / S2;
  - And commitChange;
- These three together are sufficient for her to reconstruct locally the exact amount and change for that payment and provide verifiable evidence.

In other words: **S1 / S2 are there to prevent the "I forgot / I don't remember how much I paid" style of post-fact denial**, and to embed "accountability" into the protocol.

4.4 Comparison with Zcash: why hackers there do not need intermediaries at all

Finally, we can summarize the difference in the bluntest way:

> In Zcash and similar "pure privacy coins", a hacker can launder funds alone at a keyboard.
> In my_coin, if a hacker wants to launder money, **they must at least recruit a chain of intermediaries willing to gamble with their lives**,
> and those intermediaries are structurally very fragile.

4.4.1 In Zcash, a hacker does not need intermediaries A typical path is:

1. Receive some ZEC from the victim's address;
2. Shield the funds into a z-addr (the anonymous pool);
3. In the pool, split / shuffle / merge arbitrarily over several rounds;
4. After a while, withdraw from another z-addr / t-addr somewhere else and cash out.

Throughout this process:

- No real-name / KYC is required;
- No intermediaries are needed to "lend their identity";
- There is no protocol-level notion of "you signed terms and have civil liability for these funds";
- Whether to reveal a viewing key or not, and when, is entirely up to the hacker.

This is exactly the world you described where **"you do not need anyone, do not need to trust anyone; you just type a few lines and you are done laundering".**

4.4.2 In my_coin, hackers must find intermediaries, and those intermediaries live dangerously
In my_coin, if a hacker wants to avoid being the "first-hop blue account", they must:

- Let the stolen funds enter someone else's blue account;
- Use that person's anonymous payment capability to route around in the pool;
- Only appear closer to the end of the chain (for example, via OTC trades, illegal exchanges, etc., to receive the final output).

But all of these intermediate blue accounts:

- Are KYC'ed real identities;
- Have a unique masterSeed;
- Are governed by a full "liability cap + usage terms" structure at the protocol level;
- Have both the ability (and the incentive) to reconstruct their segment of the anonymous chain using their SK.

Combined with "victims tracing via SK" and "sting operations by law enforcement", this means:

- For any professional launderer:

- Each laundering job writes a path to the chain that may later become hard evidence;
- If any layer fails, they may be named by upstream victims or downstream participants;
- For a small commission, they assume the risk of hitting a liability cap event or even criminal charges.

This is the fundamental difference between my_coin and Zcash in AML capability:
my_coin does not hide a "regulator backdoor key" in the chain; instead, it uses a combination of identity, SK, ZK, and usage terms to make money laundering a strongly negative-expected-value business from a game-theoretic perspective.

4.5 Limits and honest disclaimer: this is not an "absolutely safe" AML scheme

The previous sections mainly explained how, under **rational participants + a reasonably functioning judicial environment**, my_coin turns money laundering into a strongly negative-expected-value business. But we must be explicit: this does *not* mean that:

> "Once my_coin is used, every laundering chain can be broken 100% of the time."

In reality, there is a difference between criminal and civil liability, and there are irrational or misinformed participants. In such scenarios, my_coin has several important limitations that must be acknowledged.

4.5.1 Once an intermediary is caught, they do not necessarily have an incentive to "name the next layer" From a protocol standpoint, once an intermediary is identified, they face a seemingly simple but in fact sharp binary choice:

1. **Admit being a laundering intermediary and name the next layer**
   - Benefit: they can try to be liable only for the small portion they personally received as commission;
   - Cost: they explicitly admit "I am providing laundering services."
     - In most legal systems, this exposes them to specific criminal charges (money laundering etc.), with significant prison risk.
2. **Stonewall and refuse to admit "I am an intermediary", only admit "this money is under my name"**
   - The protocol + usage terms can only specify **civil** liabilities, not criminal sentences;
   - If they refuse to admit being an intermediary, they may argue in criminal court that this is "unexplained income" rather than a structured laundering operation (the exact classification is up to the court);
   - The cost is: within their pre-agreed "liability cap", they must undertake civil refund liability for the stolen funds — even if they actually only kept a small commission.

So for an intermediary who has been caught, it is not as simple as "they will definitely name the next layer".
In many legal environments:

- **Admitting to being a laundering intermediary = higher chance of doing prison time**;
- **Stonewalling and only taking civil responsibility = paying more money, but possibly lower criminal risk**;

This means some intermediaries, once caught, may rationally choose to "lose more money but reduce prison risk" and thus refuse to reveal downstream participants.
Therefore:

> In the worst case, a laundering chain may be blocked at some intermediary layer, and not fully broken through to the ultimate beneficiary.

This is something my_coin cannot solve with cryptography and must be stated honestly in the whitepaper.

4.5.2 Even if the chain is not fully broken, victims still have two layers of "baseline protection"
Even in the worst-case scenario above, my_coin's design still gives victims two baseline protections:

- **(1) The victim can still recover funds from that intermediary (within the latter's liability cap)**
  - If the intermediary refuses to name downstream participants, they must undertake civil liability according to the usage terms;
  - In other words, they cannot both refuse to point to downstream recipients *and* completely evade responsibility;
  - For victims, this means:
    * Even if the chain cannot be fully broken through to the final beneficiary, they can still recover some or all of their losses from that intermediary (depending on the intermediary's "liability cap" and asset situation).
- **(2) The key point is "ex ante incentives": rational people should not want to be the first-hop intermediary in the first place**

  Even acknowledging the limitation above, my_coin's AML design still constructs a very hostile environment for laundering intermediaries *before* they act:

  - Expected payoff from being a first-hop intermediary:
    * Only a very small commission;
    * But once exposed, they must either: -undertake civil liability for the full stolen amount up to their cap; or

        - Admit to being a laundering intermediary and face money-laundering criminal charges;
    - The more levels they participate in, the more likely they are to:
        - Be traced by a real upstream victim with SK;
        - Be named by downstream layers when those are caught;
        - Be tested via sting operations by law enforcement.

Under such incentives:

- **For rational participants, being the "first-layer laundering intermediary" is inherently a negative-expected-value bet**;
- Rational actors should refuse to join such chains *ex ante*, rather than hope that "if something happens, I can just stonewall and be fine".

Therefore, my_coin's aim is not "to guarantee that every existing laundering chain will be fully broken 100% of the time", but rather:

> Through protocol structure + civil liability model + SK-based traceability,
> make potential intermediaries feel *beforehand* that "this business is simply not worth doing",
> thereby **shrinking the laundering market itself** — or at least making it much harder for stable, large-scale laundering networks to form.

4.5.3 There can still be "irrational or misled first-hop participants" Finally, one more realistic caveat:

- No matter how the protocol is designed, it cannot stop some people from:
    - Failing to evaluate risks vs reward properly;
    - Being persuaded by others that "this money is definitely safe" / "you are just doing a harmless pass-through";
    - Or not understanding the usage terms they signed at all;
    - And thus stumbling into being the first or intermediate layer of a laundering chain.

Once such people are dragged in:

- From the protocol's perspective, they do bear the corresponding civil liabilities;
- But in terms of behavior, they may not have done any rational risk–return analysis.

This kind of participation resulting from irrationality / information asymmetry cannot be eliminated 100% by any protocol-level design and is a real limitation of my_coin.

5. Prototype overview and demo list

This chapter describes which scenarios are actually covered by the **single-node prototype** in the current repository, and how each demo corresponds to Chapter 3's protocol design, the specs in `docs/`, and the ZK circuits in `zkcrypto/`.

You can read Chapter 3 while cross-referencing:

- Scripts under `demo/` (end-to-end scenarios);
- Protocol specs under `docs/`;
- Rust circuits and prove/verify implementations under `zkcrypto/`;
- Python-side ZK tests under `tests/rust_api_tests/`.

5.1 Positioning of the prototype and directory navigation

The current repository implements a **single-node prototype** of the my_coin protocol. It is essentially a "local experimental environment," with the goal:

> To close the loop, at an engineering level, on the entire mechanism of BlueAccount / anonymous pool / anonymous payments / SK-based tracing — not just keep it in the whitepaper, but actually run demos and tests.

From the directory structure, the parts most relevant to this chapter are:

- `demo/`
  End-to-end scenario scripts, for example:
  - `demo1_good.py`, `demo1_bad.py`: blue address application flow;
  - `demo2_good.py`: ordinary transparent transfers;
  - `demo3_good.py`, `demo3_bad.py`: Acct → Anon;
  - `demo4_good.py`, `demo4_bad.py`: AnonPay;
  - `demo5.py`: tracing based on SK.
- `docs/`
  Protocol spec documents, split into subdirectories by function, for example:
  - `docs/blue_apply/`: blue address application (corresponding to Sections 3.2 and 3.3);
  - `docs/acct_to_anon_commit/`: non-anonymous account → anonymous commitment (corresponding to Section 3.5.3);
  - `docs/anon_pay/`: anonymous payments and change commitments (corresponding to Sections 3.6 and 3.7);
  - Other auxiliary explanatory documents.
- `zkcrypto/`
  Rust ZK engine and circuit implementations, including:
  - `zkcrypto/src/zkproof_for_ii_blue_apply_gen.rs` / `_verifier.rs`: Blue Apply II;

  - `zkcrypto/src/zkproof_for_acct_to_anon_tx_gen.rs/_verifier.rs`: Acct → Anon;
  - `zkcrypto/src/zkproof_for_anon_pay_gen.rs / _verify.rs`: AnonPay;
  - As well as base modules for Poseidon, elliptic curves, Merkle trees, etc.
- `tests/rust_api_tests/`
  Python-side ZK tests that call `zkcrypto` via FFI, used to verify:
  - Blue Apply circuits;
  - Acct → Anon circuits;
  - AnonPay circuits;
  - Base APIs for ECC / Poseidon, etc.

The following subsections of this chapter will revolve around the scripts under `demo/`, explaining which scenario each one validates, and how they correspond to Chapter 3, `docs/`, and `zkcrypto/`.

5.2 Demo 1: Blue address application (Blue Apply)

`demo1_good.py` / `demo1_bad.py`

5.2.1 Scenario description Demo 1 shows how the **BlueAccount application and derivation flow** in Sections 3.2 and 3.3 is implemented in the prototype:

> How a KYC'ed user obtains a first blue address on-chain (`color = 1`),
> and how to derive subsequent blue addresses from the same `master_seed`.

From the protocol's perspective, this corresponds to two layers of logic:

- **Blue Apply I (Clerk endorsement + cooling-off period)**
  - The Clerk, representing the offline KYC institution, cryptographically endorses the fact that "a certain real-world identity is applying for a BlueAccount";
  - After the application is submitted on-chain it enters a cooling-off period, during which the Clerk can revoke applications that were filed via identity theft;
  - Corresponding spec documents (examples):
    * `docs/blue_apply/how_to_apply_for_blue.md`
    * `docs/blue_apply/lst_of_txs_in_blocks_blue_apply.md`
- **Blue Apply II (password → master_seed → derived SK / address ZK proof)**
  - Locally derive `master_seed` from a password, then derive `master_seed_hash` (see Section 3.2.1 for the exact hash chain);
  - For a given address, construct a ZK proof that:

    * This address is derived from `master_seed` and the public `token` according to the protocol rules;

    * `master_seed` matches the `master_seed_hash` in the account state;

  - Once verification passes:
    * Mark this address as a BlueAccount (`color = 1`);

    * Record `(ID, master_seed_hash, token)`;

  - Corresponding specs and implementation:
    * Logical and circuit description: `docs/blue_apply/zkproof_ii_blue_apply.md`

    * Rust circuits:
      · `zkcrypto/src/zkproof_for_ii_blue_apply_gen.rs`

      · `zkcrypto/src/zkproof_for_ii_blue_apply_verifier.rs`

    * Python ZK tests:
      · `tests/rust_api_tests/zkproof_ii_blue_apply_tests.py`

The concrete parameters in the prototype (e.g., hash chain length, etc.) are slightly simplified in the implementation, but the overall structure is consistent with Sections 3.2 and 3.3.

5.2.2 What the good / bad scripts demonstrate

- `demo1_good.py`
  - Simulates a legitimate user:
    * First, the Clerk submits a Blue Apply I request;

    * Then, locally generates the ZK proof according to Sections 3.2 and 3.3;

    * After completing the cooling-off period, successfully obtains the first blue address, or derives additional blue addresses under the same `master_seed`.
- `demo1_bad.py`
  - Collects various invalid / malicious inputs, for example:
    * A ZK proof that does not match the public `(master_seed_hash, token, address)`;

    * An un-KYC'ed user trying to forge a Blue Apply request;

  - Demonstrates that the validating node will **refuse to upgrade to a BlueAccount** in these scenarios, without corrupting the current on-chain state.

5.3 Demo 2: Ordinary transfer (non-anonymous)

`demo2_good.py`

5.3.1 Scenario description Demo 2 corresponds to Section 3.4, showing:

> How ordinary accounts perform transparent transfers when the anonymous features are not used.

Characteristics:

- Sender address, receiver address, and amount are all in clear on-chain;
- Transfer ordering is constrained by the account's `nonce` (the "`++nonce`" rule in Section 3.4.1);
- Semantically, this is almost identical to the account model of Ethereum-like chains.

This part is mainly used to:

- Verify the most basic account-level logic of "signature verification + `nonce` check + balance updates";
- Provide the funding source for Demo 3 (Acct → Anon):
  - You must have a balance in the non-anonymous layer before you can move funds into the anonymous layer.

5.3.2 Relationship with other modules

- Demo 2 does not invoke any ZK circuits;
- But it provides the non-anonymous balances that serve as entry points to the anonymous layer in Section 3.5.

5.4 Demo 3: Transferring from a public account into the anonymous store (Acct → Anon)

`demo3_good.py` / `demo3_bad.py`

5.4.1 Scenario description Demo 3 corresponds to the ACCT (Account → Anonymous Commitment) in Section 3.5.3, and demonstrates:

> How an already-blue account
> moves part of its public balance
> into an anonymous commitment AC = anon_commit = anonymous commitment,
> and inserts it into `MerkleTreeCommit`.

This step follows the core principle from Section 3.5.2:

No ownership transfer is allowed when generating a new AC.

At the circuit level, the Acct → Anon ZK proof used in Demo 3 must ensure (conceptually matching Section 3.5.3):

- The four fields `(balance, sk, nonce, src)` of the AC satisfy AC = Poseidon(…);
- `sk` is both the control key of the note and the control key of the originating account address (`sk → PK → ADDR` statement);
- The amount is deducted from the public balance and matches the AC's `balance`;
- Ownership cannot be handed to another key at this "generate AC" step.

Corresponding specs and implementation pointers:

- Protocol documents:
  - `docs/acct_to_anon_commit/how_to_apply_for_acct2anon.md`
  - `docs/acct_to_anon_commit/zkproof_acct_to_anon_specs.md`
- Rust circuits:
  - `zkcrypto/src/zkproof_for_acct_to_anon_tx_gen.rs`
  - `zkcrypto/src/zkproof_for_acct_to_anon_tx_verifier.rs`
  - `zkcrypto/src/zkproof_for_acct_to_anon_tx_evil_gen.rs` (used for comparison / testing malicious paths)
- ZK tests:
  - `tests/rust_api_tests/zkproof_acct_to_anon_tx_tests.py`

5.4.2 What the good / bad scripts demonstrate

- `demo3_good.py`
  - Demonstrates a standard Acct → Anon flow:
    - A BlueAccount has sufficient balance in the transparent layer;
    - Locally constructs a ZK proof that matches the semantics in Section 3.5.3;
    - After the node verifies it:
      - A new AC is inserted into the anonymous pool;
      - The MerkleTreeCommit root is updated;
      - The corresponding amount is deducted from the account's transparent balance.
- `demo3_bad.py`

- Constructs various invalid or malicious Acct → Anon requests, such as:
  - Using an `sk` in the proof that does not correspond to the originating account address;
  - An AC structure that is inconsistent with `docs/acct_to_anon_commit/zkproof_acc`
- Shows how the node rejects these requests.

5.5 Demo 4: Anonymous payments (AnonPay)

`demo4_good.py` / `demo4_bad.py`

5.5.1 Scenario description Demo 4 corresponds to Section 3.6 (and the related design for nullifiers / CommitChange), and demonstrates:

> How to spend an anonymous commitment from the pool,
> pay a **public amount** to some public address,
> and at the same time generate a new anonymous change AC (CommitChange) in the pool.

From the protocol's perspective, it is exactly the pattern described in Section 3.6:

- Spend an old AC corresponding to an old note;
- Publicly reveal a `nullifier` to mark that note as spent;
- Transfer part of the amount in transparent form to a public address;
- Use the remaining amount as change to generate a new `CommitChange`:
  - `CommitChange = Poseidon(balance_change, sk, 0, nullifier);`
  - And the ZK circuit enforces that the new and old AC share the same `sk`, so it is impossible to change owners secretly within the anonymous layer.

At the circuit level, the AnonPay ZK proof used in Demo 4 must satisfy the four core statements in Section 3.6.5:

1. The old note's AC is indeed a real leaf in the current `MerkleTreeCommit`;
2. The public `nullifier` is indeed computed as `nullifier = Poseidon(sk, nonce_old, src_old);`
3. The new `CommitChange` is formed from `(balance_change, sk, 0, nullifier)`, with `sk` identical to the old AC;

4. Amounts are strictly conserved: `balance_old = pay_balance + balance_change`.

Corresponding specs and implementation pointers:

- Protocol documents:
  - `docs/anon_pay/how_to_apply_for_anon_pay.md`
  - `docs/anon_pay/lst_of_txs_in_blocks_anon_pay.md`
  - `docs/anon_pay/zkproof_anon_pay.md`
- Rust circuits:
  - `zkcrypto/src/zkproof_for_anon_pay_gen.rs`
  - `zkcrypto/src/zkproof_for_anon_pay_verify.rs`
- ZK tests:
  - `tests/rust_api_tests/zkproof_anon_pay_tests.py`

5.5.2 What the good / bad scripts demonstrate

- `demo4_good.py`
  - Constructs multiple valid anonymous payments:
    - Uses ACs that are already in the anonymous pool;
    - Generates ZK proofs according to the rules in Section 3.6;
    - Once the node verifies them:
      - A new `CommitChange` is inserted into the anonymous pool;
      - The corresponding `nullifier` is recorded as spent;
      - The public address receives an increase in transparent balance.
- `demo4_bad.py`
  - Collects several attempts at "invalid anonymous payments", for example:
    - Re-using an already-seen `nullifier` (double-spend attempt);
    - Violating amount conservation, or forging the structure of `CommitChange`;
  - Used to verify how the node defends itself against these bad inputs and ensures that the anonymous payment logic cannot be bypassed.

5.6 Demo 5: SK-based tracing (Trace by SK)

`demo5.py`

5.6.1 Scenario description Demo 5 essentially replays the design philosophy of Sections 3.5–3.7 in code (especially the structure of AC, nullifier, and `src`), and demonstrates:

> As long as you hold the private key of some BlueAccount,
> plus the on-chain records of anonymous commitments / nullifiers,
> you can reconstruct the entire anonymous payment chain "originating from that account".

The script is roughly two-step:

1. **Prepare some anonymous history**
   - Internally, it automatically executes the flows of `demo4_good.py`:
     - First, move funds from a BlueAccount into the anonymous pool (Acct → Anon);
     - Then perform several Anonymous Pay operations;
   - This yields a bunch of ACs and nullifiers, which are written into the current chain state.
2. **Given SK, enumerate all anonymous objects related to it**
   - Based on the definitions of note / AC / nullifier in Sections 3.5 and 3.6:
     - Once you know `sk`, you can locally enumerate all possible `(balance, sk, nonce, src)` combinations,
       and thus all possible ACs and nullifiers (within a bounded parameter space);
   - By comparing them to on-chain records:
     - You can find which ACs / nullifiers actually exist in the current `MerkleTreeCommit` or transaction history;
     - From there, reconstruct the "anonymous outflows under this SK".

At the demo level, `demo5.py` only implements a very simplified tracing logic, but it is already enough to illustrate the core design philosophy in Chapter 3:

> The structure of AC and nullifier is determined by `(sk, nonce, src)` and similar fields;
> The protocol forbids silently changing the control key when generating a new AC;
> Therefore, as long as the user still holds the SK, they can follow this SK and trace the flow of funds step by step.

5.6.2 Relationship with the AML mechanism From an AML perspective, Demo 5 is the engineering demonstration of Chapter 4's "**anonymous to outsiders, but traceable to insiders**" idea:

- For third-party observers, an anonymous payment only reveals the recipient address and amount; the payer's identity is hidden;

- But for the person who holds the SK of a BlueAccount (including victims, and intermediaries who are required to explain the origin of funds):
  - If they choose to, they have the ability to use their SK to compute the anonymous flows *after* "their hop";
  - Thus, in combination with the usage terms, either identify the next layer of addresses or bear the corresponding civil liability themselves.

At the prototype level, `demo5.py` shows that this is practically implementable, and provides a runnable, verifiable "technical backbone" for the attack–defense analysis in Chapter 4.

---

6. How to run the demos

- The project targets a Linux environment, ideally Ubuntu 24.04.
- My own development environment was: Windows 11, WSL 2, Ubuntu 24.04.
- You may need to install the following dependencies first:

```
sudo apt update && sudo apt upgrade
sudo apt install -y python3.12 python3.12-venv python3.12-dev python3-
```

- From the project root, create and activate a virtual environment:

```
python3 -m venv .venv
source .venv/bin/activate
```

- You need a recent Rust toolchain and `maturin` if you want to rebuild the ZK module yourself

```
cd zkcrypto
maturin develop -- release
```

- Then you can directly run the demos, for example:

```
python demo/demo1_good.py
```