Privacy-Preserving Tokenized Green Credits with Layer-2 Minting and Layer-1 Anchoring

A deployable engineering prototype for micro-generation credit issuance

ECE 910 - MEng Capstone Project

Department of Electrical and Computer Engineering

University of Alberta

Author: Youssef Ibrahim
Supervisor: Professor Majid Khabbazian
Program: Master of Engineering (MEng)
Date: December 2025

Abstract

Renewable-energy credit (REC) systems provide a mechanism for recognizing and trading verified clean generation. In practice, micro-generators (e.g., household solar) frequently encounter high friction: reporting and verification are performed via intermediaries; registry interfaces can be complex; and privacy constraints make it difficult to demonstrate eligibility without disclosing identifying information such as meter identifiers, addresses, or fine-grained production traces. Public blockchains offer transparency and automated settlement, but naïve "mint-on-chain" approaches risk (i) leaking personally identifying information (PII) and (ii) double issuance through replay or cross-domain duplication.

This report presents an engineering capstone prototype that bridges private verification to public token issuance while enforcing duplication prevention and supporting deployment to optimistic Layer-2 (L2) systems. The design separates responsibilities across three domains: (1) a private verification domain that records meter certification and oracle verdicts using lightweight smart contracts; (2) a public minting domain on L2 that validates an authorization proof, mints ERC-20 green credit tokens, and computes a canonical duplication key; and (3) a public audit domain on Layer-1 (L1) that anchors each successful L2 mint through authenticated L2→L1 messaging. A core outcome is a deployable contract suite that supports both (a) a generic optimistic-rollup messenger interface and (b) an Arbitrum-specific outbox authentication model. The prototype includes local "mock" cross-chain components to demonstrate and test the end-to-end control flow without requiring funded wallets, and it provides scripts that can be used for real testnet deployments when funding is available.

Evaluation includes unit and integration tests that cover private registry operations, duplicate prevention, and L2→L1 anchoring authentication, together with a reproducible gas report that isolates L2 minting cost from the distinct L1 execution cost of anchoring. Results show that the L2 mint path remains predictable and bounded, duplicate issuance is prevented at the smart-contract level, and the L1 anchor reliably rejects unauthenticated calls, establishing an auditable issuance record without revealing private meter data.

Keywords

Renewable energy credits; tokenization; optimistic rollups; Arbitrum; cross-chain messaging; duplicate prevention; smart contracts; privacy; verifiable authorization.

Contents

List of Figures

List of Tables

List of Acronyms

Acronym	**Meaning**
AMI	Advanced Metering Infrastructure
EOA	Externally Owned Account (Ethereum account controlled by a private key)
EVM	Ethereum Virtual Machine
GA	Governance Authority (meter certification authority in this prototype)
L1	Layer 1 (base chain; e.g., Ethereum)
L2	Layer 2 (rollup chain; e.g., Arbitrum, Optimism)
RA	Registry Administrator (private verification operator in this prototype)
REC	Renewable Energy Credit
RPC	Remote Procedure Call
VC	Verifiable Credential

1.0 Introduction

1.1 Motivation

Renewable generation is increasingly distributed. Rooftop solar, small wind installations, and community micro-generation reduce emissions and can supply local load. However, incentives and market mechanisms for recognizing this generation often assume utility-scale participants or require specialized reporting entities. Renewable Energy Credits (RECs) and similar instruments provide a standardized accounting unit (often 1 MWh) and can be traded to meet compliance obligations or voluntary sustainability targets. In many jurisdictions, the pathway from "energy produced" to "credit issued" involves a sequence of processes: meter certification, reading collection, validation (sometimes with utility participation), registry submission, issuance, and transfer.

For micro-generators, this pipeline commonly introduces three problems:

1. **Access and operational friction.** Small producers may lack the tooling or volume thresholds required by existing registries. They therefore rely on aggregators or reporting entities to interact with program administrators, and they receive compensation indirectly.

2. **Trust and auditability.** When credits are issued based on off-chain processes, it can be difficult for third parties to audit issuance without trusting a single administrator. This becomes more complex when credits are bridged into tokenized systems, potentially leading to double counting across platforms.

3. **Privacy.** Meter identifiers, site location, and high-resolution production traces can reveal occupant behavior and physical address. A public ledger provides transparency, but it also makes correlated disclosure persistent and widely accessible.

Smart contracts can automate issuance and accounting, but moving verification directly onto a public chain is often infeasible or undesirable. Even if a producer is willing to reveal data, the verification process may require confidential utility data, proprietary models, or regulated information. Conversely, purely private issuance reduces transparency. The central design question is therefore: *How can a system provide public auditability of credit issuance while keeping private meter information off-chain and preventing duplication?*

1.2 Problem Statement

The engineering problem addressed in this project is the construction of a deployable, testable pipeline that:

- Allows a micro-generator (or its delegate) to obtain a *private verification result* for a claimed amount of renewable generation;
- Allows minting of a public, transferable token representing the verified generation *without publishing PII or raw meter data*; and
- Enforces *duplicate prevention* such that the same issuance claim (defined by an unambiguous canonical tuple) cannot be minted more than once, even in the presence of replay attempts.

In addition, this project targets modern Ethereum scaling environments. Many realistic tokenization deployments will occur on optimistic L2 systems for cost and throughput reasons. If issuance happens on L2, then a practical audit mechanism is to anchor the L2 mint on L1, where the anchor can be observed and indexed by anyone. This motivates an L2-first design with authenticated L2→L1 anchoring. In this capstone, Arbitrum is selected as a concrete representative of optimistic L2 systems, while maintaining a generic messenger-based design compatible with OP Stack style messaging.

1.3 Design Goals

This capstone is a technical (engineering) project. The design goals are therefore framed as deployable requirements:

1. **Deployability:** The system must compile and execute in a local development environment and must include scripts suitable for testnet deployment on Ethereum Sepolia and Arbitrum Sepolia when funding is available.
2. **Auditability:** Every successful issuance must leave a verifiable event trail, including an L1 anchor record for L2 mints.
3. **Privacy by construction:** The public mint flow must not require publishing meter identifiers, addresses, or raw readings.
4. **Duplication prevention:** The minting contract must reject duplicates based on a canonical *Disclosure Tuple* (DT) defined by policy.
5. **Security hygiene:** Contracts must use standard safety patterns (checks-effects-interactions, strict access control, explicit custom errors, bounded execution).
6. **Extensibility:** The prototype must be structured such that stronger cryptographic proof systems (e.g., selective disclosure signatures or zk proofs) can replace the demo authorization mechanism without rewriting the rest of the pipeline.

1.4 Contributions

The contributions of this work are:

- A complete contract suite for private verification indexing (`MetReg`, `DataVer`), public L2 minting (`GTokenL2` and `GTokenL2Arb`), and L1 anchoring (`GTokenAnchor` and `GTokenAnchorArb`).
- A duplication-prevention mechanism based on a canonical disclosure tuple and a deterministic hash `dtHash`, enforced on the minting contract prior to token minting.
- Two authenticated anchoring patterns:
    1. A generic messenger-authenticated anchor that checks the originating L2 sender via `xDomainMessageSender()` (OP Stack style).
    2. An Arbitrum-specific anchor that checks `msg.sender == bridge.activeOutbox()` and the originating L2 sender via `l2ToL1Sender()` [6, 7].
- A local "mock" environment that tests the cross-chain logic deterministically without external networks, together with scripts to run demos and generate a reproducible gas report.
- An evaluation that clarifies the interpretation of gas measurements when L2→L1 messages are modeled locally: the L2 mint transaction and the subsequent L1 execution are distinct transactions on real rollups, so combined local cost numbers should not be compared directly to L2 fees.

1.5 Scope and Non-Scope

In scope:

- Smart contract implementations and tests for the pipeline described above.
- A demonstration-grade off-chain authorization mechanism implemented as an issuer ECDSA signature verified on-chain (a stand-in for stronger selective-disclosure proofs).
- L2→L1 anchoring logic that can be executed locally via mocks and can be deployed to Arbitrum testnets when configured.
- Gas measurement scripts and reporting artifacts.

Out of scope:

- Integration with real utility AMI systems, production-grade meter reading ingestion, or regulated registry onboarding.
- A full cryptographic selective disclosure credential implementation on-chain (e.g., BBS+ proofs)-discussed as future work and as an interchangeable interface.
- Economic design of token pricing, market making, or compliance certification of tokens as official RECs.

1.6 Report Roadmap

Section 2 reviews relevant background: REC issuance constraints, privacy considerations, and L2→L1 messaging models, with a focus on Arbitrum. Section 3 formalizes requirements and stakeholders. Sections 4-7 specify the architecture, protocol, and smart-contract design. Sections 8-11 provide security analysis and evaluation results including gas measurements. Sections 12-15 discuss deployment, governance, project management, and conclusions. Appendices provide interface excerpts, command runbooks, and supporting artifacts.

2.0 Background and Related Work

2.1 Renewable Energy Credits and Verification Pipelines

RECs represent a standardized accounting claim that a certain quantity of electricity was generated from eligible renewable sources. The key engineering property of a REC system is *verifiability*: a credit should only be issued if generation is measured, certified, and not already claimed elsewhere. Traditional registry designs generally rely on certified meters and reporting entities. From an implementation standpoint, the process includes:

1. Certifying a meter and associating it with a generation facility (or micro-generation site);
2. Collecting interval data or cumulative reads;
3. Validating the data against expected capacity constraints and detecting tampering;
4. Submitting verified generation to an administrator; and
5. Issuing an instrument that can be transferred and retired.

These processes are frequently designed for organizational participants rather than individuals. For micro-generation, the administrative overhead can be disproportionate to the energy volume. A practical system therefore benefits from automation and minimized disclosure: the system should prove eligibility without requiring that private site details become part of a public record.

2.1.1 Micro-generation Programs and Operational Constraints

Micro-generation refers to small-scale renewable generation installed at or near a point of consumption and interconnected to a distribution network. While the institutional details differ by jurisdiction, micro-generation programs tend to share a set of operational constraints that strongly influence any attempt to tokenize generation claims.

Interconnection is procedural and regulated. A micro-generator is not simply "plug-and-play" on a public grid. Interconnection generally requires an application, technical review, inspection, and approval by a utility or regulator. These steps exist to protect grid safety and reliability (e.g., ensuring anti-islanding behavior, proper disconnects, and compliance with technical standards). Regardless of how advanced a token system is, it cannot bypass these safety-critical processes.

Measurement is mediated by certified metering. Claims about exported energy depend on metering that the program administrator accepts. Even if a participant can measure production locally, incentive programs and credit accounting typically require a trusted measurement process with defined accuracy, calibration, and retention rules. This creates a natural separation between *measurement and verification* (a regulated activity) and *accounting and transfer* (which can benefit from automation).

Reporting introduces roles, timelines, and aggregation. The pathway from "energy production" to "credit issuance" involves multiple actors and processes: meters, data collection, validation checks, reporting entities, and administrators. For micro-generators, the overhead of this pipeline can exceed the economic value of small time-slice claims. This leads to aggregation over longer periods or delegation to intermediaries.

As a concrete example, the Alberta Utilities Commission publishes micro-generator application guidance that describes the micro-generation context and associated application processes [4]. The engineering lesson for this capstone is that a public token contract should not attempt to replicate the entire institutional verification pipeline. Instead, it should integrate with it in a way that preserves auditability while minimizing disclosure.

For this reason, the prototype introduces a *private verification domain.* In the MVP, the private domain is represented by the meter registry (`MetReg`) and verification index (`DataVer`) that store only hashes and status flags. In a deployment-grade design, the private domain would be the natural place to integrate utility data feeds, AMI exports, regulator-accepted verification processes, or multi-oracle consensus. The public chain would then enforce only a small set of properties that it can enforce well: authorization of issuance, prevention of duplicates, and creation of an auditable record that issuance occurred.

Granularity and accounting units. Traditional REC systems often use large accounting units (commonly 1 MWh). A household photovoltaic system may produce tens of kWh per day, and producing a credit in 1 MWh units can require aggregation over long periods. Tokenization can reduce friction by representing smaller denominations (e.g., kWh) while still preserving the ability to aggregate to larger units for reporting. This capstone mints in integer kWh units on L2. Aggregation and rounding are assumed to occur in the private domain during verification and issuance, where the administrator can apply conservative rounding policies to avoid accidental over-issuance.

2.1.2 Registry Accounting and Audit Expectations

A registry performs two tightly coupled functions: (i) it defines *issuance legitimacy* (what evidence qualifies and how it is validated), and (ii) it defines *accounting state* (who owns a credit and whether it has been transferred or retired). Tokenization addresses transfer and programmability very well, but it does not automatically solve issuance legitimacy. Therefore, a tokenized system must still provide answers to audit questions such as:

- Which verified evidence supports each issuance?
- Can an observer detect if the same claim was issued twice *within the system*?
- Can an observer detect if the same claim was issued elsewhere (cross-domain double counting)?

The Western Renewable Energy Generation Information System (WREGIS) is an example of a regional tracking system used for renewable certificate accounting in the western United States and is referenced in state compliance documentation [5]. While this report does not attempt to reproduce WREGIS governance or operational rules, the existence of such registries highlights an important engineering principle: **credit systems are accounting systems first**. A transfer mechanism without a robust, auditable issuance story will not satisfy regulators or auditors.

This capstone uses the L1 anchor to strengthen the issuance story without publishing private data. For each L2 mint the system produces:

1. A public L2 event (`Minted`) keyed by `dtHash` and containing the minimal disclosed tuple; and
2. A corresponding L1 anchor record (`MintAnchored`) written only via authenticated cross-chain execution.

The L1 anchor is therefore an *audit artifact*. It does not claim to make the token an official REC; rather, it provides an immutable, globally visible record that issuance occurred under the token contract's rules on L2. This is valuable in two ways. First, it provides a simple integration point for auditors: they can watch a single L1 contract for issuance records instead of indexing multiple L2 networks directly. Second, it establishes a foundation for future cross-domain duplication prevention: in principle, an ecosystem could treat the L1 anchor as a globally visible "already issued" set.

Finally, the registry perspective motivates **duplication prevention as a first-class requirement**. Even if an issuer is trusted, mistakes happen; and even if a holder is honest, transaction retries and client bugs occur. A deterministic, on-chain duplication guard eliminates an entire class of operational failures. This prototype's duplication guard is intentionally independent of cryptographic proof choices: regardless of whether authorization is provided by ECDSA signatures, verifiable credential proofs, or zk proofs, the same canonical `dtHash` rule prevents the same claim from being minted twice.

2.2 Tokenization and Double Counting Risks

Tokenizing credits on a blockchain can improve transparency and reduce settlement friction. Many tokenization efforts demonstrate that on-chain representations can provide real-time auditability and programmable transfer logic. However, tokenization creates a new class of risks: *double counting* and *cross-domain duplication*. If a credit is issued in a traditional registry and also tokenized elsewhere without a strong linkage, it may be counted twice. Even within a single token system, a user may attempt to mint multiple times based on the same underlying claim, either by replaying proofs or by exploiting ambiguous encoding.

This motivates a critical design principle: **issuance must be keyed by a canonical representation of a claim**, and the system must maintain an immutable "already used" set for this key. In this project, the key is the *Disclosure Tuple* (DT), and the on-chain duplication guard is a mapping indexed by `dtHash`.

2.3 Privacy in Energy Data

Energy consumption and production traces can reveal occupancy patterns, appliance usage, and daily schedules. Publishing meter identifiers or fine-grained readings on a public chain is therefore undesirable. A privacy-preserving design should:

- Keep raw readings off-chain;
- Avoid publishing stable identifiers (meter IDs, addresses);
- Use commitments/hashes to create auditability without disclosure; and
- Allow public validation of authorization without revealing the hidden fields.

This project achieves privacy by separating the private verification domain (where meter and reading evidence can be evaluated) from the public minting domain (which only accepts a minimal disclosure tuple and a proof/authorization of eligibility).

2.4 Verifiable Credentials and Selective Disclosure (Context)

A widely cited approach to privacy-preserving authorization is to issue a verifiable credential (VC) that attests to a statement such as: "The holder generated q kWh of eligible renewable energy in epoch e." The W3C Verifiable Credentials Data Model provides a standard format for such claims and their proofs [1]. Selective disclosure signatures, such as the CFRG BBS signatures draft [2], can allow a holder to reveal only certain fields of the credential while proving that hidden fields were also signed by a trusted issuer.

This capstone does not implement BBS proofs on-chain, primarily for engineering complexity and build-time constraints. Instead, it defines a *proof verifier interface* and provides a deployable placeholder: an ECDSA-signed authorization by a demo issuer key. The architecture is designed so that an on-chain selective-disclosure verifier can replace the placeholder without changing the minting or anchoring components.

2.5 Ethereum Scaling and Why L2 Matters

Public L1 execution offers strong security and wide observability but can be expensive for frequent mint operations. Optimistic L2 rollups execute transactions off-chain and post compressed data to L1, reducing user costs and increasing throughput. In many real deployments, issuance will occur on L2 for practicality, while L1 remains the source of truth for dispute resolution and finality.

If mints happen on L2, an L1 anchor is useful for global observability: it provides an L1-accessible record of issuance, enabling cross-system auditing and (in future work) cross-chain duplication prevention. The anchor must authenticate that a mint occurred on L2 and that the message originated from the legitimate L2 token contract.

2.6 L2→L1 Messaging Models

Two dominant patterns exist for authenticated L2→L1 messages in optimistic rollups:

Messenger-based authentication (OP Stack style). A common abstraction is a cross-domain messenger contract. The L1-side contract requires that calls come from the messenger, and it queries the messenger for the *originating L2 sender* via a function such as `xDomainMessageSender()`. The L1 contract then compares this sender to an allowlisted L2 contract address. This pattern is simple and general and is used across multiple OP Stack-derived systems.

Arbitrum outbox authentication. Arbitrum executes L2→L1 messages on L1 via an *Outbox*. The L1-side contract can authenticate the call by requiring:

1. `msg.sender` equals the active outbox returned by the Arbitrum Bridge, and
2. `IOutbox(msg.sender).l2ToL1Sender()` equals the allowlisted L2 contract address.

This project implements this pattern in `GTokenAnchorArb`. The use of `activeOutbox` and `l2ToL1Sender` follows Arbitrum's documented interface and contract address registry [6, 7].

2.7 Related Work Summary

In summary, prior systems and standards motivate three key requirements: privacy-preserving authorization, duplication prevention via canonical keys, and scalable execution via L2 with secure anchoring to L1. This capstone focuses on engineering a coherent, testable contract suite that embodies these principles and can be deployed with realistic tooling.

3.0 Requirements and Stakeholders

3.1 Stakeholders

Micro-generator / holder. The end participant who wants to receive a tokenized credit for verified renewable generation. The holder is assumed to control an EOA on L2 and may use a wallet or script to interact with contracts.

Governance Authority (GA). The actor that certifies meters (or meter certificates) and can change meter status. In production, this could correspond to a program administrator, utility, or delegated certification authority.

Registry Administrator (RA). The operator of the private verification domain. The RA manages the verification index, coordinates oracle verdict posting, and (in production) would operate credential issuance infrastructure.

Oracle. An off-chain service that checks claimed readings against a dataset or policy rules and posts signed verdicts to the verification index contract. For this capstone, a synthetic dataset is used for repeatability.

Public observers and auditors. Any third party that monitors issuance events and anchors. Observers are assumed to be adversarial from a privacy standpoint, meaning they will attempt to correlate public events to infer sensitive information.

3.2 Functional Requirements

Table 2 summarizes the functional requirements (FR) of the prototype.

Table 2: Functional requirements (FR).

ID	Requirement
FR-1	Meter registry must record meter certification hashes and allow GA to activate/revoke meters.
FR-2	Verification index must record reading commitments (hashes) and oracle verdicts.
FR-3	Verification index must allow RA to anchor a credential hash (or issuance hash) to a verified reading commitment.
FR-4	L2 minting must validate an authorization proof via a verifier interface and must mint ERC-20 tokens accordingly.
FR-5	L2 minting must compute a canonical duplication key `dtHash` and must reject duplicate mints.
FR-6	For each successful L2 mint, the system must emit an authenticated L2→L1 message to record the mint on an L1 anchor.
FR-7	L1 anchoring must reject calls that do not originate from the correct cross-chain mechanism and the allowlisted L2 sender.
FR-8	Contracts must emit events for state transitions to support indexing and audit.
FR-9	The repository must include scripts for demos, gas reporting, and (optionally) testnet deployment.

3.3 Non-Functional Requirements

Non-functional requirements (NFR) guide engineering choices:

- **NFR-1 (Privacy):** No PII or raw meter readings are written to public chains. Public interactions disclose only a minimal tuple and derived hashes.
- **NFR-2 (Security):** Minting is gated by an authorization verifier; duplication is prevented; access control is explicit; and anchoring is authenticated against spoofing.
- **NFR-3 (Testability):** The system provides deterministic unit tests and local mocks for cross-chain messaging.
- **NFR-4 (Reproducibility):** Gas measurements are reproducible and include notes about what is and is not representative of real L2 fees.

- **NFR-5 (Maintainability):** Contracts are modular, use standard libraries (OpenZeppelin) where appropriate, and keep interfaces minimal.
- **NFR-6 (Deployability):** The system supports local Hardhat execution, and it can be configured for testnet deployments via environment variables.

3.4 Assumptions and Constraints

The following assumptions are made for this capstone prototype:

1. The issuer/verifier key used in the authorization mechanism is trusted for the duration of the evaluation.
2. The oracle posting verdicts in the private domain is honest in the evaluation scenario.
3. The canonical disclosure tuple definition is policy-chosen. In this prototype, DT includes epoch index, energy type code, quantity (kWh), and a policy nonce.
4. The L1 anchor is intended as an audit record; it does not itself mint tokens or enforce economic constraints.

The primary constraint is engineering time: implementing and auditing a full selective-disclosure credential scheme on-chain is beyond the capstone scope. The chosen architecture therefore treats the proof verifier as replaceable.

4.0 System Architecture and Trust Model

4.1 Design Overview

The prototype is organized into two primary execution domains plus an anchoring interface between them:

1. **Private verification domain.** This domain exists to record meter certification, reading commitments, and oracle verdicts without publishing raw readings. It is implemented as two lightweight contracts:
   - `MetReg`: a meter registry managed by a Governance Authority (GA) role. It records a certificate hash and an `active` flag for each meter.
   - `DataVer`: a verification index managed by a Registry Administrator (RA) role and an `ORACLE_ROLE`. It stores reading commitments, oracle verdicts, and anchors a credential hash to a verified reading hash.
2. **Public minting domain (L2).** This domain is where transferable green credit tokens are minted. The core contract is `GTokenL2`, an ERC-20 token that mints only after verifying an authorization proof and only if the canonical duplication key `dtHash` has not been used before. A second L2 contract, `GTokenL2Arb`, implements the same minting interface but uses the Arbitrum precompile (`ArbSys`) to dispatch an L2→L1 message.
3. **Public audit domain (L1 anchoring).** For every successful mint on L2, the system records an auditable anchor on L1 via an authenticated cross-chain call. Two anchor contracts are provided:
   - `GTokenAnchor`: messenger-authenticated anchor using `xDomainMessageSender()`.
   - `GTokenAnchorArb`: Arbitrum outbox-authenticated anchor using `bridge.activeOutbox()` and `outbox.l2ToL1Sender()` [6].

This separation is intentional. It isolates sensitive data handling and policy-specific verification into a private domain, while leaving the public chain to enforce only the minimal properties needed for transferable tokens: authorization, duplication prevention, and auditable anchoring.

4.2 Component Diagram

Figure 1 summarizes the system at a high level and highlights the trust boundary between the private verification domain and the public minting/audit domains. The numbered arrows in the diagram correspond to the "happy path" described in Section 4.4 and the step-by-step protocol in Section 6.

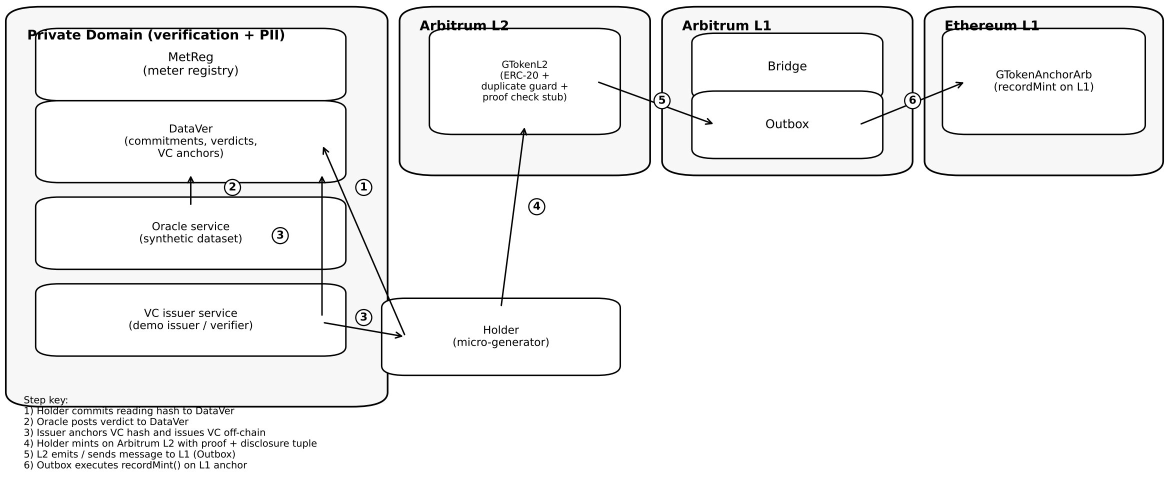
Private Domain (verification + PII)
MetReg (meter registry)
DataVer (commitments, verdicts, VC anchors)
Oracle service (synthetic dataset)
VC issuer service (demo issuer / verifier)
Holder (micro-generator)
Arbitrum L2
GTokenL2 (ERC-20 + duplicate guard + proof check stub)
Arbitrum L1
Bridge
Outbox
Ethereum L1
GTokenAnchorArb (recordMint on L1)
1
2
3
3
4
5
6
Step key:
1) Holder commits reading hash to DataVer
2) Oracle posts verdict to DataVer
3) Issuer anchors VC hash and issues VC off-chain
4) Holder mints on Arbitrum L2 with proof + disclosure tuple
5) L2 emits / sends message to L1 (Outbox)
6) Outbox executes recordMint() on L1 anchor

Figure 1: System architecture: private verification domain (MetReg, DataVer, Oracle), public minting on L2 (GTokenL2 / GTokenL2Arb + Verifier), and auditable anchoring on L1 (GTokenAnchor / GTokenAnchorArb).

4.3 Roles, Permissions, and Trust Boundaries

Roles. The system uses explicit role-based access control for mutation of private-domain state and anchor configuration:

- `GA_ROLE`: can register meters and update meter active status in `MetReg`.
- `RA_ROLE`: can query meters and (in `DataVer`) can anchor credential hashes after oracle verification.
- `ORACLE_ROLE`: can post verification verdicts for reading commitments in `DataVer`.
- `DEFAULT_ADMIN_ROLE` (anchors): can set or rotate the allowlisted L2 sender and cross-chain configuration on anchor contracts.

Trust boundaries. The system assumes that raw meter readings and identifying data exist off-chain. On-chain, only hashes are stored in the private domain, and only a minimal tuple plus hashes are stored or emitted in the public domain. The public observer is treated as adversarial: all public data is assumed visible.

Threat boundaries. Three boundaries are relevant for security analysis:

1. **Between holder and L2 minting contract.** The holder may be malicious and attempt replay or unauthorized minting.
2. **Between L2 and L1.** An attacker may attempt to call the L1 anchor directly, or spoof the cross-chain sender identity.
3. **Within the private domain.** A compromised oracle or misconfigured roles may produce incorrect verification verdicts.

4.4 Data Flow: Happy Path

The intended issuance path is:

1. **Meter registration (private).** GA registers a meter hash and certificate bytes with `MetReg`, setting its active status.
2. **Reading commitment (private).** A holder (or an RA acting on behalf) posts a commitment $\mathsf{H}(\mathsf{reading})$ to `DataVer`. The commitment is a hash computed from a canonical encoding of reading attributes (e.g., meter hash, epoch, type, and quantity). The raw reading is kept off-chain.
3. **Oracle verdict (private).** The oracle validates the reading against rules or a dataset and posts a verdict for $\mathsf{H}(\mathsf{reading})$. For the prototype this is a synthetic dataset; in production it could be a utility feed, AMI export, or multi-oracle consensus.
4. **Credential anchoring (private).** The RA anchors a credential hash $\mathsf{H}(\mathsf{VC})$ to $\mathsf{H}(\mathsf{reading})$ in `DataVer`. This provides a verifiable linkage that a credential was issued for a verified reading without publishing the credential itself.
5. **L2 mint (public).** The holder calls `mintWithProof(...)` on the L2 token contract. The call includes a minimal disclosure tuple (epoch, type, quantity, policy nonce), a hidden commitment value, and an issuer authorization signature. The L2 token verifies the signature, computes dtHash, rejects duplicates, mints the ERC-20 tokens, and emits an L2→L1 message to record the mint on the L1 anchor.
6. **L1 anchor (public audit).** The L1 anchor records the mint parameters keyed by dtHash. Only authenticated cross-chain calls are accepted.

4.5 State Machine and Duplicate Prevention

Figure 2 provides a protocol-level state machine view of the system. It complements Figure 1 by focusing on the sequence of states and the conditions that enable transitions (e.g., commitment then oracle verdict then anchoring then minting). This is particularly useful for reasoning about duplicate prevention: the mint transition is only allowed once per unique dtHash computed from the Disclosure Tuple.

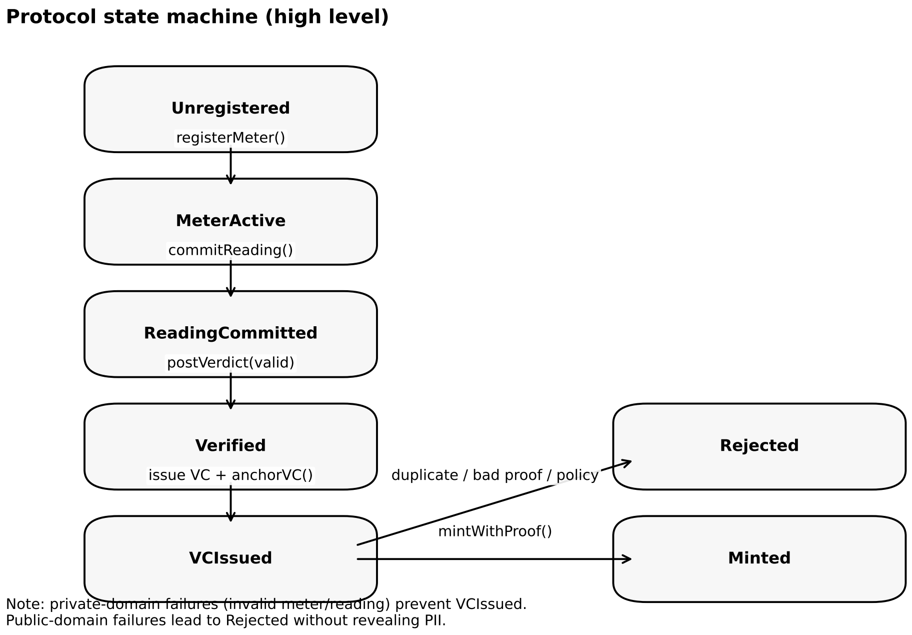
Protocol state machine (high level)
Unregistered
registerMeter()
MeterActive
commitReading()
ReadingCommitted
postVerdict(valid)
Verified
issue VC + anchorVC()
VCIssued
Rejected
duplicate / bad proof / policy
mintWithProof()
Minted
Note: private-domain failures (invalid meter/reading) prevent VCIssued.
Public-domain failures lead to Rejected without revealing PII.

Figure 2: Protocol state machine: private verification (commitment and verdict), credential hash anchoring, L2 minting with duplicate prevention, and L1 anchoring as an auditable record.

A mint is considered a claim of the form:

$$\mathrm{DT} = (\mathrm{epochIndex}, \mathrm{typeCode}, \mathrm{qtyKWh}, \mathrm{policyNonce}).$$

The duplication key dtHash is computed as:

$$\mathsf{dtHash} = \mathrm{keccak256}(\mathrm{abi.encodePacked}(\mathrm{epochIndex}, \mathrm{typeCode}, \mathrm{qtyKWh}, \mathrm{policyNonce})).$$

On the L2 token contract, dtHash indexes a mapping that is set on first successful mint and never cleared. Any subsequent attempt to mint the same tuple reverts with a custom error `DuplicateDT()` before any external effects.

4.6 Assets and Adversary Model

Assets. The security-critical assets include:

- **Issuer key material**: the authorization key used by `DemoIssuerVerifier`.
- **Duplication mapping**: the set of used `dtHash` values.
- **Anchor storage**: the L1 record of mint parameters keyed by `dtHash`.
- **Private-domain indexes**: commitments and verdicts in `DataVer`.

Adversaries. Adversaries are modeled as:

- A malicious holder attempting replay, forging authorization, or bypassing duplication prevention.
- An external attacker attempting to spoof L2→L1 anchoring or call anchors directly.
- A compromised oracle attempting to post incorrect verdicts in the private domain.
- A passive observer attempting to infer private attributes from public events.

5.0 Data Model, Canonicalization, and Authorization Interface

5.1 Design Principle: Minimal On-Chain Disclosure

The public chain should not contain enough information to identify a site or meter. Therefore the L2 mint path uses only:

- A minimal disclosure tuple (epoch, energy type, quantity, policy nonce),
- A *hidden commitment* value that can bind off-chain evidence without revealing it, and
- A proof/authorization that attests the tuple is eligible.

In this prototype, the proof is an ECDSA signature by a demo issuer key over a deterministic digest. The digest construction is public and ensures that the signature binds to the specific tuple and commitment.

5.2 Disclosure Tuple (DT) and `dtHash`

Field definitions.

- **epochIndex** (`uint64`): a discrete index for a reporting period. For example, a weekly epoch index can be computed off-chain as $\lfloor (t - t_0)/604800 \rfloor$, where t is a UTC timestamp.
- **typeCode** (`uint16`): an application-defined energy type code. In the demo scripts, `1` is used as an example value.
- **qtyKWh** (`uint256`): quantity of verified renewable generation in kWh. This quantity determines the token amount minted (one token per kWh in the default implementation).
- **policyNonce** (`uint128`): reserved for policy variants, seasonal bonuses, or future protocol evolution. In the demo it is `0`.

Canonical encoding. To prevent bypasses via ambiguous encoding, DT fields are encoded using Solidity's ABI packed encoding with fixed-width integer types. The resulting `dtHash` is deterministic across all callers for the same tuple.

5.3 Hidden Commitment

The hidden commitment is a `bytes32` value included in the signed digest and in events. It can represent a commitment to private attributes (e.g., meter hash, site hash, or a credential hash). The L2 contract does not interpret the commitment; it treats it as an opaque binding value. This design supports future extensions where the commitment is computed from a verifiable credential or a selective disclosure proof transcript.

5.4 Authorization Digest and Verifier Interface

Verifier interface. The L2 token contract depends on a single function:

Listing 1: Proof verifier interface used by the L2 token contract (excerpt).

```
interface IProofVerifier {
    function verify(
        address holder,
        uint64 epochIndex,
        uint16 typeCode,
        uint256 qtyKWh,
        uint128 policyNonce,
        bytes32 hiddenCommitment,
        bytes calldata proof
    ) external view returns (bool);
}
```

Digest construction. The demo verifier computes:

digest = keccak256(abi.encode(DOMAIN, holder, epochIndex, typeCode, qtyKWh, policyNonce, hidden

The domain separator is a constant string, preventing replay across protocols that might use the same tuple format. The holder address is included to prevent third-party replay of an issuer signature intended for a specific holder.

Signature verification. `DemoIssuerVerifier.verify(...)` checks that the ECDSA signature corresponds to the configured issuer address. It uses OpenZeppelin's `ECDSA` library to recover the signing address from the digest.

5.5 Relationship to Selective Disclosure Credentials

The demo verifier is intentionally simple. In a production-grade privacy-preserving design, `proof` would be a selective disclosure presentation proof derived from a verifiable credential, potentially based on BBS signatures [2] or a zk proof system. The rest of the architecture remains compatible:

- The verifier interface remains a single boolean predicate over disclosed fields and a commitment.
- The L2 contract remains responsible for duplication prevention, minting, and L1 anchoring.
- The private domain remains responsible for validation and issuance decisions.

5.6 Units, Rounding, and Policy Choices

The implementation treats `qtyKWh` as an integer quantity. Any rounding or aggregation is intended to occur in the private domain before issuance. This avoids on-chain floating-point representations and prevents accidental over-issuance.

The policyNonce provides a forward-compatible way to introduce policy changes without redefining the duplication key semantics. For example, if a future policy introduces seasonal multipliers, the policyNonce can encode the policy epoch so that the same (epoch, type, quantity) tuple under different policy regimes does not collide unintentionally.

6.0 Protocol Specification

This section specifies the protocol as an ordered set of steps with explicit preconditions and postconditions. Figure 3 provides a compact view of the end-to-end control flow for the "happy path," including the separation between the private verification domain, the public L2 mint transaction, and the later L1 anchor execution.

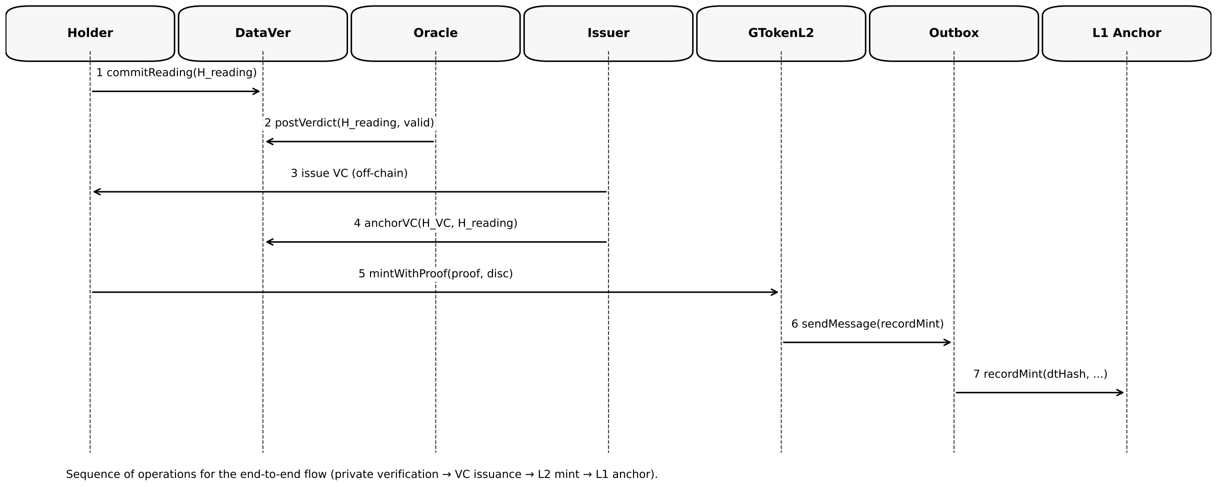
Holder
DataVer
Oracle
Issuer
GTokenL2
Outbox
L1 Anchor
1 commitReading(H_reading)
2 postVerdict(H_reading, valid)
3 issue VC (off-chain)
4 anchorVC(H_VC, H_reading)
5 mintWithProof(proof, disc)
6 sendMessage(recordMint)
7 recordMint(dtHash, ...)
Sequence of operations for the end-to-end flow (private verification → VC issuance → L2 mint → L1 anchor).

Figure 3: End-to-end protocol sequence: meter registration, reading commitment, oracle verdict, VC hash anchoring, L2 mint, and L1 anchor recording via authenticated L2→L1 messaging.

6.1 Private-Domain Protocol

6.1.1 Meter Registration

Inputs. A meter is represented by a 32-byte hash `meterHash`. The contract also stores `cert`, an opaque byte array intended to contain a certification blob (e.g., a signed statement or a structured record). The on-chain system stores only keccak256(`cert`) as `certHash`.

Transaction. `MetReg.registerMeter(meterHash, cert, status)`.

Preconditions.

- Caller must hold `GA_ROLE`.
- The meter must not already be registered.

Postconditions.

- The meter hash is recorded with its certificate hash and active flag.
- Event `MeterRegistered(meterHash, certHash, status)` is emitted.

6.1.2 Meter Verification

Transaction. `MetReg.verifyMeter(meterHash)`.

Preconditions. Caller must hold `RA_ROLE`.

Postconditions. Returns `true` if and only if `meters[meterHash].active` is true.

6.1.3 Reading Commitment

Transaction. `DataVer.commitReading(readingHash)`.

Preconditions.

- No role is required for committing.
- The commitment must not already exist in the mapping.

Postconditions.

- `committedReadings[readingHash]` is set to true.
- Event `ReadingCommitted(readingHash, from)` is emitted.

6.1.4 Oracle Verdict Posting

Transaction. `DataVer.postVerdict(readingHash, valid, oracleSig)` (or a two-argument convenience overload).

Preconditions.

- Caller must hold `ORACLE_ROLE`.
- The readingHash must be committed (`committedReadings[readingHash] == true`).

Postconditions.

- A verdict record is stored with `valid` and `verifiedAt`.
- Event `OracleVerdictPosted(readingHash, valid, oracleSig)` is emitted.

6.1.5 Credential Hash Anchoring

Transaction. `DataVer.anchorVC(vcHash, readingHash)`.

Preconditions.

- Caller must hold `RA_ROLE`.
- A verdict for `readingHash` must exist and must be valid.
- `vcHash` must not already be anchored.

Postconditions.

- The mapping `vcHash` → `readingHash` is recorded.
- Event `VCAnchored(vcHash, readingHash)` is emitted.

6.2 Public-Domain Protocol (L2 Minting and L1 Anchoring)

6.2.1 L2 Mint Request

Transaction. `GTokenL2.mintWithProof(epochIndex, typeCode, qtyKWh, policyNonce, hiddenCommitment, proof)`.

Preconditions.

- The verifier must accept the proof (`verifier.verify(...) == true`).
- The duplication key `dtHash` computed from the tuple must not have been used before.

Postconditions.

- `usedDT[dtHash]` is set to true.
- The holder receives `qtyKWh` tokens via `_mint(holder, qtyKWh)`.
- Event `Minted(holder, dtHash, epochIndex, typeCode, qtyKWh)` is emitted.
- An L2→L1 message is dispatched to the L1 anchor, requesting `recordMint(dtHash, ...)`.

6.2.2 L1 Anchor Recording (Generic Messenger Pattern)

Transaction. `GTokenAnchor.recordMint(dtHash, to, gtAmount, epochIndex, typeCode, qtyKWh)`.

Preconditions.

- The contract must be configured with a messenger address and allowlisted L2 sender.
- `msg.sender` must equal the configured messenger.
- The messenger-reported originating sender must equal the allowlisted L2 token address.
- The `dtHash` must not already be anchored.

Postconditions.

- Anchor storage is recorded under `dtHash`.
- Event `MintAnchored(dtHash, to, gtAmount, epochIndex, typeCode, qtyKWh)` is emitted.

6.2.3 L1 Anchor Recording (Arbitrum Outbox Pattern)

Transaction. `GTokenAnchorArb.recordMint(dtHash, to, gtAmount, epochIndex, typeCode, qtyKWh)`.

Preconditions.

- The contract must be configured with a bridge address and allowlisted L2 sender.
- `msg.sender` must equal `IBridge(bridge).activeOutbox()`.
- `IOutbox(msg.sender).l2ToL1Sender()` must equal the configured allowlisted L2 token address.
- The `dtHash` must not already be anchored.

Postconditions. Identical to the generic anchor case: store anchor info and emit `MintAnchored`.

Rationale. This pattern matches Arbitrum's authentication model: only the outbox executes L2→L1 messages, and it exposes the originating L2 sender identity [6, 7]. The anchor therefore rejects direct calls from EOAs and rejects outbox calls if the origin is not the configured L2 contract.

6.3 Failure Modes and Error Semantics

The contracts use explicit custom errors rather than string-based reverts. This improves clarity and reduces gas overhead. Relevant failure modes include:

- `DuplicateDT()` when the same `dtHash` is reused on L2.
- `BadProof()` when verifier validation fails.
- `NotFromMessenger()` or `NotFromBridge()` when anchors are called outside their authenticated pathway.
- `NotFromAuthorizedL2Sender()` when the cross-chain origin does not match the allowlisted L2 contract.
- `AlreadyAnchored()` when the same `dtHash` is anchored more than once.
- Private-domain errors such as `NotCommitted`, `VerdictMissing`, and `VerdictInvalid`.

6.4 Policy Extensions and Compatibility

The prototype intentionally keeps policy minimal. However, several policy extensions are straightforward:

- **Mint factor / conversion.** Instead of 1 token per kWh, mint amount could incorporate factors for energy type or region. This would require defining whether the factor is part of DT (to preserve duplication semantics) or derived from configuration.
- **Epoch policy enforcement.** A production system could enforce that epochIndex is within a sliding window, or that epochIndex is not too far in the past.
- **Revocation.** Revocation of meters or credentials is handled in the private domain; the public token is not retroactively revoked in this MVP. Future work could add a "retirement" mechanism or a revocation registry.

7.0 Smart Contract Design and Engineering

This section describes the contract suite, emphasizing interfaces, storage layouts, and security patterns. The implementation uses Solidity `^0.8.23` and OpenZeppelin libraries for ERC-20 and access control [3].

7.1 Private-Domain Contracts

7.1.1 `MetReg`: Meter Registry

Purpose. `MetReg` is a minimal registry that records meter status and a certificate hash. It is not intended to store PII or full certificate contents. The certificate bytes are stored only as a hash to provide tamper-evidence.

Storage. `MetReg` stores:

- `mapping(bytes32 => Meter) meters;`
- where `Meter` contains `bytes32 certHash;` and `bool active;`

Key functions.

- `registerMeter(bytes32 meterHash, bytes cert, bool status)`: GA-only; initializes a meter record.
- `updateMeterStatus(bytes32 meterHash, bool active)`: GA-only; toggles meter status.
- `verifyMeter(bytes32 meterHash) returns (bool)`: RA-only; returns active status.

Events. `MeterRegistered` and `MeterStatusUpdated` are emitted for audit and indexing.

7.1.2 `DataVer`: Verification Index

Purpose. `DataVer` stores three types of state:

1. Reading commitments: `readingHash` → `committed`.
2. Oracle verdicts: `readingHash` → `{valid, oracleSig, verifiedAt}`.
3. Credential anchors: `vcHash` → `readingHash`.

It is designed as an index that can be queried and audited without storing private readings.

Storage. `DataVer` uses:

- `mapping(bytes32 => bool) committedReadings;`
- `mapping(bytes32 => OracleVerdict) verdicts;`
- `mapping(bytes32 => bytes32) private _vcToReading;`

Key functions.

- `commitReading(bytes32 readingHash)`: open to any caller; rejects duplicate commitments.
- `postVerdict(bytes32 readingHash, bool valid, bytes oracleSig)`: oracle-only; requires commitment exists.
- `anchorVC(bytes32 vcHash, bytes32 readingHash)`: RA-only; requires verdict exists and is valid; rejects duplicate anchors.

Events and compatibility. The contract emits `ReadingCommitted`, `OracleVerdictPosted`, and `VCAnchored`. For developer convenience, it also includes alias functions matching earlier drafts (e.g., `storeVCCommitment`), but the tests and scripts use the core API.

7.2 Proof Verification Module

7.2.1 `IProofVerifier` Interface

The L2 minting contract depends on an external verifier contract through `IProofVerifier`. This interface design has two engineering benefits:

1. The token contract remains small and focused on minting, duplication, and messaging.
2. Cryptographic verification logic can evolve independently. For example, an ECDSA verifier can be replaced by a selective-disclosure verifier without modifying the token logic.

7.2.2 `DemoIssuerVerifier` (ECDSA Placeholder)

Purpose. `DemoIssuerVerifier` implements `IProofVerifier` and verifies an issuer signature over a deterministic digest. It is a deployable stand-in for a stronger credential proof.

Key design decisions.

- The digest includes `holder` to prevent signature replay by other addresses.
- The digest includes `hiddenCommitment` to bind the authorization to a specific private claim.
- A domain separator constant is used to avoid cross-protocol replay.

Limitations. ECDSA signatures do not provide selective disclosure or zero knowledge. They do, however, provide an unforgeable authorization primitive that can be measured and deployed easily. This aligns with the capstone emphasis on delivering a working artifact.

7.3 Public-Domain Contracts (L2)

7.3.1 `GTokenL2`: ERC-20 Minting on L2

Purpose. `GTokenL2` mints transferable tokens representing verified green generation. It enforces:

- Authorization via `IProofVerifier.verify(...)`.
- Duplicate prevention using dtHash.
- L2→L1 anchoring via a messenger contract.

Inheritance and libraries. `GTokenL2` inherits `ERC20` and `AccessControl` from OpenZeppelin.

Storage and state.

- `IProofVerifier public verifier;`
- `mapping(bytes32 => bool) public usedDT;`
- `address public l1Anchor;`
- `address public messenger;`

Mint logic. The mint logic is intentionally minimal and auditable:

1. Compute `dtHash`.
2. Revert if `usedDT[dtHash]` is already true.
3. Require `verifier.verify(...)` returns true.
4. Set `usedDT[dtHash] = true`.
5. Mint `qtyKWh` ERC-20 tokens to the caller.
6. Emit `Minted(...)`.
7. Dispatch a cross-chain message to L1 anchor via `sendMessage`.

7.3.2 `GTokenL2Arb`: Arbitrum-Specific L2 Minting

Purpose. `GTokenL2Arb` is an Arbitrum-specific version of the L2 token that uses the `ArbSys` precompile (`0x000...64`) to send an L2→L1 message. The mint and duplicate prevention semantics remain identical.

7.4 Public-Domain Contracts (L1 Anchors)

7.4.1 `GTokenAnchor`: Messenger-Authenticated L1 Anchor

Purpose. `GTokenAnchor` stores an immutable L1 record of L2 mints. It enforces that only authenticated cross-chain calls from the configured messenger and allowlisted L2 sender are accepted.

7.4.2 `GTokenAnchorArb`: Arbitrum Outbox-Authenticated L1 Anchor

Purpose. `GTokenAnchorArb` provides the same anchor record but using Arbitrum's outbox authentication. It stores the Arbitrum bridge address as an immutable constructor parameter and determines the active outbox at call time.

7.5 Local Mocks and Test Harness

To make cross-chain authentication testable without external networks, the repository includes mocks:

- `MockCrossDomainMessenger`: simulates messenger behavior and returns a configurable `xDomainMessageSender`.

- `MockArbBridge` and `MockArbOutbox`: simulate `activeOutbox` and `l2ToL1Sender`.

These mocks enable deterministic unit tests that validate both rejection paths and success paths for L1 anchors.

7.6 Safety Patterns and Engineering Choices

Custom errors. All contracts use custom errors rather than revert strings.

Checks-effects-interactions. Minting follows CEI: check duplicate and proof, set `usedDT`, then mint and dispatch the cross-chain message.

Bounded execution. No contract iterates over user-controlled dynamic arrays; all operations are constant-time per call, except bounded cryptographic verification in the verifier.

No proxy upgrades. The prototype avoids proxy upgrades to reduce surface area.

8.0 Security and Privacy Analysis

8.1 Security Goals

- **SG-1 (Authorization):** A mint succeeds only if the caller presents a valid authorization proof for the disclosed tuple and commitment.
- **SG-2 (Duplication prevention):** For a fixed disclosure tuple DT, at most one successful mint can occur.
- **SG-3 (Authenticated anchoring):** Anchor records on L1 can only be written via the legitimate cross-chain mechanism and must originate from the allowlisted L2 token contract.
- **SG-4 (Auditability):** Every mint produces machine-indexable events on L2 and a corresponding anchor record on L1.

8.2 Privacy Goals

- **PG-1 (No PII on public chain):** The mint interface and events do not require meter identifiers, addresses, or raw readings.
- **PG-2 (Minimal linkability):** Public observers should not be able to link an issuance to a particular meter or site from on-chain data alone.

8.3 Threat Analysis: L2 Minting

8.3.1 Replay and Duplicate Mint Attacks

A holder may attempt to mint multiple times with the same tuple, either by replaying the same proof/signature or by recomputing equivalent inputs. The duplication mapping `usedDT[dtHash]` ensures a second mint attempt reverts with `DuplicateDT()` before token minting occurs. Because `dtHash` is computed from fixed-width typed inputs, alternate encodings cannot bypass the mapping.

8.3.2 Forged Authorization Proofs

An attacker attempts to mint without issuer approval. For the demo verifier, forging requires producing an ECDSA signature that recovers to the configured issuer address. Under standard assumptions of ECDSA security, this is infeasible without the issuer private key. The digest also binds the signature to the holder address, preventing third-party replay.

8.3.3 Front-Running and Mempool Attacks

A mempool observer cannot steal a mint by copying calldata because the issuer authorization includes the holder address. Replaying from a different address fails verification.

8.4 Threat Analysis: L1 Anchoring

Direct calls to anchors revert because both anchor variants require that calls originate from the correct cross-chain mechanism. In addition, both anchors validate the originating L2 sender identity, preventing unauthorized L2 contracts from anchoring records.

8.5 Privacy Analysis

The L2 mint and L1 anchor expose only the minimal tuple, dtHash, and a commitment value. No meter identifiers or raw readings are placed on public chain. Privacy depends on operational separation: the preimages of commitments and the mapping from commitments to real identities must remain off-chain.

8.6 Residual Risks

The main residual risks are issuer key compromise, oracle centralization in the private domain, and policy mis-specification of DT fields.

9.0 Implementation and Tooling

9.1 Repository Structure

The implementation is organized as a Hardhat project with Solidity contracts, TypeScript scripts, and tests. Key directories include:

- `contracts/private`: `MetReg.sol`, `DataVer.sol`.
- `contracts/l2`: `GTokenL2.sol`, `GTokenL2Arb.sol`.
- `contracts/l1`: `GTokenAnchor.sol`, `GTokenAnchorArb.sol`.
- `contracts/mocks`: mock cross-chain contracts and `DemoIssuerVerifier.sol`.
- `scripts`: demos, deployment helpers, and gas report scripts.
- `test`: unit and integration tests.

9.2 Build and Execution Environment

The system compiles under Hardhat with Solidity compiler settings:

- **Solidity:** `0.8.23`
- **Optimizer:** enabled, 200 runs
- **EVM target:** Paris
- **viaIR:** enabled

9.3 Scripts and Commands

The repository defines npm scripts for a reproducible workflow:

- `npm test`: run Hardhat tests.
- `npm run demo:v2`: run the local end-to-end L2 mint + immediate L1 anchor demo using mocks.
- `npm run demo:arb:mock`: run the Arbitrum L1 anchor demo using `MockArbOutbox` and `MockArbBridge`.
- `npm run gas:v2`: generate a local gas report JSON file in `docs/`.

9.4 Testing Strategy

The unit test suite covers: private-domain correctness, L2 mint behavior, and L1 anchoring authentication (including both rejection and acceptance paths).

10.0 Evaluation Methodology

10.1 Metrics

- **Correctness:** unit tests and demo scripts demonstrate expected state transitions and revert cases.
- **Gas usage:** a reproducible gas script measures L2 minting, verifier checks, anchor configuration, and modeled L1 execution gas for anchoring.
- **Interpretability:** results explicitly separate the L2 mint transaction from the later L1 execution in real rollup deployments.

10.2 Experimental Setup

All measurements in this report were produced on a local Hardhat network using the repository scripts. The gas script deploys contracts fresh, executes representative mint and anchor calls, and records the gas used and calldata sizes.

11.0 Results

11.1 Functional Correctness Results

The final test suite executes successfully with all tests passing. The local demo script `demo:v2` produces a mint transaction, shows the holder balance increment, confirms that the L1 anchor has recorded the mint, and then demonstrates that a duplicate mint attempt reverts with `DuplicateDT()`.

11.2 Gas Report Results (Local)

Table 3 reproduces the measured gas report generated by `npm run gas:v2`. These values are for a local EVM execution and are intended to support relative comparison and engineering reasoning.

Table 3: Measured gas usage (local Hardhat) for key V2 operations.

Operation	**Gas Used**	**Calldata (bytes)**	**Notes**
L2 `mintWithProof` (no L1 relay)	116,310	356	Includes duplicate-guar ERC-20 mint, and messa dispatch event
`DemoIssuerVerifier.verify` (estimate-Gas)	32,343	-	ECDSA-path verifier es mate
L1 anchor `setConfig`	49,053	68	One-time admin configur tion
L1 relay execution: `recordMint`	157,794	388	Models the later L1 exec tion (separate transacti on real rollups)
Combined local mint+anchor (for unit test)	241,600	356	Convenience for tests; n representative of a sing real L2 tx

11.3 Interpretation and Comparison

Two interpretation notes are essential:

L2 vs L1 cost separation. On a real optimistic rollup, the holder submits a single L2 transaction for `mintWithProof`. The L1 anchor is executed later as an L1 transaction.

Therefore, the combined local cost number is not a meaningful “user gas cost” on L2.

Verifier cost is modular. The verifier is intentionally replaceable. The gas report isolates the mint baseline so future work can measure the marginal cost of stronger privacy-preserving proof verification.

11.4 Arbitrum Anchor Demo (Mock)

The `demo:arb:mock` script demonstrates:

1. Direct calls revert with `NotFromBridge()`.
2. Outbox calls with incorrect origin revert with `NotFromAuthorizedL2Sender()`.
3. Outbox calls with correct origin succeed and store anchor info.
4. Duplicate anchors revert with `AlreadyAnchored()`.

12.0 Deployment and Operations

12.1 Local (No-Funds) Workflow

A reproducible workflow without funded wallets:

1. `npm install`
2. `npm test`
3. `npm run demo:v2`
4. `npm run demo:arb:mock`
5. `npm run gas:v2`

12.2 Testnet Deployment Workflow (Requires Funds)

Optional scripts exist for Sepolia (L1) and Arbitrum Sepolia (L2). They require RPC URLs and a funded deployer key (see Appendix B).

12.3 Monitoring and Auditability

A monitoring pipeline can index L2 `Minted` events and confirm L1 `MintAnchored` events appear after the rollup confirmation period.

13.0 Governance, Policy, and Ethical Considerations

13.1 Governance Model

Governance is modeled via explicit roles (GA, RA, ORACLE) and admin configuration on anchors.

13.2 Policy Definition and the Disclosure Tuple

The choice of DT fields is a policy decision; the selected DT balances duplicate prevention with minimal public disclosure.

13.3 Ethical Use and Limitations

This prototype does not claim regulatory equivalence to official RECs. Any compliant deployment would require program alignment, audits, and regulator acceptance.

14.0 Project Management

14.1 Milestones

1. Implement private-domain contracts and tests.
2. Implement L2 minting with duplication prevention and verifier interface.
3. Implement L1 anchoring with authentication (generic messenger and Arbitrum outbox).
4. Add local mock demos and a reproducible gas report.
5. Integrate deployment scripts for optional testnet execution.

14.2 Risk Register (Summary)

Key risks: cryptographic complexity (mitigated by verifier modularity), cross-chain correctness (mitigated by tests/mocks), and time/scope (mitigated by prioritizing deployability).

15.0 Conclusions and Future Work

15.1 Conclusions

This report presented a deployable engineering prototype for tokenized green credit issuance that emphasizes privacy, duplicate prevention, and scalability. The system separates private verification from public minting and records L2 issuance on L1 via authenticated anchoring.

15.2 Future Work

1. Replace ECDSA authorization with selective disclosure credentials or zk proofs [2].
2. Move from single-oracle to multi-oracle threshold verdicts in the private domain.
3. Add pausability and incident response hardening on minting contracts.
4. Extend anchors into cross-chain duplication prevention across multiple L2 deployments.
5. Measure real testnet fees and latency when funding and time permit.

Bibliography

[1] W3C, "Verifiable Credentials Data Model v1.1," W3C Recommendation, Mar. 2022. [Online]. Available: `https://www.w3.org/TR/vc-data-model/`

[2] T. Looker, V. Kalos, A. Williams, and M. Lodder, "The BBS Signature Scheme," IRTF CFRG Internet-Draft, draft-irtf-cfrg-bbs-signatures-09, Jul. 2025. [Online]. Available: `https://datatracker.ietf.org/doc/draft-irtf-cfrg-bbs-signatures/`

[3] OpenZeppelin, "ERC20 and AccessControl Contracts," OpenZeppelin Contracts documentation, accessed 2025. [Online]. Available: `https://docs.openzeppelin.com/contracts/`

[4] Alberta Utilities Commission, "Micro-Generator Application Guideline," Jul. 2013 (PDF), accessed 2025. [Online]. Available: `https://hme.ca/connecttothegrid/Alberta%20Utilities%20Commission%20Micro-Generator%20Application%20Guideline%202013%2007%2005%20v1-3.pdf`

[5] Center for Resource Solutions, "Renewable Energy Tracking Systems," Jun. 2017 (PDF), accessed 2025. [Online]. Available: `https://resource-solutions.org/wp-content/uploads/2017/06/Renewable_Energy_Tracking_Systems.pdf`

[6] Offchain Labs / Arbitrum, "Outbox and L2→L1 sender authentication (`l2ToL1Sender`)," Arbitrum documentation and published source code, accessed 2025. [Online]. Available: `https://docs.arbitrum.io/`

[7] Arbitrum Foundation / Offchain Labs, "Arbitrum contract addresses (including Arbitrum Sepolia Bridge and Outbox)," Arbitrum documentation, accessed 2025. [Online]. Available: `https://docs.arbitrum.io/`

[8] V. Buterin and others, "EIP-196: Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128," Ethereum Improvement Proposal, 2017. [Online]. Available: `https://eips.ethereum.org/EIPS/eip-196`

[9] V. Buterin and others, “EIP-197: Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128,” Ethereum Improvement Proposal, 2017. [Online]. Available: `https://eips.ethereum.org/EIPS/eip-197`

[10] G. Wood, “Ethereum: A Secure Decentralised Generalised Transaction Ledger (Yellow Paper),” Ethereum, accessed 2025. [Online]. Available: `https://ethereum.github.io/yellowpaper/paper.pdf`

[11] Nomic Foundation, “Hardhat Documentation,” accessed 2025. [Online]. Available: `https://hardhat.org/`

Appendix A: Reproducibility Commands

Listing 2: Core reproducibility commands.

```
npm install
npx hardhat clean
npm test

# Local demos (no external networks)
npm run demo:v2
npm run demo:arb:mock

# Gas report (writes JSON into docs/)
npm run gas:v2
```

Appendix B: Environment Variable Template (Optional Testnets)

The repository includes `.env.example`. To run deployments on Sepolia and Arbitrum Sepolia, populate:

- `DEPLOYER_PRIVATE_KEY`
- `SEPOLIA_RPC_URL`
- `ARBITRUM_SEPOLIA_RPC_URL`
- `ARB_L1_BRIDGE_SEPOLIA` (see Arbitrum docs [7])

Appendix C: Key Contract Interfaces (Excerpts)

C.1 L2 Mint Interface

Listing 3: L2 mint interface (excerpt).

```
function mintWithProof(
    uint64 epochIndex,
    uint16 typeCode,
    uint256 qtyKWh,
    uint128 policyNonce,
    bytes32 hiddenCommitment,
    bytes calldata proof
) external returns (bytes32 dtHash);
```

C.2 L1 Anchor Interface

Listing 4: L1 anchor interface (excerpt).

```
function recordMint(
    bytes32 dtHash,
    address to,
    uint256 gtAmount,
    uint64 epochIndex,
    uint16 typeCode,
    uint256 qtyKWh
) external;
```