Provenance

A Modular Blockchain Protocol for Product Provenance and Authenticity Verification

Hachem Nasri

October 11, 2025

Table of Contents

Abstract

Fake products are a major problem. They hurt customer trust, cause companies to lose money, and damage a brand's good name. It's also hard to prove that a digital record is truly connected to the real physical item. This is known as the "physical twin" problem. This project solves this with a special blockchain system built using the **EIP-2535 Diamond Standard**, which keeps the code organized and makes it easy to upgrade in the future.

Each product gets a unique **ERC-721 token**, or a "digital twin," that follows its journey on the blockchain with clear steps, from when a **Brand** creates it to when a **Retailer** sells it. To create a secure link between the real product and its digital twin, each item has a special microchip called a **Physically Unclonable Function (PUF)** that cannot be copied. To check if a product is real, a consumer's device sends a random challenge to the chip. The chip creates a unique, one-time signature as a response, which is then verified by the smart contract. The contract uses up the challenge so it can't be used again, which stops hackers from replaying old verifications. By combining a strong code structure with secure physical chips, this system proves a product's authenticity with a high degree of certainty and brings trust back to the supply chain.

1. Introduction & Problem Domain

1.1. The Global Impact of Counterfeiting and Illicit Trade

Fake products are a massive problem around the world. It is not just about small-time sellers; it is a massive industry that costs real companies billions of dollars every year in lost sales. When a market is full of fakes, it becomes hard for customers to trust even the real brands, which hurts the company's reputation that they worked hard to build. Regular people also lose money when they buy a low-quality item that they were led to believe was genuine.

1.2. Defining the Problem Space: A Multi-faceted Threat

The global market is saturated with illicit goods, a problem that extends far beyond simple knockoffs. This issue presents a multi-faceted threat that erodes consumer trust, diminishes brand equity, and poses significant economic and safety risks. Our system is designed to address the following interconnected challenges:

1.2.1 Outright Counterfeiting

The most direct threat is the proliferation of counterfeit goods, unauthorized replicas that are often visually indistinguishable from genuine products. These items are typically manufactured with inferior materials and lack quality control, deceiving consumers and leading to direct revenue loss for legitimate brands. For the consumer, there is no reliable method at the point of sale to definitively verify a product's legitimacy.

1.2.2 Illicit Market Diversion (The Grey Market)

A more nuanced challenge is the diversion of *authentic* products into unauthorized "grey markets." This occurs when goods are stolen during transit, sold through unofficial liquidators, or imported from a different region to exploit price differences. While the product itself is genuine, it bypasses the brand's official quality controls, warranty agreements, and distribution channels. This undermines a brand's pricing strategy and severs the chain of trust, as consumers may be purchasing legitimate but mishandled or improperly stored goods.

1.2.3 The Provenance Black Box

The trade in counterfeit and pirated goods is not a minor issue; rather, it represents a widespread global enterprise that causes significant economic and social damage. This multi-trillion-dollar shadow economy undermines legitimate commerce, resulting in substantial revenue losses for companies and decreased tax income for governments. For consumers, the effects are direct and often personal, leading to financial losses from purchasing fraudulent products that do not meet expectations in quality and durability. Beyond the economic impact, the proliferation of fake goods systematically damages a brand's reputation, weakening the value and trust that companies spend years building. This creates an environment where consumers become increasingly skeptical, and the integrity of authentic products is consistently under threat.

1.3 Project Objectives and Scope

The main goal of this project is to design a blockchain-based system that tracks products from their origin. We aim to provide consumers with a reliable way to verify the authenticity of a product right at the store. We will create a transparent and secure on-chain record that follows the product's journey from the brand to an authorized retailer.

To keep our research focused, this paper will concentrate on designing the core on-chain protocol, which includes the smart contracts, the different actor roles, and how the status of a product changes over time. We will exclude the physical implementation of QR tags, the development of a consumer-facing mobile app, and the financial payments between the brand and retailers from the scope of this project.

1.4. Contribution and Approach

Our approach to solving this problem is to create a **"digital twin"** for every physical product using a Non-Fungible Token (NFT). This project introduces a unique protocol that tracks each product's journey through clear on-chain statuses, from the factory to the retailer. The key innovation is the final verification step at the point of sale, where an authorized retailer must use their cryptographic signature to finalize the purchase. This action creates a permanent and trustworthy record on the blockchain, confirming the product's authenticity for the consumer at the most critical moment.

2. Background & Technical Foundations

2.1. Blockchain and Smart Contracts: The Foundation of Trust

At its core, a blockchain acts as a shared digital notebook that users pass around a vast network of computers. Once someone fills a page with information, they seal it and add it to the chain, which ensures it cannot be secretly changed later. This design makes the blockchain incredibly secure and transparent. Smart contracts serve as automated "if-then" rules that reside within this notebook. They consist of code that executes exactly as written without requiring an intermediary. For our project, this setup is perfect because we can establish rules for our products and ensure that everyone follows them, creating a system built on trust.

2.2. The ERC-721 Standard: Creating Digital Twins

To represent each physical product digitally, we use a specific type of token called a **Non-Fungible Token (NFT)**. We chose the **ERC-721** standard because it is the official blueprint for creating these on the blockchain. Unlike money, where every dollar is the same, each ERC-721 token is one-of-a-kind, just like a unique serial number. This allows us to create a "digital twin" for every single product. The standard provides built-in functions to identify the **owner** and assign a **unique ID** to each item, exactly what is needed for a tracking system.

2.3. Role-Based Access Control: Securing the Protocol

A supply chain system requires different permissions for different users; for example, only a registered brand should be able to create new products. To manage this securely, this protocol implements a robust **Role-Based Access Control (RBAC)** system. Instead of concentrating all power in a single "owner" account, which introduces a central point of failure, RBAC distributes permissions across several distinct roles like ADMIN_ROLE, BRAND_ROLE, and RETAILER_ROLE.

The core logic for this system is encapsulated within a custom **LibAccessControl library**, which is based on OpenZeppelin's battle-tested AccessControl patterns. This library is called internally by the various facets to enforce permissions for all state-changing functions. This architectural choice ensures that each participant can only perform the actions they are explicitly permitted to, creating a secure, organized, and gas-efficient on-chain governance model.

2.4. The EIP-2535 Diamond Standard

This protocol leverages the EIP-2535 Diamond Standard as its core architectural framework to facilitate a modular, scalable, and upgradeable on-chain system.

2.4.1 Core Concepts

The standard decouples a contract's persistent identity and state from its logical implementation through a few key components:

- **The Diamond Proxy:** A central, lightweight contract that serves as the system's permanent address and state-holder. It contains minimal logic itself.
- **Facets:** Independent, stateless contracts that contain discrete logical implementations (e.g., AdminFacet, BrandFacet).
- **Dispatch Mechanism:** The Diamond's fallback function acts as a dynamic router. It uses a mapping to look up which Facet is responsible for an incoming function call and forwards the call using delegatecall, executing the Facet's logic within the proxy's storage context.

2.4.2 Strategic Rationale

The decision to adopt this advanced architecture was driven by three key advantages:

- **Modularity & Organization:** The pattern allows the protocol's complex domain logic to be compartmentalized into distinct, manageable Facets. This enforces a strict separation of concerns, which improves code organization and mitigates the risk of unintended state interactions.
- **Scalability Beyond EVM Limits:** This architecture effectively circumvents the EVM's 24 KB contract bytecode size limit. By splitting the code across multiple Facets, the system can scale to a level of complexity that would be impossible within a monolithic contract.
- **Secure Upgradeability:** The diamondCut function provides a formal, on-chain governance interface to atomically add, replace, or remove Facets. This is crucial for long-term maintenance, bug fixes, and future feature extensions without requiring a disruptive and costly data migration.

2.4.3 Protocol Implementation

This project applies the Diamond Standard in the following way:

- **The Proxy:** The Provenance contract serves as the central Diamond proxy, the system's sole entry point, and the vessel for the AppStorage.
- **Facet Taxonomy:** The system's logic is divided into two categories:
    - **Application Logic Facets** (AdminFacet, BrandFacet, etc.) encapsulate the core, domain-specific business rules.
    - **Standard Interface Facets** (ERC721Facet, AccessControlFacet) provide the public-facing API for token standards and role inspection.
- **Internal Logic:** Shared, internal logic is further encapsulated in **Libraries** (LibAccessControl, LibERC721). This facet -> library pattern enhances security and gas efficiency for inter-component communication.

3. Protocol Design & System Architecture

This chapter details the core technical design of our protocol, moving from high-level architecture to specific on-chain logic. We will examine the four key pillars of the system: the **actors** who participate, the **data structures** that organize information, the **product lifecycle** that defines its journey, and the **core functions** that enable these interactions.

3.1. System Actors

Our protocol is designed around four key actors, each with a specific set of responsibilities and permissions enforced by the smart contract.

3.1.1 The Protocol Authority (ADMIN)

- **Definition:** The top-level administrative body that governs the entire ecosystem.
- **Core Responsibilities:** This actor is responsible for vetting and registering legitimate Brands. They manage the official list of participants and have the power to set a brand's status to Active or Suspended, ensuring the overall integrity of the platform.

3.1.2 The Brand

- **Definition:** An officially registered company that manufactures the products.
- **Core Responsibilities:** The Brand is the only actor that can create, or "mint," new product NFTs. They are responsible for starting the product's journey on the blockchain and for managing their list of Authorized Retailers.

3.1.3 The Authorized Retailer

- **Definition:** A brand-approved partner, such as a physical store or an official online shop.
- **Core Responsibilities:** The Retailer acts as a crucial link in the supply chain. Their main jobs are to cryptographically certify that they have received a shipment from the Brand and to execute the final, secure sale to the Consumer.

3.1.4 The Consumer

- **Definition:** The end-user who purchases the product.
- **Core Responsibilities:** The Consumer uses the system to verify a product's authenticity before buying it. By participating in the final sale, they receive the product's NFT, which acts as a permanent and undeniable proof of authentic ownership.

3.2. System Overview: Use Case Diagram

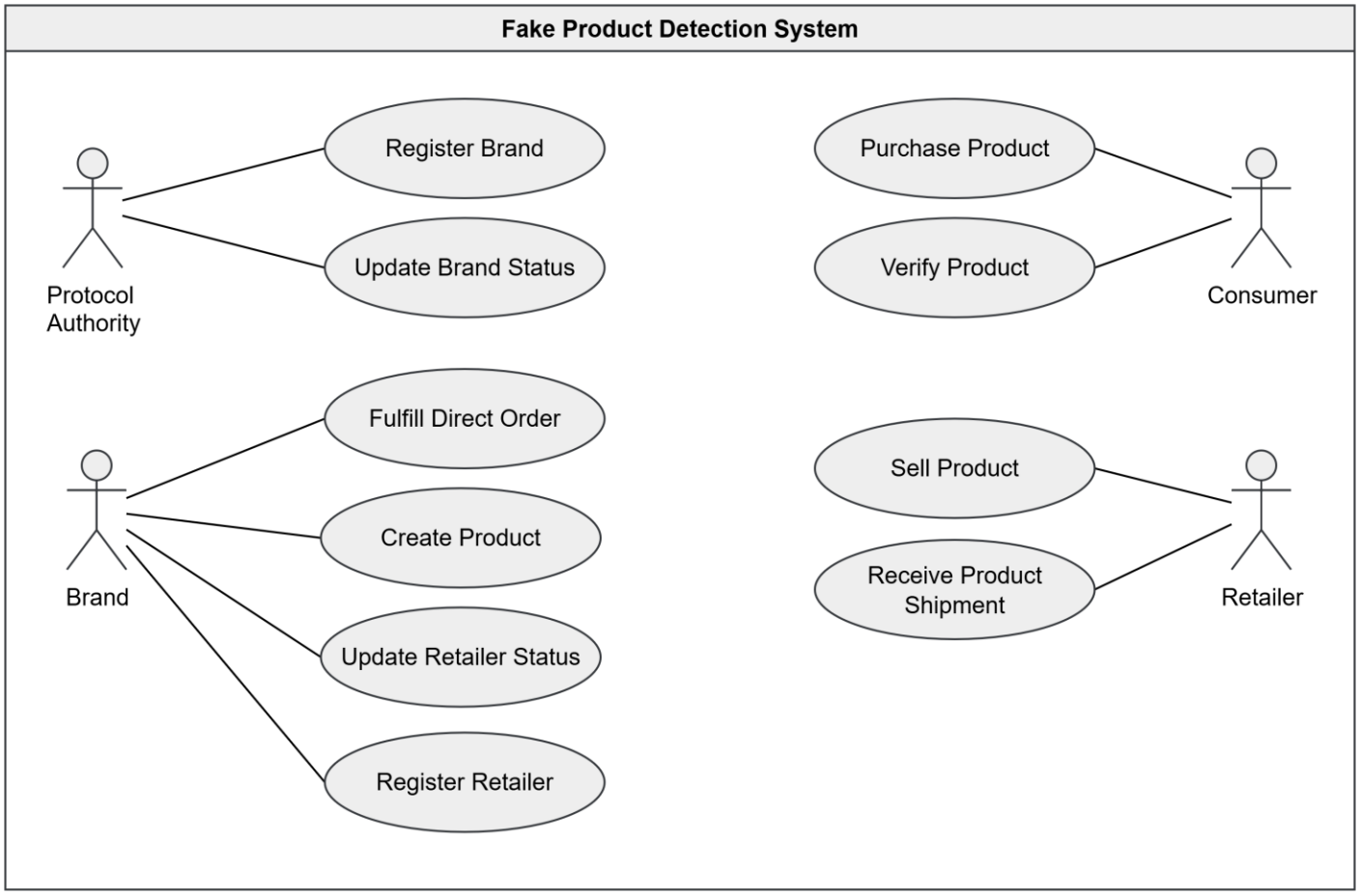

Figure 1: System Actors and Core Use Cases

This Use Case Diagram illustrates the high-level functionalities of the Fake Product Detection System and the interactions between its four key actors. The system is designed to manage the entire product lifecycle from creation to sale, with clear roles and permissions for each participant.

- **Protocol Authority**: This actor serves as the system administrator, with exclusive responsibilities for managing the brands on the platform. Their functions include registering new brands and updating the status of existing brands.
- **Brand**: The Brand actor has the most diverse set of functions, covering product creation and supply chain management. They can create products, register and manage the status of retailers, and fulfill direct orders.
- **Retailer**: This actor is the intermediary in the supply chain. Their responsibilities include receiving product shipments from the brand and selling the product to the end consumer.
- **Consumer**: The Consumer is the final actor in the product's journey. They can interact with the system to verify a product's authenticity and to purchase the product.

3.3. Core Data Structures

To keep our on-chain data organized, clean, and efficient, we use custom data types called structs. These act as blueprints for the key entities in our system. The class diagram below provides a complete visual overview of the smart contract's architecture, including its relationship with parent contracts, its core data structures, and the functions that manage them.

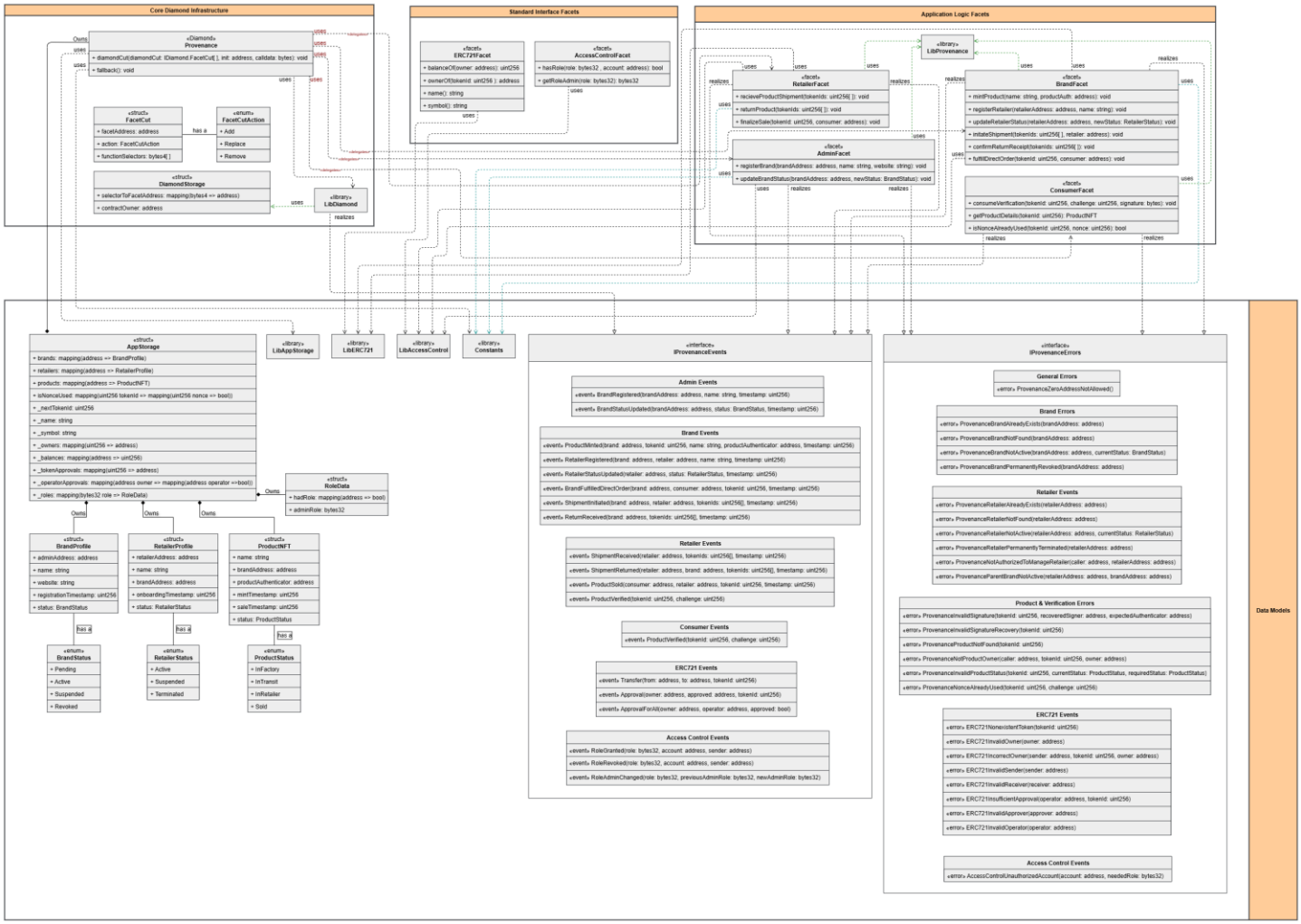

Figure 2: System Architecture Class Diagram. A high-resolution, scalable version of this diagram is available in the project repository at: ***documentation/diagrams/globalClassDiagram.svg***

The diagram above illustrates the complete on-chain architecture of the system, which is implemented using the **EIP-2535 Diamond Standard** for maximum modularity and upgradeability. The key components are as follows:

- **The Provenance Contract (Diamond Proxy):** This is the central, lightweight smart contract that serves as the permanent address and state holder for the entire system. It contains minimal logic

itself; its primary role is to use its fallback function to delegate all external calls to the appropriate facet contract.

- **Facets (Modular Logic):** Instead of inheriting functionality, the system's logic is compartmentalized into independent facet contracts. These facets are attached to the diamond and handle specific domains of functionality. They are divided into **Application Logic Facets** (like AdminFacet and BrandFacet) and **Standard Interface Facets** (ERC721Facet, AccessControlFacet), which provide the system's public-facing API.
- **Libraries (Internal Engine):** Core, reusable logic is encapsulated in internal libraries, such as LibAccessControl and LibERC721. This is a secure and gas-efficient pattern that allows different facets to share functionality without making complex external calls back through the diamond.
- **Data Models (Structs, Enums, Events & Errors):** The system's state and communication patterns are defined by several key data structures. **Structs** (BrandProfile, RetailerProfile, ProductNFT) and **Enums** (BrandStatus, ProductStatus) create the blueprint for on-chain state. In addition, the system uses two central interfaces to define its communication contract: **IProvenanceEvents** defines all the events the system can emit, creating a reliable data feed for off-chain applications, while **IProvenanceErrors** defines all custom errors, providing clear and gas-efficient reasons for transaction failures.
- **Relationships:** The diagram illustrates several key relationships. A **Composition** relationship (◆—) shows that the Provenance contract is composed of both its storage structs and its many facets. A **<<use>> Dependency** (- - ->) shows how the facets call the internal libraries to execute their logic.

3.4. Product Lifecycle & State Machine:

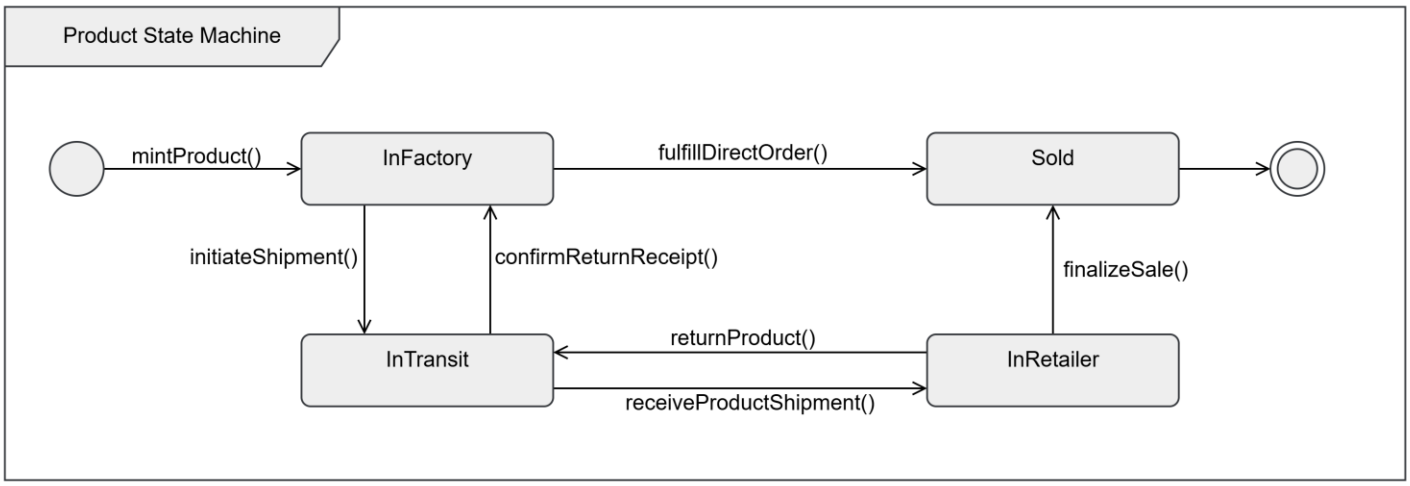

Figure 3: Product Lifecycle State Machine

The lifecycle of each product within our protocol is managed by a formal state machine, which is visually represented in the diagram above. This ensures that a product can only exist in one of a few well-defined states at any given time and can only transition between these states through specific, permissioned functions.

The process begins when a *Brand* calls the **mintProduct()** function. This action creates the unique product NFT and places it in its initial state: **InFactory**. From this point, the product can follow one of several paths.

The standard distribution path involves the *Brand* calling **initiateShipment()** to move the product to the **InTransit** state. Upon physical receipt of the item, the *Retailer* then calls **receiveProductShipment()**, transitioning the product to the **InRetailer** state, where it is officially available for sale. The primary path concludes when the *Retailer* calls **finalizeSale()**, moving the product to its terminal state, **Sold**.

The protocol also accounts for two alternative paths. First, a *Brand* can sell directly to a consumer by calling **fulfillDirectOrder()**, which moves the product's state directly from **InFactory** to **Sold**. Second, a *Retailer* can return an unsold product. This is a two-step process where the *Retailer* first calls **returnProduct()** to move the item from **InRetailer** to **InTransit**. The loop is completed when the *Brand* confirms its return by calling **confirmReturnReceipt()**, moving the product from **InTransit** back to the **InFactory** state.

Once a product enters the **Sold** state, its journey within the supply chain is considered complete, and no further state transitions are possible.

3.5. Core Functions

This section describes the primary functions of the protocol. This logic is compartmentalized across several independent facet contracts, and each function is protected by a robust role-based access control system to ensure the integrity of the on-chain state machine.

3.5.1. registerBrand()

- **Purpose:** Onboards a new, vetted company onto the platform.
- **Access Control:** Resides in the **AdminFacet**. This function can only be called by an address holding the **ADMIN_ROLE**.
- **Key Logic:** The function creates a new BrandProfile struct with the company's details, sets its initial status to Pending, stores it in the central brands mapping, and grants the new company the **BRAND_ROLE**.

3.5.2. mintProduct()

- **Purpose:** Creates the on-chain "digital twin" for a new physical product.
- **Access Control:** Resides in the **BrandFacet**. This function can only be called by a registered Brand with an Active status.
- **Key Logic:** The function first confirms the calling brand's Active status. It then creates a new ProductNFT record, assigns it the next available tokenId, and makes an internal call to LibERC721.mint to create the token, assigning the Brand itself as the initial owner.

3.5.3. initiateShipment()

- **Purpose:** Begins the secure, on-chain transfer of a batch of products from a Brand to one of its authorized Retailers.
- **Access Control:** Resides in the **BrandFacet**. Can only be called by an address holding the **BRAND_ROLE**.
- **Key Logic:** It verifies that both the calling Brand and the specified Retailer are in an Active state. It then loops through the provided array of tokenIds, calling LibERC721.transferFrom to transfer ownership of each NFT to the Retailer and updating its status to InTransit.

3.5.4. receiveProductShipment()

- **Purpose:** Allows an authorized Retailer to cryptographically confirm that they have taken physical possession of a shipment.
- **Access Control:** Resides in the **RetailerFacet**. Can only be called by an address holding the **RETAILER_ROLE**.
- **Key Logic:** The function verifies the Retailer's Active status. It then loops through the tokenIds, checking for each one that the caller is the legitimate owner and that the product's status is InTransit. If all checks pass, it updates the status of each product to InRetailer.

3.5.5. finalizeSale()

- **Purpose:** Executes the final, secure sale of a product to a consumer, creating a permanent, auditable record of the transaction.
- **Access Control:** Resides in the **RetailerFacet**. Can only be called by an authorized Retailer with an Active status.
- **Key Logic:** The function verifies the Retailer is active, is the current owner of the token, and that the product's status is InRetailer. It then calls LibERC721.transferFrom to transfer the NFT's ownership to the consumer, updates the product's status to Sold, and records the saleTimestamp.

3.5.6. consumeVerification()

- **Purpose:** Allows a consumer to cryptographically verify a product's physical authenticity via a challenge-response protocol, consuming a one-time-use signature to prevent replay attacks.
- **Access Control:** Resides in the **ConsumerFacet**. This is a public function and requires no special role.
- **Key Logic:** It takes a tokenId, a challenge (nonce), and a signature as input. The function first checks if the nonce has already been used for that tokenId. It then re-creates the expected message hash on-chain and uses ecrecover to derive the signer's address from the signature. Finally, it compares this recovered address to the official productAuthenticator address stored on-chain. If all checks pass, it marks the nonce as used, completing the verification

3.6. Key Interactions: Sequence Diagrams

This section provides a detailed, step-by-step visualization of the protocol's most critical functions. Each sequence diagram illustrates the flow of messages between the actors and the smart contracts to complete a specific task.

3.6.1. Interaction Flow: Minting a Product

This diagram illustrates the secure process for creating a new product NFT within the Diamond architecture. The sequence begins when the **Brand** actor calls mintProduct() on the central **:Provenance (Diamond Proxy)**. The proxy, acting as a router, immediately forwards the request via **delegatecall** to the **:BrandFacet**, where the core logic is executed. The **:BrandFacet** then performs a series of internal checks, first verifying the caller's BRAND_ROLE and Active status. If these checks pass, it stores the new product data and completes the process by making an internal call to the **LibERC721** library to mint the token.

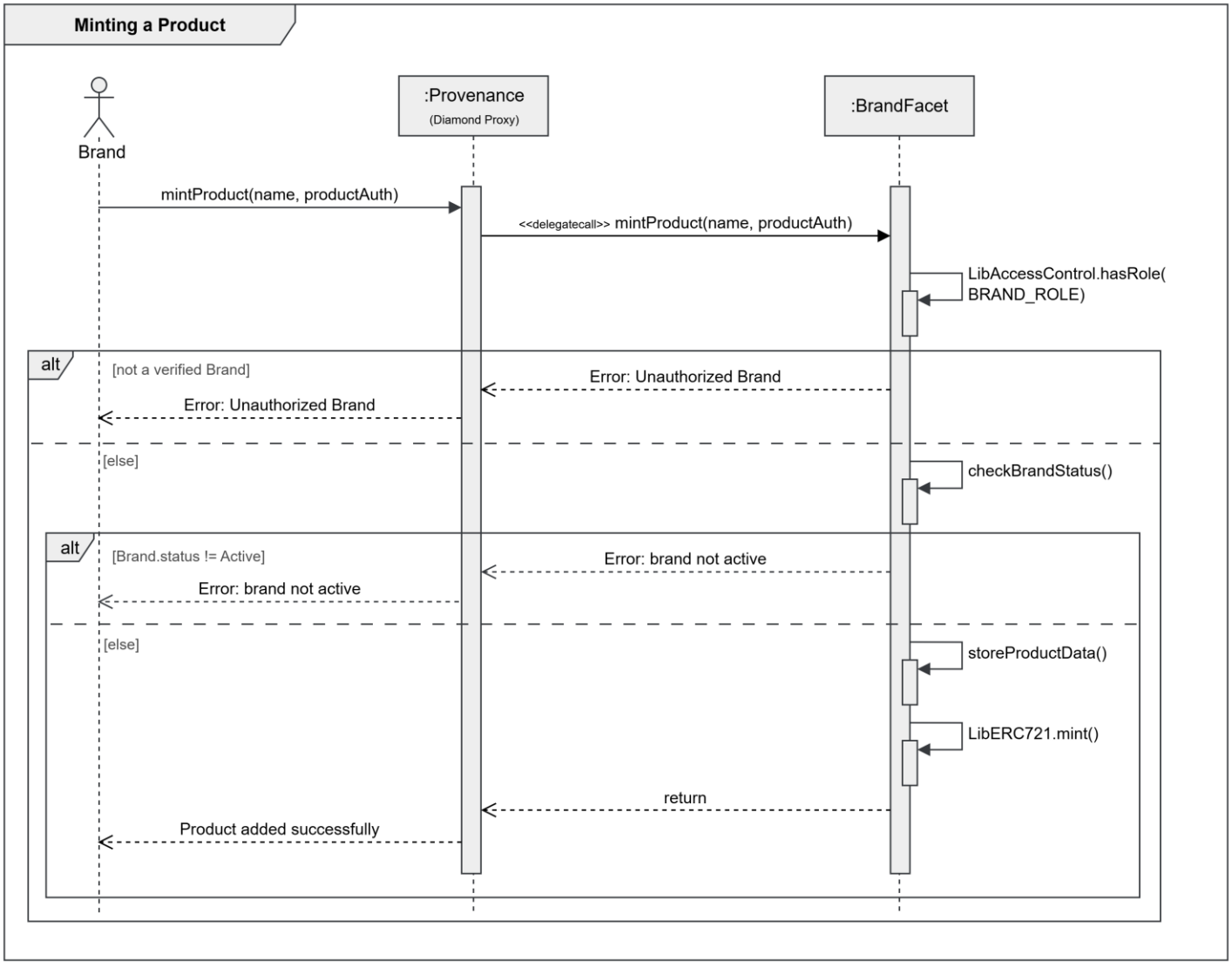

Figure 4: Sequence Diagram - Minting a Product

3.6.2. Interaction Flow: Initiating a Shipment

This sequence models the first half of the supply chain handoff, where a Brand securely transfers a batch of products to a Retailer. The process begins when the **Brand** actor calls initiateShipment() on the **:Provenance (Diamond Proxy)**. The proxy then forwards the call via **delegatecall** to the **:BrandFacet**, where the core logic is executed. The :BrandFacet first verifies the caller's role and confirms that both the Brand and the target Retailer are in an Active state. If these checks pass, it enters a loop to process each tokenId, making an internal call to **LibERC721.transferFrom** and setting the product's status to InTransit.

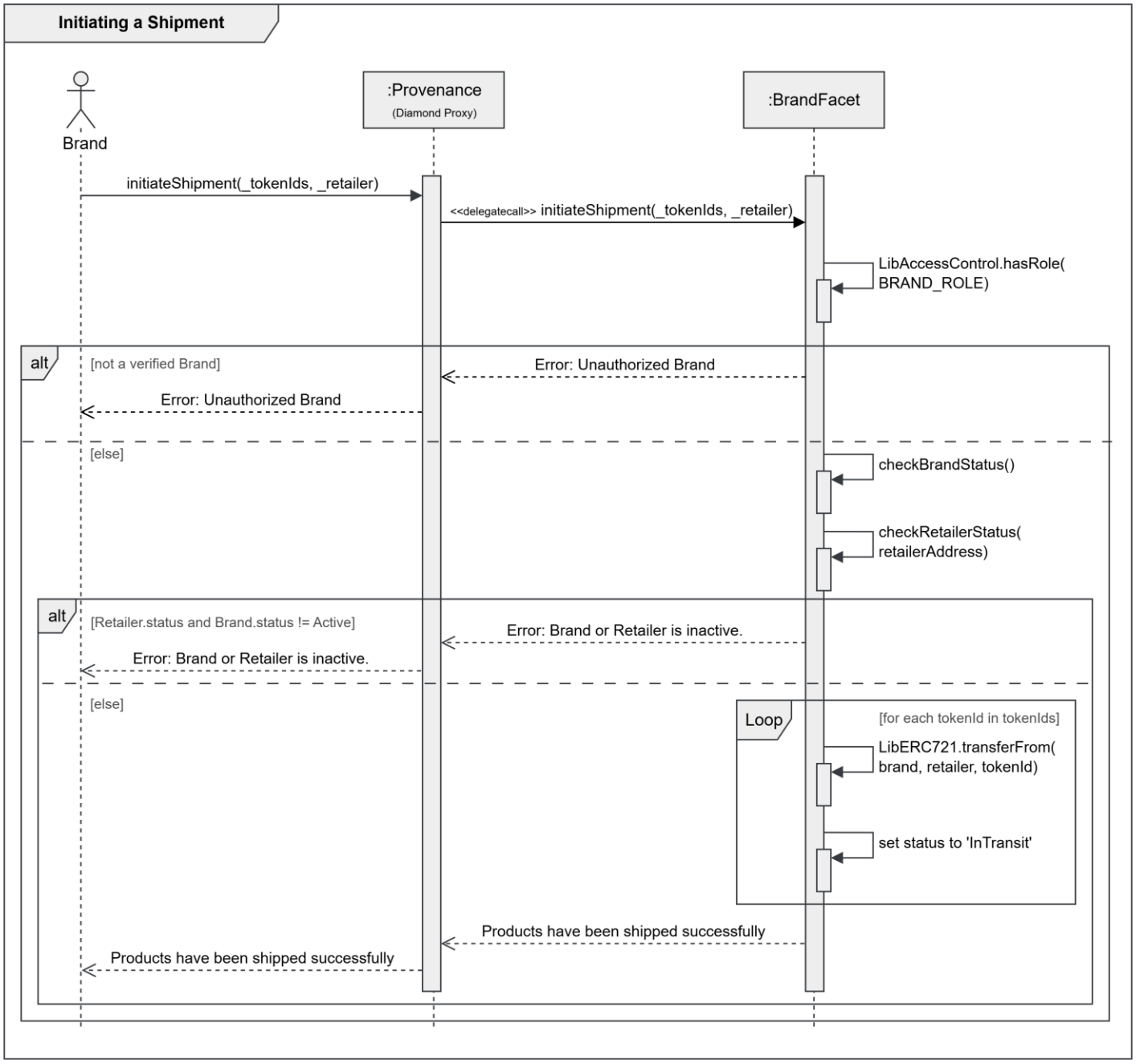

Figure 5: Sequence Diagram - Initiating a Shipment

3.6.3. Interaction Flow: Receiving a Shipment

This diagram illustrates the second half of the supply chain handoff, where the Retailer cryptographically confirms possession of the goods. The interaction starts when the **Retailer** actor calls receiveProductShipment() on the **:Provenance (Diamond Proxy)**. The proxy then forwards this call via **delegatecall** to the **:RetailerFacet**, which executes the core logic. The :RetailerFacet first verifies the caller's RETAILER_ROLE and Active status. It then enters a loop and, for each tokenId, confirms that the

caller is the token's owner and that its status is InTransit. If all conditions are met, it updates the product's status to InRetailer.

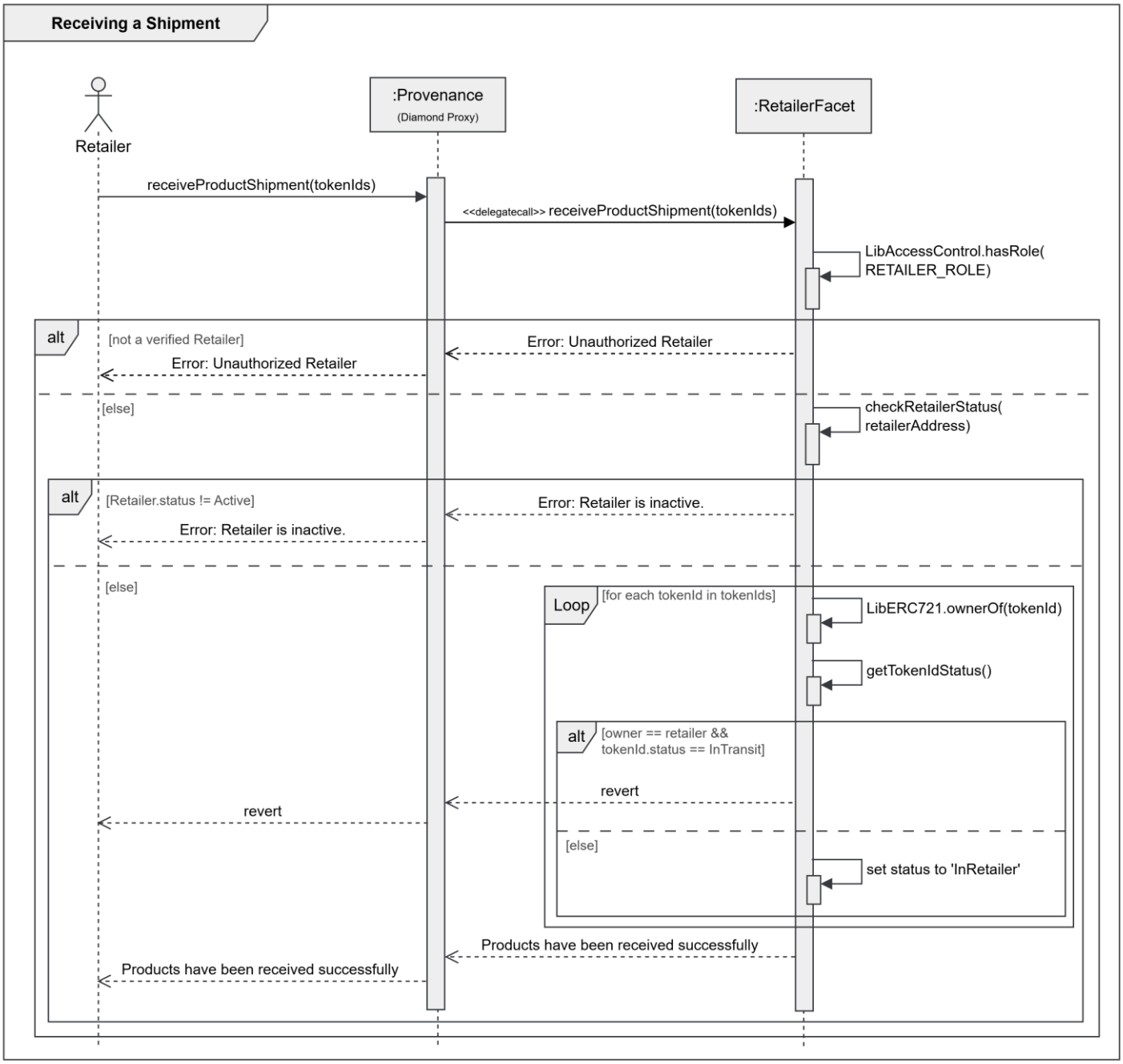

Figure 6: Sequence Diagram - Receiving a Shipment

3.6.4. Interaction Flow: Finalizing a Sale

This sequence shows the culmination of the product's journey: the secure, on-chain sale to a consumer. The **Retailer** calls finalizeSale() on the **:Provenance (Diamond Proxy)**, which forwards the call via **delegatecall** to the **:RetailerFacet**. The facet then verifies the retailer's role, active status, and the

product's InRetailer status. If all checks pass, it updates the internal state and makes a final library call to **LibERC721.transferFrom** to transfer the product's ownership to the consumer.

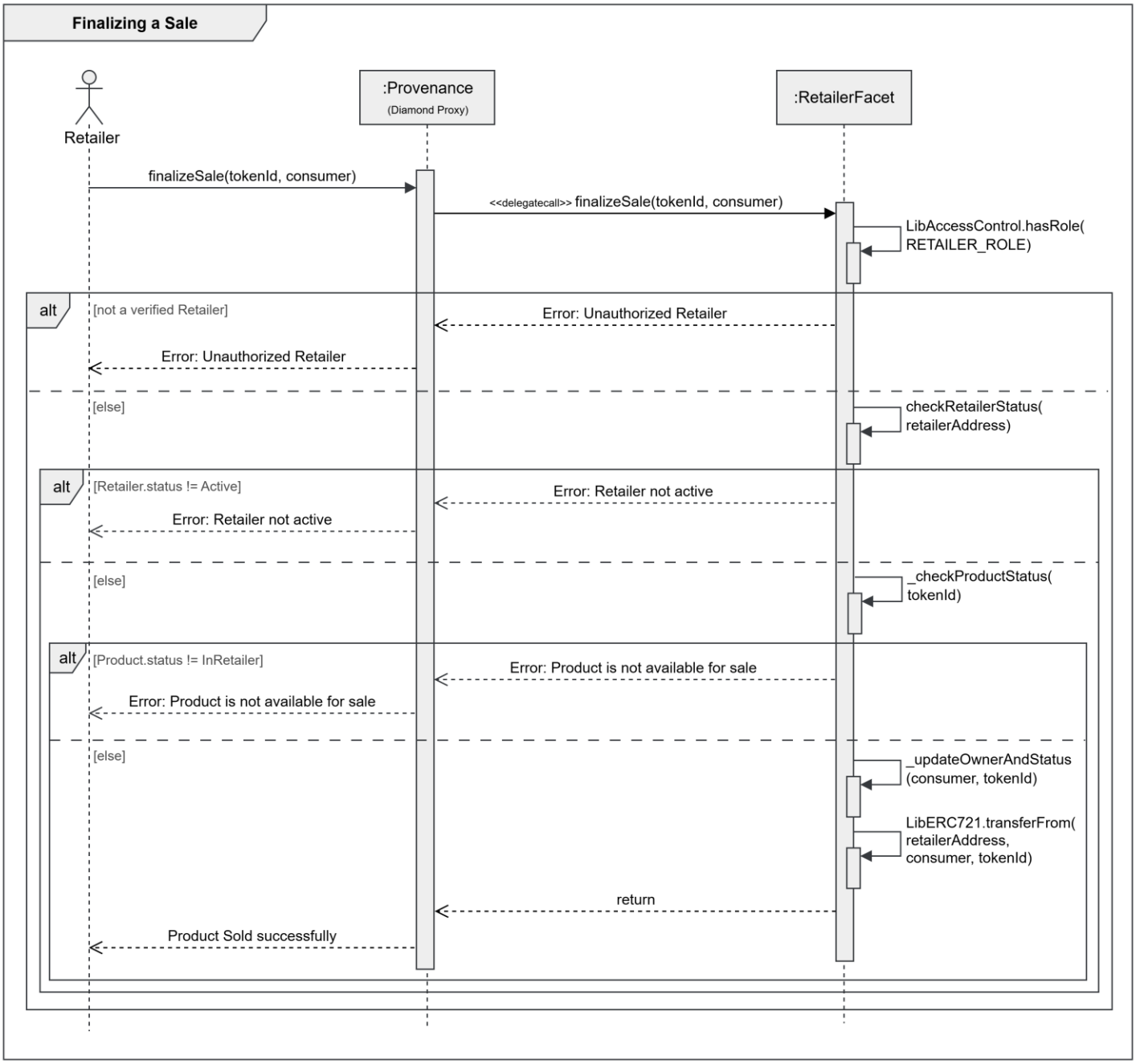

Figure 7: Sequence Diagram - Finalizing a Sale

3.6.5. Interaction Flow: Verifying a Product (Challenge-Response)

This diagram models the most critical security function of the protocol: the cryptographic verification of a product's physical authenticity. The interaction is divided into two distinct phases: an initial off-chain challenge-response with the PUF to generate a unique proof, and a subsequent on-chain transaction that validates and consumes this proof to prevent forgery.

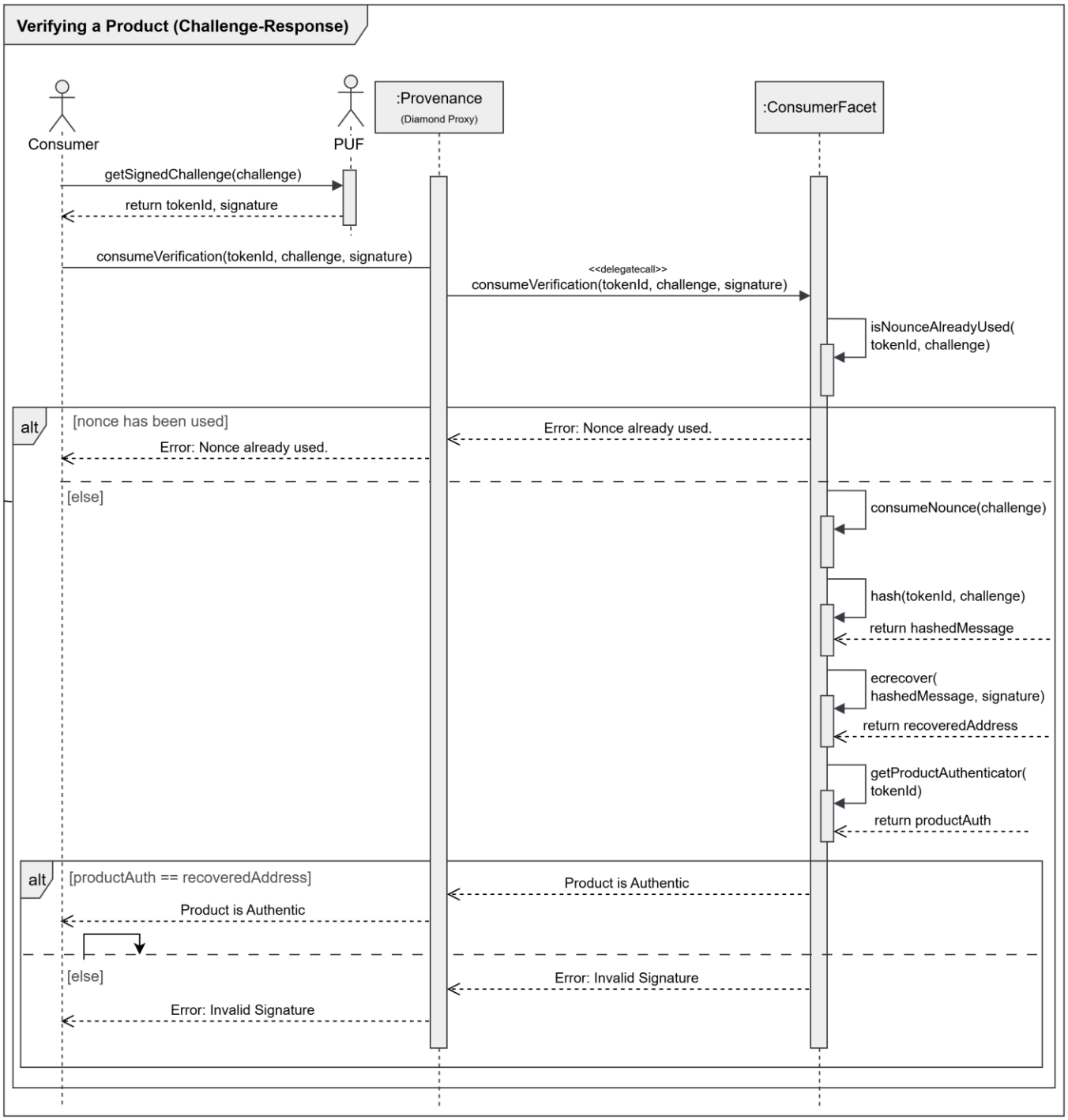

Figure 8: Sequence Diagram - Verifying a Product (Challenge-Response)

The initial off-chain interaction is designed to generate a single-use proof of physical possession. The Consumer's application generates a random, unpredictable number (the challenge) and presents it to the product's embedded PUF. The PUF, using its unique physical microstructure, generates a signature (the response) that is cryptographically tied to that specific challenge.

This proof is then submitted to the **Provenance** contract, which delegates the call to the **ConsumerFacet** for final, authoritative validation. The facet's **consumeVerification** function first performs a critical nonce check, querying its isNonceUsed mapping to ensure the challenge has not been previously consumed for this tokenId. This is the primary defense against replay and relay attacks. If the nonce is unique, the contract re-hashes the tokenId and challenge and uses ecrecover to recover the signer's address from the provided signature. This recovered address is then compared against the official productAuthenticator address stored on-chain for that product. Only if the nonce is unique and the addresses match does the verification succeed, providing the highest degree of cryptographic certainty of the product's authenticity.

4. Security Analysis

The security of this protocol is multi-layered, addressing threats at both the on-chain (protocol) level and the physical-to-digital (asset) level. The design combines standard smart contract best practices with a novel cryptographic approach to create a robust and trustworthy system.

4.1. On-Chain Protocol Security

This layer of security governs the interactions between the actors and the smart contract itself, ensuring that all on-chain actions are legitimate and adhere to the system's rules.

4.1.1. Role-Based Access Control (RBAC)

The primary security model is a robust, role-based access control (RBAC) system. Critical functions are protected by specific roles (**ADMIN_ROLE**, **BRAND_ROLE**, and **RETAILER_ROLE**), ensuring that only authorized participants can perform state-changing actions. The logic is encapsulated in a custom **LibAccessControl** library, based on OpenZeppelin's battle-tested patterns, which is called internally by each facet. This prevents unauthorized access across the entire protocol. The risk of a compromised key is further mitigated by the status management system (Active, Suspended), which enables the protocol's admin to rapidly deactivate any suspicious account.

4.1.2. State Machine Enforcement

The protocol's integrity is guaranteed by the strict enforcement of the product lifecycle's state machine. Every function that modifies a product's state includes require statements that validate the product's current status before allowing a transition to occur. For example, the finalizeSale function requires the product's status to be InRetailer. This prevents all invalid state transitions, such as selling a product that is still in transit or has already been sold.

4.1.3. Common Vulnerability Mitigation

The protocol is protected against common exploits by adhering to established security patterns. The key mitigations include:

- **Checks-Effects-Interactions Pattern:** All state-changing functions perform checks and update internal state *before* making any external calls, which is the primary defense against re-entrancy attacks.

- **Architectural Security:** The modular **EIP-2535 Diamond Standard** architecture compartmentalizes logic into separate facets. This reduces the attack surface of any single component and prevents complex, unintended interactions. Furthermore, using internal libraries (LibAccessControl, LibERC721) for core logic is inherently more secure than re-entrant external calls between facets.
- **Solidity Version:** The use of Solidity version ^0.8.20 provides built-in protection against integer overflow and underflow vulnerabilities.

4.2. Physical-to-Digital Security

This layer of security addresses the most difficult challenge in any digital twin system: ensuring the authenticity of the physical asset itself.

4.2.1. Solving the "Physical Twin" Problem with PUFs

A simple QR code or serial number can be copied, allowing a counterfeiter to place a valid identifier on a fake product. Our protocol addresses the "Physical Twin" problem by utilizing Physically Unclonable Functions (PUFs). Each product is embedded with a PUF microchip that has a unique "silicon fingerprint" derived from its physical structure, which is impossible to clone. This PUF is used to generate a unique, on-chain productAuthenticator address, creating an unforgeable link between the physical item and its digital twin.

4.2.2. Preventing Replay and Relay Attacks

While a PUF prevents cloning, a sophisticated attacker could still attempt to defeat the system by either recording a valid verification to "replay" later or by setting up a real-time "relay attack." Our protocol mitigates both of these threats through a mandatory **on-chain nonce consumption** model.

- **The Challenge-Response**: To verify a product, a user's app sends a random, one-time number (the challenge) to the PUF. The PUF generates a unique response (a signature).
- **On-Chain Consumption**: This (challenge, response) pair is sent to the smart contract. The contract first checks a usedNonces mapping to ensure this exact challenge has never been used for this product before. If it is new, the contract verifies the response and then immediately records the challenge as used. This on-chain check ensures that every verification is a unique, one-time event, making both replay and relay attacks impossible.

4.3. Remaining Trust Assumptions

While the PUF model provides cryptographic certainty of a product's *authenticity*, the system still requires a degree of trust in its registered actors to report the product's *logistical state*. The protocol must trust the Brand and Retailer to honestly call the initiateShipment and receiveProductShipment functions to reflect the product's real-world location. This remaining trust assumption, often referred to as the "Oracle Problem," represents a well-defined boundary of the system.

5. Implementation & Verification

5.1. Development Environment

The protocol was implemented and verified using a modern, professional-grade toolchain. The test suite, written in TypeScript, achieves over 95% line coverage on all core business logic, ensuring the system is robust and reliable. The key components of the development environment are as follows:

- **Solidity:** The smart contracts were written in Solidity ^0.8.20, leveraging modern language features and built-in security protections.
- **Hardhat:** A flexible and extensible Ethereum development environment used for compiling, deploying, testing, and debugging the smart contract.
- **OpenZeppelin Patterns:** While not directly inheriting from OpenZeppelin's standard contracts, the implementation is deeply informed by their battle-tested patterns. The core logic for the ERC-721 token standard and the Access Control system was carefully adapted into custom internal libraries (LibERC721, LibAccessControl) to function securely and efficiently within the EIP-2535 Diamond Standard architecture.
- **Ethers.js:** A comprehensive JavaScript/TypeScript library for interacting with the Ethereum blockchain, used for writing the test suite and deployment scripts.
- **TypeScript:** The language used for writing the test suite and deployment scripts, providing strong typing to prevent common errors.
- **Chai:** An assertion library used alongside the testing framework to write clear and expressive tests for the smart contract's logic.

5.2. Testing Strategy

To ensure the correctness, security, and reliability of the protocol, a comprehensive unit testing strategy was implemented using the Hardhat development framework. The test suite, written in TypeScript with the Ethers.js and Chai libraries, is organized by actor roles (Admin, Brand, Retailer, and Consumer) to systematically validate every function and logical path within the smart contract system.

The testing methodology focused on covering three critical areas:

- **Happy Path Validation:** Each function was tested to ensure it performs correctly under normal conditions. These tests verify that valid inputs from authorized users result in the expected state changes and the emission of the correct on-chain events with the proper arguments.
- **Failure Path and Revert Conditions:** Every potential failure point was explicitly tested to guarantee the contract is secure and behaves predictably. This included testing every require statement by simulating invalid conditions, such as:
    - **Access Control Violations:** Attempting to call protected functions from unauthorized accounts.
    - **State Machine Errors:** Attempting invalid state transitions, like trying to sell a product that is still InTransit or has already been sold.
    - **Invalid Inputs:** Passing invalid arguments, such as the zero address, to functions that prohibit them.
- **Security Scenario Testing:** Specific tests were designed to verify the protocol's defenses against common attack vectors. This included testing the consumeVerification function to ensure it correctly rejects fraudulent signatures from unauthorized wallets and prevents replay attacks by reverting when a nonce is used more than once.

This rigorous testing strategy resulted in over 95% line coverage on all core business logic, providing a high degree of confidence in the protocol's security and correctness.

5.3. Deployment

The deployment of the smart contract system was managed using Hardhat Ignition, a modern, declarative deployment tool that ensures robustness, repeatability, and resilience. Unlike traditional imperative scripts that execute a series of step-by-step transactions, Ignition allows for the definition of a desired on-chain state, which the tool then intelligently and safely achieves.

5.3.1 Deployed Contract Addresses

The following contracts have been deployed to the **Sepolia testnet** and are available for interaction. The primary entry point for the system is the Provenance (Diamond Proxy) address.

Primary Contracts

- **Provenance (Diamond Proxy):** 0x81eEB7A87E91f490FbcbFfCbd70793886aE83a59
- **DemoRoleFaucet:** 0x0442a21D30346d753664F5CB2fDee23C8D9689B5

Facet Contracts (Implementation Logic)

- **AdminFacet:** 0xEab7bD1CB91DEB3ae9ae24055c8491BD44ca49e8
- **BrandFacet:** 0xeceCC6A0E111c259A956Ed5ADD79a85b7F5052d8
- **RetailerFacet:** 0xfA749B3e946a03288a389AEDe1d5B4d0c1c02535
- **ConsumerFacet:** 0x8BBCcE1a4bC66031B3a2204b381b2E17aF7A4cAB
- **ERC721Facet:** 0x28E56E64f3841AD06875AEF4CBa6D78Bc2ea7f76
- **AccessControlFacet:** 0x4162174a021DAFcf7C3C8b85817C057237FDfF5b

5.3.2 Deployment Architecture

The deployment is executed via a single Ignition Module (DeployProvenance.ts) that defines a blueprint of the entire system architecture. This blueprint specifies three core categories of components:

1. **The Facet Contracts:** The module first defines the deployment of all the individual, stateless logic contracts (AdminFacet, BrandFacet, RetailerFacet, ConsumerFacet, etc.).
2. **The Provenance Diamond (Proxy):** The module then defines the deployment of the main Provenance contract. It passes the addresses and function selectors of all the facet contracts from the previous step into the constructor, which performs the initial diamondCut and assembles the diamond.
3. **The DemoRoleFaucet Contract:** Finally, the module defines the deployment of the DemoRoleFaucet as a separate, external contract, passing the address of the newly created Provenance diamond into its constructor to link the two.

5.3.3 Post-Deployment Configuration

A critical part of the deployment is the automated on-chain setup. After all contracts are deployed, the Ignition module defines a final, crucial transaction: a call to the grantRole function on the Provenance diamond.

This call grants the DEFAULT_ADMIN_ROLE, ADMIN_ROLE and BRAND_ROLE to the newly deployed DemoRoleFaucet contract. This automated setup transaction is what empowers the faucet to act as a trusted administrator for the demo, allowing it to grant roles to reviewers. This

declarative approach ensures that the entire system, the contracts and their initial permissions, is deployed and configured correctly in a single, reliable operation.

6. Conclusion & Future Work

6.1. Conclusion

This project successfully designed and implemented a decentralized provenance protocol to combat product counterfeiting by establishing a secure, unforgeable link between physical goods and their digital representations. The core contribution is a holistic framework that solves the "physical twin" problem by integrating Physically Unclonable Functions (PUFs) with a robust, on-chain state machine. The system's architecture, built on the EIP-2535 Diamond Standard, provides a modular, scalable, and upgradeable foundation, demonstrating a sophisticated approach to smart contract engineering.

Through a multi-layered security model, the protocol ensures integrity at every stage. On-chain, a hierarchical Role-Based Access Control (RBAC) system and a strictly enforced product lifecycle prevent unauthorized actions and invalid state transitions. At the physical layer, a cryptographic challenge-response mechanism, protected by an on-chain nonce consumption model, provides the highest degree of certainty of a product's authenticity while mitigating replay and relay attacks. The comprehensive test suite, achieving over 95% line coverage, provides strong evidence of the protocol's correctness and reliability. Ultimately, this work presents a complete and verifiable solution that restores trust and transparency to the supply chain.

6.2. Future Work

While this project provides a complete and functional end-to-end system, several compelling avenues exist for future research and improvement.

- **Hardware Integration and Automated Minting:** The current system relies on a simulated PUF and manual product minting. A critical next step would be to integrate the protocol with a real-world manufacturing line. The mintProduct function could be triggered automatically by the factory's software upon the successful enrollment of a physical PUF microchip. This would close the final gap in the automation process, ensuring that the creation of the physical product and its digital twin are a single, atomic event, further enhancing security and efficiency.
- **Privacy-Preserving Verification with Zero-Knowledge Proofs (ZKPs):** The current protocol stores product and ownership data publicly. Future work could explore a privacy-preserving model using Zero-Knowledge Proofs. In this architecture, a consumer could prove they own an

authentic product from a specific brand *without* revealing their wallet address or the product's unique ID to the public. The smart contract's role would evolve to that of a trustless on-chain verifier for these proofs, enabling confidential authenticity checks.

- **Decentralized Governance and Dispute Resolution:** The current system relies on a centralized ADMIN_ROLE to vet and manage brands. A significant enhancement would be to decentralize this governance. A DAO (Decentralized Autonomous Organization) comprised of trusted brands and stakeholders could vote on onboarding new participants. Furthermore, an on-chain dispute resolution mechanism could be introduced to handle real-world edge cases, such as a product being physically stolen while the NFT remains in the owner's wallet, adding another layer of social and legal robustness to the system.
- **Scalability and Off-Chain Data Indexing:** As the number of products scales into the millions, querying historical data directly from the blockchain will become inefficient for front-end applications. A future implementation should include an off-chain indexing service using a protocol like The Graph. This service would listen to on-chain events (ProductMinted, ShipmentInitiated, ProductSold, etc.) and organize this data into a high-performance, queryable database. This would provide the UI with a fast and powerful API to fetch data (e.g., "show all products sold by this retailer") without sacrificing the security of the on-chain source of truth.

Appendix

Appendix A: Glossary of Terms

ABI (Application Binary Interface):

The standard "instruction manual" for a smart contract. It's a JSON file that defines the contract's functions and events, allowing off-chain applications (like a website) to know how to interact with it.

Address:

A unique identifier on the blockchain, similar to a bank account number. It can represent a user's wallet or a smart contract.

AppStorage:

A project-specific term for the central struct that defines the storage layout for all the protocol's state variables. All facets interact with this single, shared storage structure inside the Diamond proxy.

Blockchain:

A distributed, immutable digital ledger that records transactions in a secure and transparent manner. It is the foundational technology that makes this system possible.

Bytecode:

The low-level, compiled version of Solidity code that is actually deployed and executed by the Ethereum Virtual Machine (EVM).

Challenge-Response:

A cryptographic security protocol where one party presents a question (the "challenge") and another party must provide a valid answer (the "response") to be authenticated. In this project, the consumer's device provides a random number as a challenge, and the PUF provides a valid signature as the response.

Delegatecall:

A low-level operation in Solidity that allows a contract (the Diamond proxy) to execute code from another contract (a facet) within its own storage context. This is the core mechanism of the EIP-2535 Diamond Standard.

Digital Twin:

A virtual, on-chain representation of a physical object. In this project, each product's ERC-721 token acts as its unique and verifiable digital twin.

EIP-2535 (Diamond Standard):

An advanced smart contract architecture for building modular and upgradeable systems. It uses a central proxy contract to delegate calls to independent "facet" contracts, allowing for a clean separation of logic and the ability to add or replace functionality after deployment.

ERC-721:

A standard for creating Non-Fungible Tokens (NFTs) on the Ethereum blockchain. It ensures that each token is unique and has a verifiable owner, making it the perfect standard for representing one-of-a-kind products.

Event:

A mechanism in Solidity for a smart contract to log activities on the blockchain. Off-chain applications can listen for these events to track state changes, such as when a new product is minted or sold.

Facet:

An independent smart contract that contains a specific subset of the protocol's logic. In this project, facets like AdminFacet and BrandFacet are attached to the main Provenance Diamond to provide its functionality.

Gas:

The fee required to execute a transaction or smart contract function on the Ethereum blockchain.

Hardhat:

A professional Ethereum development environment used to compile, deploy, test, and debug smart contracts.

Mapping:

A data structure in Solidity that stores key-value pairs, similar to a hash table or dictionary. This project uses mappings to associate addresses with brand profiles and token IDs with product data.

NFT (Non-Fungible Token):

A unique digital asset whose ownership is tracked on a blockchain. Unlike cryptocurrencies, each NFT is one-of-a-kind.

Nonce:

A "number used once." In cryptography, a nonce is a random or semi-random number that is used to ensure a communication is unique and cannot be reused in a replay attack.

On-Chain vs. Off-Chain:

"On-chain" refers to data or logic that exists directly on the blockchain, ensuring maximum security and permanence. "Off-chain" refers to components that exist outside the blockchain, such as a website's user interface or a traditional database.

Provenance:

The documented history of an asset's origin, custody, and ownership. This protocol creates an immutable, on-chain record of a product's provenance.

PUF (Physically Unclonable Function):

A physical microchip with a unique "fingerprint" derived from its physical microstructure, making it impossible to clone. In this project, it is used to generate a secure, one-time signature to prove a product's physical authenticity.

RBAC (Role-Based Access Control):

A security model that restricts system access to authorized users based on their assigned roles. This project uses ADMIN_ROLE, BRAND_ROLE, and RETAILER_ROLE to manage permissions.

Smart Contract:

A self-executing program that runs on the blockchain. It automatically enforces the rules and agreements of a protocol without the need for a central intermediary.

Solidity:

The primary programming language used for writing smart contracts on the Ethereum blockchain.

Wallet:

A digital application (e.g., MetaMask) that allows users to store and manage their digital assets and interact with decentralized applications.

Appendix B: Smart Contract API Reference

This appendix provides a detailed reference for the public and external functions of the Provenance Protocol. The API is distributed across several modular facets. All interactions with the protocol should be directed to the main Provenance (Diamond Proxy) address.

B.1 AdminFacet

```
function registerBrand(
      address _brandAddress,
      string memory _name,
      string memory _website
   ) external
```

- **Description:** Onboards a new, vetted company onto the platform.
- **Access Control:** Can only be called by an address with the ADMIN_ROLE.
- **Parameters:**
  - **_brandAddress (address):** The wallet address of the new brand.
  - **_name (string):** The official name of the brand.
  - **_website (string):** The official website of the brand.

```
function updateBrandStatus(
      address _brandAddress,
      BrandStatus _newStatus
   ) external
```

- **Description:** Updates the operational status of a registered brand (e.g., to Active or Suspended).
- **Access Control:** Can only be called by an address with the ADMIN_ROLE.
- **Parameters:**
  - **_brandAddress (address):** The address of the brand to update.
  - **_newStatus (enum BrandStatus):** The new status for the brand.

```
function getBrandProfile(
      address _brandAddress
   ) external view returns (BrandProfile memory brandProfile)
```

- **Description:** Retrieves the profile data for a registered brand.
- **Access Control:** Public read-only function.
- **Parameters:**
  - **_brandAddress (address):** The address of the brand to query.
- **Returns:**
  - **brandProfile (BrandProfile struct):** A BrandProfile struct containing the brand's on-chain data.

B.2 BrandFacet

```
function mintProduct(
      string memory _name,
      address _productAuth
   ) external
```

- **Description:** Creates (mints) a new product NFT.
- **Access Control:** Can only be called by an address with the BRAND_ROLE and an Active status. It creates the product's on-chain data, including its unique authenticator address, and mints the ERC721 token to the brand itself.
- **Parameters:**
  - **_name (string):** The official name of the product being created.
  - **_productAuth (address):** The unique, product-specific address used for future signature verifications.

```
function registerRetailer(
      address _retailerAddress,
      string memory _name
   ) external
```

- **Description:** Allows a brand to register a new authorized retailer.
- **Access Control:** Can only be called by an address with the BRAND_ROLE and an Active status. Creates a RetailerProfile, sets its status to Active, and grants it the RETAILER_ROLE.

- **Parameters:**
  - **_retailerAddress (address):** The wallet address of the new retailer.
  - **_name (string):** The official name of the new retailer.

```
function updateRetailerStatus(
     address _retailerAddress,
      RetailerStatus _newStatus
   ) external
```

- **Description:** Allows a brand to update the status of one of its authorized retailers.
- **Access Control:** Can only be called by the brand that registered the retailer. If the new status is 'Terminated', the retailer's role is also permanently revoked.
- **Parameters:**
  - **_retailerAddress (address):** The wallet address of the retailer to update.
  - **_newStatus (enum RetailerStatus):** The new operational status for the retailer (Active, Suspended, or Terminated).

```
function initiateShipment(
      uint256[] memory _tokenIds,
      address _retailer
   ) external
```

- **Description:** Initiates the shipment of a batch of products to a retailer.
- **Access Control:** Can only be called by an active brand. Verifies the recipient retailer is also active. Loops through an array of tokenIds, transfers ownership of each, and updates its status to InTransit.
- **Parameters:**
  - **_tokenIds (uint256[]):** An array of token IDs to be shipped.
  - **_retailer (address):** The address of the recipient retailer.

```
function confirmReturnReceipt(
      uint256[] memory _tokenIds
   ) external
```

- **Description:** Allows a brand to confirm the receipt of a returned shipment.
- **Access Control:** Can only be called by an active brand. Updates status from InTransit back to InFactory.
- **Parameters:**
  - **_tokenIds (uint256[]):** An array of token IDs that have been returned.

```
function fulfillDirectOrder(
      uint256 _tokenId,
      address _consumer
   ) external
```

- **Description:** Fulfills a direct-to-consumer sale, bypassing the retail channel.
- **Access Control:** Can only be called by an active brand for a product that is currently in the factory. Transfers ownership and updates the product's status to Sold.
- **Parameters:**
  - **_tokenId (uint256):** The ID of the token being sold directly.
  - **_consumer (address):** The wallet address of the purchasing consumer.

```
function getRetailerProfile(address _retailerAddress) external view returns
(RetailerProfile memory retailerProfile)
```

- **Description:** Retrieves the profile data for a registered retailer.
- **Access Control:** Public read-only function.
- **Parameters:**
  - **_retailerAddress (address):** The address of the retailer to query.
- **Returns:**
  - **retailerProfile (RetailerProfile struct):** A RetailerProfile struct containing the retailer's data.

B.3 RetailerFacet

```
function receiveProductShipment(
      uint256[] memory _tokenIds
   ) external
```

- **Description:** Allows a retailer to confirm the receipt of a shipment of products.
- **Access Control:** Can only be called by an active retailer. It loops through an array of tokenIds, verifying that the caller is the owner and the status is InTransit for each, before updating the status to InRetailer.
- **Parameters:**
  - **_tokenIds (uint256[]):** An array of token IDs that have been physically received.

```
function returnProducts(
      uint256[] memory _tokenIds
   ) external
```

- **Description:** Allows a retailer to return a batch of unsold products to their parent brand.
- **Access Control:** Can only be called by an active retailer. Transfers ownership back to the brand and sets status to InTransit.
- **Parameters:**
  - **_tokenIds (uint256[]):** An array of token IDs to be returned.

```
function finalizeSale(
      uint256 _tokenId,
      address _consumer
   ) external
```

- **Description:** Finalizes the sale of a single product to a consumer.
- **Access Control:** Can only be called by an active retailer who is the current owner of the product. Transfers ownership to the consumer and updates the product's status to Sold.
- **Parameters:**

- **_tokenId (uint256):** The ID of the token being sold. _consumer (address): The wallet address of the purchasing consumer.

B.4 ConsumerFacet

```
function consumeVerification(
      uint256 _tokenId,
      uint256 _challenge,
      bytes memory _signature
   ) external
```

- **Description:** Confirms a product's authenticity by verifying a signature from its physical authenticator (PUF).
- **Access Control:** Public. Can be called by any user. Verifies that the _signature for a _tokenId and _challenge was created by the product's registered authenticator. Consumes the challenge as a nonce to prevent replay attacks. Emits a ProductVerified event on success.
- **Parameters:**
  - **_tokenId (uint256):** The unique ID of the product being verified.
  - **_challenge (uint256):** The unique, one-time number provided to the PUF to generate the signature.
  - **_signature (bytes):** The cryptographic signature produced by the PUF.

```
function getProductDetails(
      uint256 _tokenId
   ) external view returns (ProductNFT memory)
```

- **Description:** Returns the on-chain data for a specific product.
- **Access Control:** Public read-only function.
- **Parameters:**
  - **_tokenId (uint256):** The ID of the token to query.
  - Returns:
  - **(ProductNFT struct):** A ProductNFT struct containing the product's metadata.

```
function isNonceAlreadyUsed(
      uint256 _tokenId,
      uint256 _nonce
```

```
) public view returns (bool)
```

- **Description:** Checks if a specific nonce has already been used for a given token ID.
- **Access Control:** Public read-only function to allow off-chain applications to pre-check a nonce's validity before sending a state-changing transaction.
- **Parameters:**
  - **_tokenId (uint256):** The ID of the token to check.
  - **_nonce (uint256):** The nonce to check.
- **Returns:**
  - **(bool):** True if the nonce has already been used, false otherwise.

B.5 ERC721Facet

```
function balanceOf(address owner) external view returns (uint256) {
      return LibERC721.balanceOf(owner);
   }
```

- **Description:** Returns the number of tokens owned by a given address.
- **Access Control:** Public read-only function.
- **Parameters:**
  - **owner (address):** The address to query the balance of.
- **Returns:**
  - **(uint256):** The number of tokens owned by the address.

```
function ownerOf(uint256 tokenId) external view returns (address) {
      return LibERC721.ownerOf(tokenId);
   }
```

- **Description:** Returns the owner of a given token ID.
- **Access Control:** Public read-only function.
- **Parameters:**
  - **tokenId (uint256):** The token ID to query the owner of.

- **Returns:**
  - **(address):** The address of the owner of the token.

```
function name() external view returns (string memory) {
      return LibERC721.name();
   }
```

- **Description:** Returns the name of the token collection.
- **Access Control:** Public read-only function.
- **Returns:**
  - **(string):** The name of the token collection ("Provenance Digital Twin").

```
function symbol() external view returns (string memory) {
      return LibERC721.symbol();
   }
```

- **Description:** Returns the symbol of the token collection.
- **Access Control:** Public read-only function.
- **Returns:**
  - **(string):** The symbol of the token collection ("PROV").

B.6 AccessControlFacet

```
function hasRole(bytes32 role, address account) external view returns (bool)
```

- **Description:** Returns true if account has been granted role.
- **Access Control:** Public read-only function.
- **Parameters:**
  - **role (bytes32):** The role identifier to check. account (address): The account address to check.
- **Returns:**
  - **(bool):** True if the account has the role, false otherwise.

```
function getRoleAdmin(bytes32 role) external view returns (bytes32)
```

- **Description:** Returns the admin role that controls role.
- **Access Control:** Public read-only function.
- **Parameters:**
  - **role (bytes32):** The role identifier to query.
- **Returns:**
  - **(bytes32):** The admin role identifier for the given role.

```
function grantRole(bytes32 role, address account) external
```

- **Description:** Grants a role to an account.
- **Access Control:** The caller (msg.sender) must have the admin role for the role being granted. This function is a public-facing wrapper that securely delegates its logic to the internal LibAccessControl library.
- **Parameters:**
  - role (bytes32): The bytes32 identifier for the role to be granted.
  - account (address): The address of the account to receive the role.

```
function revokeRole(bytes32 role, address account) external
```

- **Description:** Revokes a role from an account.
- **Access Control:** The caller (msg.sender) must have the admin role for the role being revoked. This function is a public-facing wrapper that securely delegates its logic to the internal LibAccessControl library.
- **Parameters:**
  - **role (bytes32):** The bytes32 identifier for the role to be revoked. account (address): The address of the account from which to revoke the role.

B.7 Provenance.sol (Diamond Proxy)

```
function diamondCut(
        FacetCut[] calldata _diamondCut,
        address _init,
```

```
        bytes calldata _calldata
    ) external
```

- **Description:** Adds, replaces, or removes functions in the Diamond.
- **Access Control:** This is the master control function for all future upgrades. It can only be called by the contract owner. It allows for the modification of the function-to-facet mapping and can optionally run an initializer function as part of an upgrade.
- **Parameters:**
  - **_diamondCut (FacetCut[]):** An array of FacetCut structs detailing the upgrade actions.
  - **_init (address):** The address of a contract to call for initialization after the cut. Use address(0) if none.
  - **_calldata (bytes):** The function call data to send to the _init contract. Use empty bytes if none.

B.8 DemoRoleFaucet.sol

```
function requestAdminRoleRole() external
```

- **Description:** Allows any user to request the ADMIN_ROLE for themselves.
- **Access Control:** Public function for demonstration purposes. Also revokes any other roles the user might have for a clean demo state.

```
function requestBrandRole() external
```

- **Description:** Allows any user to request the BRAND_ROLE for themselves.
- **Access Control:** Public function for demonstration purposes. Also revokes any other roles the user might have for a clean demo state.

```
function requestRetailerRole() external
```

- **Description:** Allows any user to request the RETAILER_ROLE for themselves.
- **Access Control:** Public function for demonstration purposes. Also grants BRAND_ROLE, as retailers are managed by brands.

```
function revokeAllMyRoles() external
```

- **Description:** Allows a user to revoke any special roles to become a default user.
- **Access Control:** Public function for demonstration purposes.

B.9 Data Structures

```
enum BrandStatus {Pending, Active, Suspended, Revoked}
struct BrandProfile {
    address brandAddress;
    string name;
    string website;
    uint256 registrationTimestamp;
    BrandStatus status;
}
```

```
enum RetailerStatus {Active, Suspended, Terminated}
struct RetailerProfile {
    string name;
    address brandAddress;
    uint256 onboardingTimestamp;
    RetailerStatus status;
}
```

```
enum ProductStatus {InFactory, InTransit, InRetailer, Sold}
struct ProductNFT {
    string name;
    address brandAddress;
    address productAuthenticator;
    uint256 mintTimestamp;
    uint256 saleTimestamp;
    ProductStatus status;
}
```

Appendix C: Testing & Verification

The following report was generated by the solidity-coverage tool after the successful execution of the complete 69-test suite. The results demonstrate a rigorous and comprehensive testing strategy focused on the core business logic of the protocol.

```
  69 passing (1m)

| File Path 📦                              | Line % 📈 | Statement % 📈 | Uncovered Lines 🔍          | Partially Covered Lines 🔍                                                  |
| ---------------------------------------- | -------- | ------------- | -------------------------- | -------------------------------------------------------------------------- |
| contracts\libraries\LibAppStorage.sol    | 100.00   | 100.00        | -                          | -                                                                          |
| contracts\libraries\LibDiamond.sol       | 95.00    | 50.00         | 37, 45                     | 73-85, 89-92                                                               |
| contracts\Provenance.sol                 | 91.89    | 84.21         | 53, 95-96                  | -                                                                          |
| contracts\facets\AccessControlFacet.sol  | 87.50    | 57.14         | 22                         | 33-35                                                                      |
| contracts\facets\AdminFacet.sol          | 100.00   | 93.10         | -                          | 89-91                                                                      |
| contracts\facets\BrandFacet.sol          | 100.00   | 98.20         | -                          | 244-246                                                                    |
| contracts\facets\ERC721Facet.sol         | 25.00    | 25.00         | 17, 25, 29                 | -                                                                          |
| contracts\facets\RetailerFacet.sol       | 100.00   | 96.72         | -                          | 111-113                                                                    |
| contracts\facets\ConsumerFacet.sol       | 100.00   | 90.91         | -                          | 45-47                                                                      |
| contracts\utils\DemoRoleFaucet.sol       | 100.00   | 76.47         | -                          | 74-79                                                                      |
| contracts\libraries\LibAccessControl.sol | 80.00    | 72.00         | 84-88                      | 33-35                                                                      |
| contracts\libraries\LibProvenance.sol    | 100.00   | 100.00        | -                          | -                                                                          |
| contracts\libraries\LibERC721.sol        | 87.36    | 56.58         | 23-27, 44-45, 52-53, 63-64 | 24-26, 87-89, 91-93, 104-106, 108-110, 112-114, 157-159, 209-218, 230-232 |
| ---------------------------------------- | -------- | ------------- | -------------------------- | -------------------------------------------------------------------------- |
| Total                                    | 95.19    | 80.91         |                            |                                                                            |
```

Figure 9: solidity-coverage report

Analysis of Coverage Results

The report confirms an overall line coverage of **95.19%** and a statement/branch coverage of **80.91%**. More importantly, it shows that all core **Application Logic Facets** achieved their primary testing objectives:

- **AdminFacet, BrandFacet, RetailerFacet, and ConsumerFacet all achieved 100% line coverage.** This is a critical result, as it proves that every single line of code responsible for the protocol's business rules, state transitions, and security checks was executed by the test suite.
- The high statement/branch coverage scores *(e.g., BrandFacet at 98.20%)* indicate that the tests successfully validated nearly every logical path, including both "happy path" scenarios and all critical failure conditions (revert statements).

Analysis of Intentionally Uncovered Code

The remaining gaps in coverage are intentional and located in non-essential framework code that falls outside the core focus of the project's business logic. This reflects a strategic decision to prioritize testing the custom-built application over re-testing standard framework patterns. The primary areas of uncovered code are:

- **LibDiamond.sol:** The test suite only covers the Add action of the diamondCut function, as this is all that is required for the initial deployment. The Replace and Remove branches, which relate to future upgradeability, were not part of this project's testing scope.
- **ERC721Facet.sol & LibERC721.sol:** The tests focus on the functions critical to the provenance lifecycle (mint, ownerOf, balanceOf). The standard approve and setApprovalForAll functions, which relate to token tradability, were intentionally omitted from the public facet and are therefore not tested.
- **AccessControlFacet.sol & LibAccessControl.sol:** While the core grantRole, revokeRole, and hasRole functions are fully tested through their usage in other facets, the administrative setRoleAdmin function was not included in the test suite.

This targeted testing strategy provides the highest degree of confidence in the security and correctness of the protocol's unique and essential features.