Universal Goods Protocol:
Tokenising Physical Products

James Albarracin
james@universalgoods.org
www.universalgoods.org

Felix Hildebrandt
felix.hildebrandt@gmx.net
fhildeb.com

Abstract. The Universal Goods Protocol is a method of tokenising physical products on the Ethereum Virtual Machine (EVM). Each product line is a token collection owned by the issuing smart-contract account. Each physical item is one token in that collection, and its identifier is the hash of the Unique Product Identifier (UPI) printed on the item. The accounts are smart contracts built on the **LUKSO** Standard Proposals (LSPs), where an account's address never changes while its keys are added, replaced or recovered, and a relayer can pay for its transactions. The same contracts therefore serve an enterprise issuing a million items and a person holding one. Anyone who can read the label can compute the token and find its collection on a public endpoint. From there they can read who issued the item, who holds it now, and whether the passport still matches the hash the issuer recorded.

1. Introduction

Counterfeiting and piracy persist because a physical product has no identity that anyone outside its maker can verify. A copy can potentially carry every mark the original carries, an outsider cannot say with certainty who issued an item, and a claim to own it remains only a claim. What would settle these questions is a public record of who issued an item and who holds it. We propose a protocol in which physical items are represented by a token, minted in a collection that the manufacturer's own account controls. The token's identifier is the hash of the identifier printed on the item, so who issued the item and who holds it become public facts that a stranger can read from the ledger with one call. Every issuance and every transfer is recorded on that ledger as a signed and timestamped transaction. Neither the issuer nor any other party can alter the record afterwards without the alteration being evident.

The obstacle to recording physical products on a public ledger has not been the token contract but the account required to use it. A key-based account is lost with its key, pays its own fees in the network's currency, has no permissions, and shows the person signing a hexadecimal string. That is not a tool an enterprise can hand to its staff or a normal user can be asked to keep. Recording physical products therefore requires a transition from key-based accounts to smart-contract accounts. Such an account keeps a permanent address while the keys that control it are added, restricted and removed. Its transactions can be submitted and paid for by another party, and it is also notified when an asset arrives and can accept or refuse it. LSPs specify an account with these properties called the **Universal Profile**.

2. Digital ownership

A blockchain is a public ledger that many independent parties maintain and that none of them can rewrite. Programs stored on such a ledger are called smart contracts, and their rules execute identically for everyone who calls them. A non-fungible token is a record in a smart contract that assigns one unique identifier to one holder. Together they make it possible, for the first time, to own an asset digitally in a way anyone can verify. Applied to a physical item, this provides three properties that a copied item cannot reproduce:

1) **One identifier, one token.** The token's identifier is the hash of the identifier printed on the item (section 4). Each printed identifier therefore names exactly one token in a collection, and a second mint of it reverts.
2) **Issuance is attributable.** Only a collection's owner can mint into it, so every token carries the account that issued it.
3) **Ownership is public.** Anyone can read the current holder of a token with a single call to `tokenOwnerOf(tokenId)`. The token's history from mint to present holder is the sequence of `Transfer` events emitted by the collection, each fixed in a block with a timestamp.

A token moves only when its holder offers it and the receiver accepts it. The holder calls `authorizeOperator(receiver, tokenId, upi)`, which notifies the receiver's account and hands it the item's identifier. The receiver then calls `transfer(holder, receiver, tokenId, force, upi)` to accept the token. Both calls revert unless `keccak256(upi)` equals the token identifier, so every movement names the object it moves and records both parties on the ledger permanently. Ownership of this kind is non-custodial by construction: the issuer holds nothing for anyone else, and a person who accepts a token holds it in an account only their own keys control. Beyond this two-step transfer, such ownership requires an account that organisations and individuals can operate in practice. The account must survive a lost key, be usable by staff and customers without the network's currency and be able to refuse what it does not want.

3. Smart contracts

The solution we propose begins with the latest smart-contract standards, the **LUKSO** Standard Proposals [3], which replace the key-based account with a smart-contract account called a **Universal Profile**. To these standards the protocol adds a single contract of its own: the collection contract, which holds the tokens of one product line.

3.1 The account

The pieces, in the order a transaction meets them:

- **LSP0 ERC725Account** is the account itself: a contract with an `execute` function (ERC725X) through which it calls any other contract, and a key-value store (ERC725Y) in which it keeps its data under 32-byte keys. It also implements ERC1271, through which other contracts verify that a signature is valid for the account.
- **LSP6 Key Manager** owns the account and decides what each controller may do. Permissions are granular: a controller can be limited to particular actions, contracts, functions and data keys. They

are stored in the account's own storage per controller address. Keys can therefore be added, rotated, lost and recovered, which a key-based account cannot survive.

- **LSP1 Universal Receiver** is called whenever an asset arrives or an offer is made, with the data the sender attached. The account can record the asset (the standard delegate lists received assets under LSP5), react to it or refuse it.
- **LSP25 Execute Relay Call** lets anyone submit a transaction the controller has already signed and pay its fee. The relayer cannot change what was signed. A user who accepts an item therefore never needs to hold the network's currency.
- **LSP2, LSP3, LSP4 and LSP12** define how data is laid out in ERC725Y storage, the account's profile (such as its name, description, links and images), a token contract's metadata, and the list of collections an account has issued.
- **LSP8 Identifiable Digital Asset** is the token standard for assets that are each distinct, identified by a 32-byte token identifier, with per-token storage. The protocol's collection is an LSP8 contract (3.2).

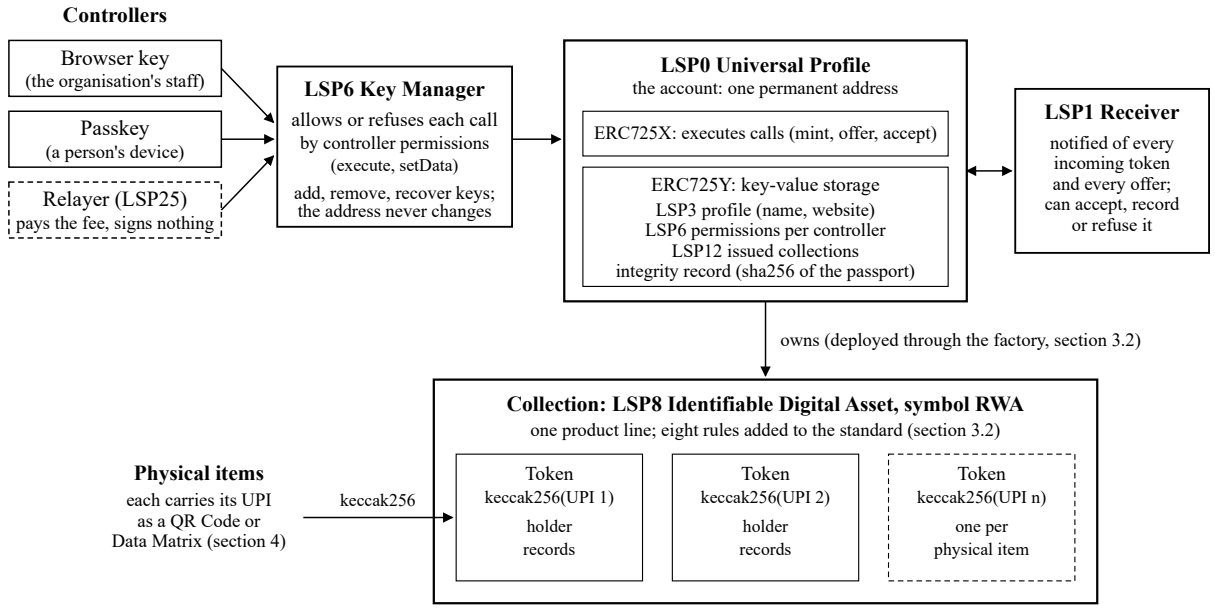


Figure 1. Smart-contract architecture.

3.2 The collection contract

A collection is **LUKSO**'s `LSP8Mintable` (`@lukso/lsp8-contracts` 0.16.7) with eight added rules. At construction it refuses a zero owner and fixes four values that never change:

Data key	Key (hex)	Value
`LSP4TokenSymbol`	`0x2f0a68ab07768e01943a599e73362a0e17a63a72e94dd2e384d2c1d4db932756`	the three bytes `RWA`
`LSP4TokenType`	`0xe0261fa95db2eb3b5439bd033cda66d56b96f92f243a8228fd87550ed7bdfdb3`	`2` (collection)
`LSP8TokenIdFormat`	`0xf675e9361af1c1664c1868cfa3eb97672d6b1a513aa5b81dec34c9ee330e818d`	`4` (hash)

Data key	Key (hex)	Value
`LSP4TokenName`	`0xdeba1e292f8ba88238e10ab3c7f88bd4be4fac56cad5194b6ecceaf653468af1`	the product line's name

The symbol `RWA` (real-world asset) marks the collection as standing for physical objects; it is a category anyone may use, not a credential. The token type declares many distinct items, and the token-ID format declares that identifiers are hashes, with section 4 fixing what they are hashes of. A fifth key, `LSP4Metadata`, is written by the owner after deployment as a `VerifiableURI`: the hash of a JSON document describing the product line (such as its name, description, image and Global Trade Item Number, GTIN) together with the document's URL, so the collection commits to a description of the product line that anyone can fetch and check.

#	Rule	Reverts with
R1	Only the collection owner may mint	the owner check
R2	A mint's recipient must be the collection owner	`UG_MintRecipientMustBeOwner(address,address)`
R3	The token identifier `keccak256("")` may not be minted	`UG_TokenIdMustNotBeTheEmptyUpiHash()`
R4	Ownership cannot be renounced	`UG_OwnershipIsNotRenounceable()`
R5	The symbol is exactly `RWA`, enforced at construction	`UG_SymbolMustBeExactlyRwa(string)`
R6	`transfer` reverts unless `keccak256(data) == tokenId`, and reverts when the caller is the token's holder	`UG_UpiDoesNotMatchTokenId(bytes32,bytes32)`, `UG_DirectTransferRefused_OfferWithAuthorizeOperator(bytes32)`
R7	`authorizeOperator` reverts unless `keccak256(operatorNotificationData) == tokenId`	`UG_UpiDoesNotMatchTokenId(bytes32,bytes32)`
R8	A passport event may be appended only by the owner or an authorised appender, and must carry a non-zero content hash	`UG_NotAPassportEventAppender(address,bytes32)`, `UG_PassportEventContentHashRequired(bytes32)`

The rules in the reference contract, `RealWorldAssetsV3.sol` [7], with comments removed. Everything not shown is the unmodified **LUKSO** base contract, `LSP8Mintable`.

```
contract RealWorldAssetsV3 is LSP8Mintable {
    bytes32 internal constant _EMPTY_UPI_TOKEN_ID = keccak256("");

    constructor(string memory name_, string memory symbol_, address owner_)
        LSP8Mintable(name_, symbol_, owner_,
                     _LSP4_TOKEN_TYPE_COLLECTION, _LSP8_TOKENID_FORMAT_HASH)
    {
        requireRwaSymbol(symbol_);                                  // R5
    }

    function mint(address to, bytes32 tokenId, bool force, bytes memory data)
        public virtual override onlyOwner                           // R1
    {
        address collectionOwner = owner();
        if (to != collectionOwner)                                  // R2
            revert UG_MintRecipientMustBeOwner(to, collectionOwner);
        if (tokenId == _EMPTY_UPI_TOKEN_ID)                         // R3
            revert UG_TokenIdMustNotBeTheEmptyUpiHash();
        _mint(to, tokenId, force, data);
    }

    function transfer(address from, address to, bytes32 tokenId, bool force, bytes memory data)
        public virtual override
    {
        _requireUpi(tokenId, data);                                 // R6, first half
        if (msg.sender == tokenOwnerOf(tokenId)) {
```

```
            revert UG_DirectTransferRefused_OfferWithAuthorizeOperator(tokenId);   // R6
        }
        super.transfer(from, to, tokenId, force, data);
    }

    function authorizeOperator(address operator, bytes32 tokenId, bytes memory data)
        public virtual override
    {
        _requireUpi(tokenId, data);                                  // R7
        super.authorizeOperator(operator, tokenId, data);
    }

    function _requireUpi(bytes32 tokenId, bytes memory upi) internal pure {
        bytes32 suppliedUpiHash = keccak256(upi);
        if (suppliedUpiHash != tokenId)
            revert UG_UpiDoesNotMatchTokenId(tokenId, suppliedUpiHash);
    }

    function renounceOwnership() public virtual override {
        revert UG_OwnershipIsNotRenounceable();                      // R4
    }
}
```

Collections are deployed through a factory with CREATE2, so an address is fixed by its contents:

```
initCode = creationCode(RealWorldAssetsV3) || abi.encode(name, symbol, owner)
address  = keccak256( 0xff || factory || salt || keccak256(initCode) )[12:]
```

If code already exists at the predicted address the factory returns it rather than reverting: a front-runner can only place the issuer's own collection there, at the front-runner's expense. After deploying, the factory reads the owner back and reverts on a mismatch. The collection bytecode is embedded in the factory, so the predicted address and the deployed contract come from one build.

Chain	Chain ID	Factory
LUKSO mainnet	42	`0x2787f9a97A1C6CA0D201D3B6333208F1a749E8BC`
LUKSO test network	4201	`0x11d51E1253f3962eDDf09A3D3dc6c29E533E0FDE`

3.3 Issuing a product line

The steps to issue a product line are as follows:

1) The issuer's profile calls the factory with the product line's name, `RWA`, its own address and a salt.
2) The issuer writes `LSP4Metadata` on the collection and lists the collection on its profile under `LSP12IssuedAssets[]`.
3) For each item the issuer forms its UPI (section 4), computes the token identifier and mints it to itself, in batches sized to the chain's gas limit.
4) The issuer publishes the passport at each UPI and records its integrity hash on its profile (section 5).

The measured cost of each operation in gas:

Operation	Gas
Create a collection	2,576,608
Mint 1 item	299,665 to 316,914
Mint 136 items (the largest batch under the 16,777,216 cap)	14,198,861 to 14,198,993
Offer one item (`authorizeOperator`)	168,242

Operation	Gas
Offer / accept one item	165,052 / 335,741
Record a passport integrity hash	108,621

Besides its holder, each token has an append-only store of 32-byte commitments, governed by rule R8 and laid out as an LSP2 array under the token:

Item	Value
Array key `UGPassportEvents[]`	`keccak256("UGPassportEvents[]")` = `0x76fe6ac3522f2ed08afe290a3c815d317eb87fc7436f7de6526e80b7fb32ea58`
Length	per token under the array key, a 16-byte big-endian `uint128`
Element key at index `i`	`bytes16(arrayKey) || bytes16(uint128(i))`, `i` from 0 (LSP2 array layout)
Element value	`abi.encode(bytes32 contentHash, address author)`, 64 bytes
Scope	per token, through LSP8 `setDataForTokenId` / `getDataForTokenId`

```
function appendPassportEvent(bytes32 tokenId, bytes32 contentHash)
    external virtual returns (uint128 index)
{
    _existsOrError(tokenId);
    if (!canAppendPassportEvent(msg.sender, tokenId))                // R8
        revert UG_NotAPassportEventAppender(msg.sender, tokenId);
    if (contentHash == bytes32(0))
        revert UG_PassportEventContentHashRequired(tokenId);

    index = passportEventCount(tokenId);
    _setDataForTokenId(
        tokenId,
        LSP2Utils.generateArrayElementKeyAtIndex(UG_PASSPORT_EVENTS_ARRAY_KEY, index),
        abi.encode(contentHash, msg.sender)              // 64 bytes: hash, then author
    );
    _setDataForTokenId(tokenId, UG_PASSPORT_EVENTS_ARRAY_KEY, abi.encodePacked(index + 1));
    emit PassportEventAppended(tokenId, msg.sender, index, contentHash);
}
```

Only the collection owner and the parties it has authorised can append an entry. Each entry stores the address of the party that wrote it, so every statement remains attributable to its author. The owner retains the standard write functions and could overwrite an entry, but such a write does not emit `PassportEventAppended`, so any overwrite is evident from the contract's event log, which is stored permanently on the ledger.

4. Unique Product Identifier

GS1 is the standards organisation that administers product identifiers and barcodes [2]. The UPI is an absolute GS1 Digital Link URL with an item-level serial:

`https://{host}/01/{gtin}/21/{serial}`

It is what is printed on the item and what a user scans to access the digital passport. A UPI uses `https` only; has a lowercase host of at least two labels with no port and no user information; carries a GTIN of 8, 12, 13 or 14 digits under application identifier `01`; carries a serial of 1 to 20 characters from the unreserved character set of the URI syntax standard RFC 3986 [4] under `21`; and carries no lot, query or fragment. The serial set is narrower than GS1 permits on purpose: a conforming UPI is identical before and after any percent-encoder touches it, so its bytes and its hash are reproducible by an implementation

that has never spoken to us. The reference implementation issues each serial as ten random uppercase base-36 characters drawn from a cryptographic source. A fixed width keeps the 2D symbol one size through a production run, and a random value does not reveal how many items were produced.

The token identifier is the Keccak-256 hash of the UPI's UTF-8 bytes, exactly as printed:

```
tokenId = keccak256( UTF-8 bytes of the UPI, exactly as printed )
```

No case folding, no trailing-slash change, no trimming, and no percent-encoding applied or removed; an identifier that does not match the grammar is refused, not repaired. Character case is significant, so `/21/00000A` and `/21/00000a` identify different tokens. The reference implementation [7]:

```
import { keccak256, toUtf8Bytes } from 'ethers';

const AI_VALUE = '[A-Za-z0-9\\-._~]{1,20}';
export const UPI_TOKENID_GRAMMAR = new RegExp(
  '^https://' +
    '(?![0-9.]+/)' +                                  // no all-numeric or IPv4 host
    '[a-z0-9](?:[a-z0-9-]*[a-z0-9])?' +               // host: lowercase labels, no port
    '(?:\\.[a-z0-9](?:[a-z0-9-]*[a-z0-9])?)+' +
    '/01/(?:\\d{8}|\\d{12}|\\d{13}|\\d{14})' +        // GTIN-8, UPC-A, EAN-13, GTIN-14
    `/21/${AI_VALUE}$`,                               // the serial, required
);

export function itemTokenId(upi: string): `0x${string}` {
  if (!upi) throw new Error('empty unique product identifier');
  if (upi.includes('#') || upi.includes('?')) throw new Error('fragment or query refused');
  if (!UPI_TOKENID_GRAMMAR.test(upi)) throw new Error('not an item-level identifier');
  return keccak256(toUtf8Bytes(upi)) as `0x${string}`;
}
```

The token identifier is derived from the UPI by a one-way hash. The UPI cannot be computed from the token identifier, only confirmed by hashing a candidate UPI. A reader computes the token identifier from the label alone, without a lookup in any registry or database, and then asks the ledger which collection minted it.

The UPI is carried on the item as a QR Code or Data Matrix printed or engraved on the item or its label, or written to an NFC or RAIN RFID tag, with 2D print quality graded against ISO/IEC 15415. The carrier only locates the record and does not attest to it. The host named in the UPI serves the item's public passport as JSON by default and as HTML to a browser. The JSON carries the chain ID of the EVM network, the collection address and the token identifier. Together these locate the ledger entry without a second lookup. The host is part of the hashed bytes. Any change to it produces a different token identifier, so an issuer must treat the printed host as permanent for the life of the item.

The UPI is publicly visible on the item and can be scanned by anyone. Serials are drawn at random, which prevents an observer from inferring production volume or deriving one item's identifier from another's. The integrity of the record does not depend on the identifier being hidden. The identifier is a pointer to the record, and the record is protected by the rules of the collection contract. A copied UPI on a counterfeit resolves to the genuine item's record, whose token lives in the brand's collection and is held by whoever the ledger says. A counterfeiter cannot mint a second token for the same identifier in the brand's collection. Only the collection owner can mint, and a mint of an existing token identifier reverts. The counterfeiter cannot change the holder, because only the holder can offer the token. Nor can the counterfeiter write to the record, because only the collection owner or a party it authorises can. What makes an item authentic in the protocol's sense is therefore not the identifier but the ownership and provenance behind it.

5. Verification

A third party cannot be asked to trust a company's database about that company's products, and a seller cannot be allowed to choose the verifier. Verification under the protocol therefore reads the decentralised public ledger through an endpoint the reader chose, starting from nothing but the printed identifier. The reader relies on three assumptions: the integrity of the ledger, the collision resistance of the hash functions, and the premise that a transaction signed by the issuer's account was authorised by the issuer. The steps are as follows:

1) **Obtain the identifier** exactly as printed or encoded: same case, no added or removed slash, no query, no fragment.
2) **Derive the token identifier**, `keccak256` of its bytes (section 4).
3) **Locate the mint.** Query a public endpoint for the LSP8 `Transfer` event whose `from` is the zero address and whose token identifier is the one derived in step 2. The emitting address is the collection, and no such event on any collection means no record.
4) **Read the issuer.** The collection's `owner()` names the issuer account. A contract can report any owner, so the reader confirms the claim from the account's side: the account lists the collections it has issued under `LSP12IssuedAssets[]` (section 3.3), and a collection it does not list was not issued by it. The account's profile names its website (LSP3), which a reader compares with the brand's site it already knows. The profile is written by the account itself, so that comparison is conclusive only when the brand's own site names the account in return.
5) **Retrieve the holder.** `tokenOwnerOf(tokenId)` on the collection.

The same steps as JSON-RPC calls against any mainnet endpoint are given below. The first constant is the LSP8 `Transfer` event topic:

```
const TRANSFER_TOPIC = '0xb333c813a7426a7a11e2b190cad52c44119421594b47f6f32ace6d8c7207b2bf';
const tokenId = itemTokenId(upi);                                    // section 4

// 3. the mint: which collection(s) minted this token ID
const logs = await rpc('eth_getLogs', [{
  fromBlock: '0x0', toBlock: 'latest',
  // topics: event, from = the zero address, to (any), tokenId
  topics: [TRANSFER_TOPIC, ZERO_ADDRESS_TOPIC, null, tokenId],
}]);
const collections = [...new Set(logs.map((l) => l.address))];     // more than one is possible

// 4 and 5. for each candidate: issuer (owner) and holder
for (const collection of collections) {
  const issuer = decodeAddress(await rpc('eth_call',
    [{ to: collection, data: selector('owner()') }, 'latest']));
  const holder = decodeAddress(await rpc('eth_call',
    [{ to: collection, data: selector('tokenOwnerOf(bytes32)') + tokenId.slice(2) },
     'latest']));
  // the issuer's own profile must list this collection (LSP12IssuedAssets[])
  const listed = await isIssuedAsset(issuer, collection);
  report({
    collection, issuer, holder, listed,
    holderIsIssuer: sameAddress(holder, issuer),
  });
}
```

The procedure above establishes the issuer of a token and its current holder. A second procedure verifies that the content of the passport has remained unchanged since the issuer committed to it. The passport is issued as an SD-JWT verifiable credential signed by a key that the organisation holds in its own browser

(ES256K, issuer `did:pkh:eip155:{chainId}:{profile}`). The issuer then writes the following record to its own profile with a standard `setData` call:

```
record key   = keccak256("UG_VerifiableData_product_{productId}")
record value = sha256( UTF-8 bytes of the credential text )
```

A party holding the credential first computes its SHA-256 hash and compares it with the value that `getData(recordKey)` returns from the issuing account. It then compares the passport data served by the resolver with the data sealed in the credential. The second comparison checks whether the issuer has altered the served passport after committing to it. The result is `verified` when both comparisons agree and `invalid` when either fails. It is `unverified` when the issuer has recorded no hash. It is `cannot_verify` when the check could not be completed, for example because the ledger could not be read. The resolver reports the same four outcomes in the `passportCheck` property of the passport JSON, where they are named `verified`, `changed_after_signing`, `not_signed` and `cannot_check`. This property includes the chain, a public RPC endpoint, the account, the record key with its pre-image and the recorded value. Any reader can therefore repeat the check independently. The hash currently covers the entire credential, including its restricted sections. Only a party that holds the full credential can therefore recompute it.

6. Digital interoperability

The protocol introduces as little new specification as possible and relies on established standards, so that a record made under it can be read by systems with no knowledge of the protocol:

Layer	Standard	Role in the protocol
Identifier	GS1 GTIN and GS1 Digital Link; ISO/IEC 15459 [2]	the UPI's form; the GTIN comes from the brand's own GS1 office
Data carrier	ISO/IEC 18004 (QR Code), ISO/IEC 16022 (Data Matrix), ISO/IEC 15415 [2]	how the UPI is carried on the item and how print quality is graded
Hash and ledger	Keccak-256; the Ethereum Virtual Machine	the token identifier; contracts in ordinary Solidity
Account	ERC-725, ERC-1271; LSP0, LSP1, LSP6, LSP25 [3]	the issuer's and holder's accounts, permissions, notification and relayed execution
Token and metadata	LSP8; LSP2, LSP4, LSP12 [3]	the collection, per-token storage, metadata and the issued-assets list
Passport delivery	HTTPS with content negotiation (EN 18216); the EN 18223 passport header [1]; JSON and JSON-LD	how the resolver answers a scan
Content sealing	SD-JWT; JOSE; SHA-256 [5]	the credential and the integrity record

We tested how AI models read and act on these records in pre-registered studies. The plan, amendments, raw data and checksums of each are held in the study repository [6]. Three results are relevant to interoperability:

a) Three models (OpenAI, Kimi and Claude, pooled) were asked by a shopper to choose between two matched batteries on live shop pages. The product that carried a passport was recommended in 81% of trials, against 39% when neither product carried one. In 81% of those answers the stated reasons cited the passport.

b) Against a rival with a fake passport the pooled advantage vanished (39%, the baseline). In 72 trials where a public record existed, none of the three models opened it. In the same study one model chose the fake product in 6 of its 12 trials. In two live-site studies no model followed a link to the ledger itself in 570 trials. The one model that consistently opened the public record (Meta muse-spark, 24 of 24 trials) chose the genuine product in all 12 trials against the fake.

c) In a further study the genuine product again faced the rival with the fake passport. When the first procedure of section 5, which reads the issuer and the holder, was made available to the models as a function they could call (the verify tool at commit `dc97cc6` [8]), the pooled models recommended the genuine product in 89% of trials against 33% without it. No model chose the fake product after the tool reported that no record existed for it.

AI readers already reward a passport, and they reward a convincing fake passport equally when verification takes more than one step. A record on a decentralised public ledger can be checked by any person or machine with one call, without an account or a relationship with the issuer. When that call was available to the models, the fake product lost its advantage. A record in a privately operated database cannot be checked this way without trusting its operator. We expect verification tools to become a standard part of how agents read product pages. Once such checks become common practice, the gap between a passport that can be verified publicly and one that can only be asserted will widen.

7. Incentives

Tokenising a product under the protocol is not only a technical improvement, it changes the commercial position of the manufacturer and the utility of the product. Today a manufacturer's relationship with its product ends at the first sale. The retailer knows the first buyer, and every later owner is a stranger to the brand. Under the protocol every item has a holder of record, and each change of hands is a transfer the holder authorises. The current owner of a specific item can therefore be addressed directly. Warranty, recall, service and resale then follow the item rather than the receipt. The holder controls both the token and the account that holds it. The protocol gives the issuer no right to move the token or to act on that account. The manufacturer gains a verifiable channel to every past and present holder for the whole life of the product.

Because the token is public and built on open standards, it can be read and acted on by third-party systems without any approval from the issuer. A lender can confirm that a borrower holds an item and that its issuer is the stated manufacturer. An insurer can read the item's provenance before writing a policy. A resale platform can list an item against its record and transfer the token with the sale, so that the next owner inherits the record and the channel continues. None of these parties needs an account with the issuer, because the record is on a decentralised public ledger under open standards. The same property that lets an AI reader check a passport (section 6) lets finance, insurance, resale and repair build on it. The outcome is an open ecosystem around the physical product, made feasible by a public blockchain and the new smart-contract standards.

8. Limitations and security

The limitations below apply to any party that issues, holds or reads a token under the protocol, on any deployment.

a) The protocol makes an assertion attributable, tamper-evident and portable. It does not make it true. A false value committed with a valid key remains false, and the protocol records only who committed it and when.
b) A label can be copied onto a counterfeit, swapped between items or removed from an item. A copied label does not duplicate the token (section 4). A reader must therefore check the issuer and the holder of the token (section 5), and not only that the identifier resolves. Presenting the identifier does not prove possession of the item, because anyone who has seen the label can present it.
c) Every transfer permanently records the item's identifier together with the accounts of both parties. An account may constitute personal data when it can be linked to a natural person. For this reason, the protocol writes no passport content to the ledger. Only identifiers, authorisations and hashes are stored there.
d) The host in the UPI is part of the hashed bytes, so it cannot be changed without naming a different token. If the domain registration lapsed or the host became unavailable, labels would stop resolving, though token identifiers would remain computable and tokens readable. An issuer should therefore ensure that the host remains available beyond the life of its products.

9. Conclusion

We have proposed a system for tokenising physical products on the Ethereum Virtual Machine. The system is built on the account standards published by the **LUKSO** team, which abstract the account from its keys and allow transactions to be relayed. Neither a business nor the person holding its product has to acquire the network's currency or safeguard a seed phrase, and no intermediary holds assets on their behalf. Each physical product is represented by a token whose identifier is the hash of the identifier printed on the product. The token is minted in a collection owned by the manufacturer's smart-contract account, and the protocol enforces eight rules that anyone can inspect.

Verifiable digital ownership of a physical item and its recorded provenance are the basis on which the protocol addresses counterfeiting and piracy. A copied label leads to the record of the genuine item, which identifies a holder that a counterfeiter cannot change. Any party can read the record with a single call to a public endpoint, without requiring an account or permission from the manufacturer. Because the record uses open identifiers and open standards on a decentralised public ledger, it can be read by any system that implements those standards. The research and development have shown that a digital, verifiable provenance record can further improve the recommendation and verification of physical products as global commerce becomes agentic, with AI agents choosing products on behalf of buyers.

References

1. EN 18216:2026, EN 18219:2026, EN 18220:2026, EN 18223:2026 (May 2026), cited under the ESPR by Commission Implementing Decision (EU) 2026/1736.
2. GS1 General Specifications; GS1 Digital Link standard; ISO/IEC 15459 series; ISO/IEC 18004:2024 (QR Code); ISO/IEC 16022:2024 (Data Matrix); ISO/IEC 15415:2024.
3. LUKSO Standard Proposals: LSP0 (ERC725 account), LSP1 (universal receiver), LSP2 (ERC725Y JSON schema), LSP3 (profile metadata), LSP4 (digital asset metadata), LSP6 (key manager), LSP8 (identifiable digital asset), LSP12 (issued assets), LSP17 (contract extension), LSP25 (execute relay call); ERC-725; ERC-20 (F. Vogelsteller, V. Buterin, 2015). Universal Profiles, LUKSO technical documentation, https://docs.lukso.tech/standards/accounts/introduction (Apache License 2.0).
4. RFC 3986 (URI generic syntax); RFC 8785 (JSON Canonicalization Scheme).
5. IETF, Selective Disclosure for JWTs (SD-JWT); W3C Verifiable Credentials Data Model; FIPS 180-4 (SHA-256); Keccak (SHA-3 family) as used by the EVM.
6. Family Labs Pty Ltd, Agent DPP Verification Study: pre-registration, amendments, results and raw data with SHA-256 checksums (`PREREGISTRATION.md`, `RESULTS-*.md`, `results-archive/`), https://github.com/family-labs/agent-dpp-verification-study.
7. Family Labs Pty Ltd, Universal Goods Protocol: contracts, packages and documentation, https://github.com/family-labs/universal-goods-protocol (Apache License 2.0).
8. Universal Goods Protocol, Verify a product passport: guide (`docs/docs/verify`) and the `packages/verify` library, command and MCP server, https://github.com/family-labs/universal-goods-protocol/tree/dc97cc6 (commit `dc97cc6`).