Smart Contract Patterns

Suggest Edits

Overview

A core functionality of Verite is that it enables smart contracts to require successful verification of Verifiable Credentials in order to safely and securely assess proofs of identity, credit and risk scoring, insurance, accredited investor status and other identity-related claims.

Today, smart contracts on most chains (such as ethereum) are not technically capable and/or economically practical at executing the verification operations themselves, and they cannot call upon external verification or other network services beyond the constraints of their own chains. Verification in-contract would also require transmitting credentials on-chain, which may leak sensitive personal information and violate the first principle of 'Privacy by Design.'

Instead, verification can be done by a web-based "front-end", or outsourced to a verifier service that communicates to the dApp directly (via offchain transactions delivered to the smart contract along with the transactions it authorizes) or indirectly (via on-chain registries, oracles, or other mechanisms).

All of these architectures have in common a pattern in which verifications -- including credential exchanges between verifiers and subjects, revocation and validation checks, and verifiable credential signature verifications -- are executed "off-chain" and then cryptographically validated by on-chain smart contracts that associate Verification Record s with on-chain addresses. These can live in a Verification Registry on-chain if the registry observes a few lightweight, privacy-preserving best practices. (Note: detailed guidance on this is forthcoming, care of the Verite On-Chain Best Practices Working Group)

This document describes the architecture of this design, illustrates two primary options for implementation and two options for storage, and references example source and tests. The reference implementations are scoped toSolidity , but are intended to illustrate patterns applicable to other chains with similar smart contract programming capability.

Registry-Based Architecture

Smart contracts follow a verification pattern in which verifications are performed off-chain and then confirmed on-chain. An off-chain verifier handles Verifiable Credential exchange in the usual manner and, upon successful verification, the verifier creates a lightweight privacy-preservingVerification Result object (represented in JSON).

The verifier then hashes and signs the Verification Result. The signature, along with the result itself, enables subsequent validation by smart contracts known asVerification Registries .

The verifier either registers the result directly with a Verification Registry contract as part of the verification process (the “Verifier Submission ” pattern), or else returns it to subjects for them to use in smart contract transactions (the “Subject Submission ” pattern). Both sequences are discussed below.

The registry smart contract validates the Verification Result and the verifier’s signature, and creates a privacy-preservingVerification Record that enables subsequent apps, contracts, and tokens to query whether a particular verification is associated with a particular address.

Verification Results

Upon completing successful verification of Verifiable Credentials, verifiers create Verification Results and sign them. Verification Results contain minimal information:

- A schema identifier identifying the type of credential that was verified
- The public on-chain address associated with the verified subject, and
- The expiration time of the result (which may occur sooner than the expiration of the credential)

A Verification Result created by a Verifier is never directly persisted to a blockchain, but it may be (if on-chain persistence is needed by the use case) persisted as a minimized Verification Record . Best practices guidance for minimizing privacy risks when persisting these records are forthcoming.

A Verification Result and the verifier’s signature are passed as parameters to a Verification Registry contract function for validation. The registry uses the signature and the Verification Result to determine whether or not the result corresponds to the public address of a known verifier configured in the registry contract.

A Verification Registry executes this confirmation by accepting the Verification Result and signature as input parameters, checking any relevant contract logic against data in the Verification Result, and then re-hashing the Verification Result and using the hash and the submitted signature to confirm that the signature was generated by the private signing key of a corresponding public address of a known verifier configured in the contract. To prevent replay attacks, the Verification Registry also ensures that the calling address is the verifier (for verifier submissions), or that the calling address is the same as the subject of the VerificationResult (for subject submissions).

In order to safeguard against potential issues with hashing, signatures, and key recovery in Ethereum, the reference implementations utilizeEIP-712 as the preferred hashing and signing approach when evaluating Verification Results. Specifically, the examples employ theOpenZeppelin EIP-712 implementation . The reference implementations also leverage OpenZeppelin's Ownable andERC20 implementations.

Verification Records

Upon successful confirmation, the Verification Registry smart contract generates a Verification Record with basic information about the verification. This is the data model that is emitted as an event by the contract, and the data model that is optionally persisted on-chain.

A Verification Record contains:

- A UUID associated with the verification qua event (Note: we also recommend logging events by the same UUID for cleaner auditing and forensics of verification functions
-)
- Verifier's and subject's public chain addresses
- The time of verification confirmation in the contract
- The expiration time of the verification (which is not necessarily the same as that of the credential), and
- A boolean flag to denote whether the record is revoked (verifiers can revoke records they create).

If the registry is persistent, then this is information that is recorded on-chain about a successful verification. If the registry is not persistent, then the data in a Verification Record is still emitted as an event upon successful validation. No additional information about the underlying credential, issuer, subject, or verifier is contained in the Verification Record.

Verification Records allow counterparties, DeFi pools, Dapps, and tokens to query whether a particular address has had a given claim -- KYC/IDV, accredited investor status, etc. -- verified before or during a transaction, without leaking Personally Identifiable Information (PII) about the queried address. Verification Records also enable audit and forensics sequences, discussed below.

Expiration and Revocation

Verification Records have expiration times which may occur sooner than the expiration of the original Verifiable Credential. In other words, verifiers may enforce their own expiration logic in order to force re-verification, or to create more conservative time windows for automatic credential expiration. However, the Verification Record must never have an expiration time that exceeds that of the original Verifiable Credential.

Verification Records may also be revoked by the verifier address that submitted the original Verification Result that led to a Verification Record's creation. This is appropriate for verifier-submitted results, but may not be effective for subject-submitted results, as subjects may submit to registries unknown to verifiers.

A subject may also remove any persistent Verification Records that have been associated with its own subject address. That is, an individual credential holder who has been verified has the ability to execute the removal of a Verification Record about his/her own verifications. Note that data is never truly removed from on-chain storage, but the act of executing a removal transaction will invalidate and zero-out the record from subsequent blocks in the chain. A specific registry implementation may remove this capability based on the contract logic; for example, a specific KYC registry may prevent removal of verifier-revoked credentials, or resubmission of any existing equivalent credential. These decisions are implementation-specific.

Verifier Management

VerificationRegistry contracts manage trusted verifiers. Specifically, registries map public verifier chain addresses toVerifierInfo objects that contain the following on-chain information:

- A Decentralized Identifier (DID) for the verifier
- A human-readable name for the verifier
- A URL pointing to more information about a verifier
- A public on-chain signer address which corresponds to a private key that the verifier uses to sign its VerificationResults

This information about a verifier is persisted on-chain. The registry contract's owner (by default its deployer) has the ability to add, update, and remove verifiers by supplying new VerifierInfo objects.

The registry contract's owner is therefore critically important -- the owner must vet and trust verifiers to execute proper verifications, and update or remove them if necessary. The owner is also capable of replacing itself with a new owner address.

The registry provides external-scoped accessor methods to obtain VerifierInfo for any configured verifier given a verifier's public chain address. The issuer associated with a particular credential and verification is not persisted or included in either the result or the record, but the verifier associated with a verification is persisted and is publicly visible. Verifiers may be contacted out of band by authorities in order to provide further data associated with a particular verification which they may persist privately.

Verifier Submission Patterns: Live or On-Chain

A verifier backend may submit a Verification Result or subset of its contents directly to a dapp or other relying smart contract, expressed as a transaction in the appropriate VM in principle, which the relying smart contract could verify without executing. This transaction could even be custodied and submitted by the wallet itself, if there were reason to justify the added UX complexity.

Another verifier-submission pattern involves an additional actor, an onchain-registry, which one or more smart contracts read asynchronously. In this version, a verifier executes the off-chain verification of a credential and then registers a risk-minimized subset of the Verification Result object called a Verification Record in some form of on-chain registry. A relying decentralized application or other client never accesses the credential or the Verification Result directly, instead relying on a trusted intermediary to translate verbose results to opaque Verification Records and maintain them on-chain. While this adds another indirection and on-chain risks and costs, its persistence on-chain simplifies on-going or multi-audience querying, as one or more smart contracts can execute a view (no cost) function on the registry contract over time (e.g. at each additional transaction, to check non-revocation).

This latter pattern supports granular "whitelisted account" and “addresses with attestations” approaches. It is available to web apps, dapps, and other smart contracts. It also integrates seamlessly with ERC-20 and similar standards since no custom functions are necessary.

Example Permissioned Pool or Token

The following sequence illustrates an example of the Verifier Submission Pattern in a hypothetical permissioned pool example:

Example Verification within an IDV and KYC Process

The permissioned example could also be embedded within an IDV/KYC sequence. In this example, an identity verifier might not only generate a Verifiable Credential attesting proof of KYC, but also register verification of the attestation with a registry contract.

For example:

Subject Submission Pattern

In the subject submission pattern, a dapp still utilizes an off-chain verifier for verification, but the verifier does not directly register the result with a smart contract. Instead, the verifier returns the Verification Result and its signature to the subject, and the subject passes the Verification Result and verifier’s signature to a smart contract's custom function, which can validate the result in addition to other logic in the same transaction.

This approach enables a dapp to execute verification and other contract logic in a single transaction and requires no fees from a verifier. However, trade-offs exist: In ethereum, it requires custom function usage so does not integrate seamlessly with existing transfer standards (such as ERC20), requires contract inheritance in order to ensure that the subject is the caller of the result validation function rather than the contract being the caller (to prevent replay attacks).

Example Risk Score Subject Submission

The following example illustrates subject-submitted verification results for the sake of providing hypothetical credit/risk scoring to a lending pool smart contract:

In this hypothetical example, a contract in the lending pool could inherit the registry contract (or implement the registry interface) to invoke internal verification registration methods in the same transaction as the borrow logic. This ensures that the caller is the subject (represented by the dapp) as opposed to the inheriting contract becoming the caller. There is no persistence of credit verifications in this example.

Subject Submission Pattern Risks and Mitigations

Verification registries that support both subject-managed verification results and verifier revocation must ensure they don't inadvertently allow subjects to override a verifier revocation through resubmission, as demonstrated in the following flow:

1. Subject registers verification result
2. Verifier revokes
3. Subject deletes, then resubmits original verification result (causing the record to appear unrevoked)

This risk can be avoided through policies enforced in the implementation of a registry that supports verifier revocation:

- Disallow subject submission
- Allow subject submission, but disallow subject deletion
- Disallow deletion of a revoked record

This can also be mitigated by the verifier applying an aggressive expiration for subject-held verification results, narrowing the window of effectiveness of such a threat.

Storage Patterns

Upon generating a Verification Record, a contract may follow one of two storage patterns: (a) in-contract storage, and (b) off-chain storage with optional oracle integration.

In-contract storage involves persisting Verification Records in a manner that associates verifier addresses with all of the Verification Records that a verifier's address has approved, and maps subject addresses to all Verification Records associated with that subject -- exposing no PII or verification result payloads, but providing proof of verification types, timestamps, and expirations that the subject has successfully passed.

An example that executes this pattern is theVerificationRegistry.sol contract.

The off-chain storage pattern involves no such on-chain storage. External oracles and chain scanners can observe the Verification Record events emitted by a contract, but there is no persistence of records on-chain.

This storage option may be appropriate for transient verifications with short-lived credentials (such as the risk score example above). For off-chain storage to support the Verifier Submission sequence, an oracle could be implemented to access accurate off-chain verification information.

Auditing and Forensics Sequences

No personal data or credentials are stored on-chain. Instead, Verification Records are used as references to a verifier where additional information about a verification may be found. The verifier’s private off-chain information includes, among other data, a record of the issuer of a subject's credential.

So while basic Verification Record information associated with addresses is public and can be correlated to transaction activity, and callers can verify that an address has a particular claim, an enforcement authority interested in obtaining PII about a subject would need to take the additional steps of gaining the issuer identifier from the verifier and then contacting that issuer directly.

A provable yet privacy-preserving audit path is simple. However, a deeper forensics path that accesses personal data is intended to be possible only for law enforcement with appropriate authority, and not possible for others to execute.

Privacy Considerations

While verification records may be extended to store additional properties, implementors should ensure data stored on-chain does not contribute to the re-identification of the individual to which it applies.

In general, this requires registry design to target clear use cases determined by stakeholders, without reliance of on-chain data for additional optionality (relying instead on offchain protocols when this is needed).

Alternative Patterns and Future Work

These patterns are not definitive, they serve only as initial examples to illustrate integrating decentralized identity standards -- Verifiable Credentials and Presentation Exchange usage -- into smart contracts and DeFi programs.

Future directions to explore include verification in-contract on chains that are capable of executing verifications efficiently and economically, deeper filtering of credentials rules in-contract instead of solely in verifier logic, alternative patterns and registry designs for specific schemas and credential types, and usage of technologies such as zero-knowledge proofs to encapsulate credential data on-chain for selective disclosure scenarios.

The source code examples referenced here are also not intended for production use. The implementations exist to illustrate the design patterns, not to support commercial production deployment.

Reference Implementations and Test Suite

A reference implementation smart contract that executes Verifier Management, Verification Result validation, and Verification Record persistence is the IVerificationRegistry interface and its VerificationRegistry.sol implementation:

https://github.com/circlefin/verite/tree/main/packages/contract/contracts

This registry implementation is used by two examples, PermissionedToken.sol (a verifier submission example) and ThresholdToken.sol (a subject submission example).

The test suite leverages hardhat and waffle, and is located in the ‘test’ subdirectory of the above repo:

https://github.com/circlefin/verite/blob/main/packages/contract/test/VerificationRegistryTests.ts

An example Dapp with MetaMask integration that uses the ThresholdToken is available here:

https://github.com/circlefin/verite/tree/main/packages/e2e-demo/components/demos/dapp Updated5 months ago

* Table of Contents * * Overview * * Registry-Based Architecture * * * Verification Results * * * Verification Records * * * Expiration and Revocation * * * Verifier Management * * Verifier Submission Patterns: Live or On-Chain * * * Example Permissioned Pool or Token * * * Example Verification within an IDV and KYC Process * * Subject Submission Pattern * * * Example Risk Score Subject Submission * * * Subject Submission Pattern Risks and Mitigations * * Storage Patterns * * Auditing and Forensics Sequences * * Privacy Considerations * * Alternative Patterns and Future Work * * Reference Implementations and Test Suite