An Architecture for Trustworthy and Transparent Digital Supply Chains
draft-birkholz-scitt-architecture-00

Abstract

Traceability of physical and digital artifacts in supply chains is a long-standing, but increasingly serious security concern. The rise in popularity of verifiable data structures as a mechanism to make actors more accountable for breaching their compliance promises has found some successful applications to specific use cases (such as the supply chain for digital certificates), but lacks a generic and scalable architecture that can address a wider range of use cases.

This memo defines a generic and scalable architecture to enable transparency across any supply chain with minimum adoption barriers for producers (who can register their claims on any TS, with the guarantee that all consumers will be able to verify them) and enough flexibility to allow different implementations of Transparency Services with various auditing and compliance requirements.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 8 September 2022.
Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the
document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal
license-info) in effect on the date of publication of this document.
Please review these documents carefully, as they describe your rights
and restrictions with respect to this document. Code Components
extracted from this document must include Revised BSD License text as
described in Section 4.e of the Trust Legal Provisions and are
provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction ...................................................... 3
   1.1. Requirements Notation ........................................ 5
2. Use Cases .......................................................... 5
   2.1. Software Bill of Materials (SBOM) ............................ 5
   2.2. Confidential Computing ....................................... 6
   2.3. Cold Chains for Seafood ..................................... 7
3. Terminology ......................................................... 7
4. Definition of Transparency ......................................... 8
5. Architecture Overview .............................................. 9
   5.1. Claim Issuance .................................................. 10
      5.1.1. Issuer Identity ........................................... 11
      5.1.2. Naming Artifacts ......................................... 11
      5.1.3. Claim Metadata ............................................ 11
   5.2. Transparency Service (TS) .................................... 12
      5.2.1. Service Identity, Remote Attestation, and Keying .... 12
      5.2.2. Registration Policies ..................................... 13
      5.2.3. Ledger Security Requirements ............................. 15
   5.3. Verifying Transparent Claims .................................. 17
6. Claim Issuance, Registration, and Verification ..................... 18
   6.1. Envelope and Claim Format .................................... 18
   6.2. Claim Issuance ................................................ 20
   6.3. Registering Signed Claims ................................... 20
   6.4. Validation of Transparent Claims ............................. 21
7. Federation .......................................................... 22
8. Transparency Service API ........................................... 22
   8.1. Messages ...................................................... 22
      8.1.1. Register Signed Claims .................................. 22
      8.1.2. Retrieve Registration Receipt ............................. 23
9. Privacy Considerations ............................................. 24
10. Security Considerations .......................................... 24
11. IANA Considerations .............................................. 25
12. References ......................................................... 25
1. Introduction

This document describes a scalable and flexible decentralized architecture to enhance auditability and accountability in various existing and emerging supply chains. It achieves this goal by enforcing the following complementary security guarantees:

1. statements made by issuers about supply chain artifacts must be identifiable, authentic, and non-repudiable;

2. such statements must be registered on a secure append-only ledger so that their provenance and history can be independently and consistently audited;

3. issuers can efficiently prove to any other party the registration of their claims; verifying this proof ensures that the issuer is consistent and non-equivocal when making claims.

The first guarantee is achieved by requiring issuers to sign their statements and associated metadata using a distributed public key infrastructure. The second guarantee is achieved by storing the signed statement on an immutable, append-only, transparent ledger. The last guarantee is achieved by implementing the ledger using a verifiable data structure (such as a Merkle Tree), and by requiring a transparency service (TS) that operates the ledger to endorse its state at the time of registration.

The guarantees and techniques used in this document generalize those of Certificate Transparency [RFC9162], which can be re-interpreted as an instance of this architecture for the supply chain of X.509 certificates. However, the range of use cases and applications in this document is much broader, which requires much more flexibility in how each TS implements and operates its ledger. Each service may enforce its own policy for authorizing entities to register their claims on the TS. Some TS may also enforce access control policies to limit who can audit the full ledger, or keep some information on the ledger encrypted. Nevertheless, it is critical to provide global interoperability for all TS instances as the composition and configuration of involved supply chain entities and their system components is ever changing and always in flux.
A TS provides visibility into claims issued by supply chain entities and their sub-systems. These claims are called Digital Supply Chain Artifacts (DSCA). A TS vouches for specific and well-defined metadata about these DSCAs. Some metadata is selected (and signed) by the issuer, indicating, e.g., "who issued the DSCA" or "what type of DSCA is described" or "what is the DSCA version"; whereas additional metadata is selected (and countersigned) by the TS, indicating, e.g., "when was the DSCA registered in the ledger". The DSCA contents can be opaque to the TS, if so desired: it is the metadata that must always be transparent in order to warrant trust.

Transparent claims provide a common basis for holding issuers accountable for the DSCA they release and (more generally) principals accountable for auxiliary claims they make about DSCAs. Hence, issuers may register new claims about their artifacts, but they cannot delete or alter earlier claims, or hide their claims from third parties such as auditors.

Trust in the TS itself is supported both by protecting their implementation (using, for instance, replication, trusted hardware, and remote attestation of systems) and by enabling independent audits of the correctness and consistency of its ledger, thereby holding the organization accountable that operates it. Unlike CT, where independent auditors are responsible for enforcing the consistency of multiple independent instances of the same global ledger, we require each TS to guarantee the consistency of its own ledger (for instance, through the use of a consensus algorithm between replicas of the ledger), but assume no consistency between different transparency services.

The TS specified in this architecture caters to two types of audiences:

1. DSCA Issuers: entities, stakeholders, and users involved in supply chain interactions that need to release DSCAs to a definable set of peers; and

2. DSCA Consumers: entities, stakeholders, and users involved in supply chain interactions that need to access, validate, and trust DSCAs.
DSCA Issuers rely on being discoverable and represented as the responsible parties for released DSCAs by the TS in a believable manner. Analogously, DSCA Consumers rely on verifiable trustworthiness assertions associated with DSCAs and their processing in a believable manner. If trust can be put into the operations that record DSCAs in a secure, append-only ledger via an online operation, the same trust can be put into a corresponding receipt that is the result of these online operations issued by the TS and that can be validated in offline operations.

The TS specified in this architecture can be implemented by various different types of services in various types of languages provided via various variants of API layouts.

The global interoperability enabled and guaranteed by the TS is enabled via core components (architectural constituents) that come with prescriptive requirements (that are typically hidden away from the user audience via APIs). The core components are based on the Concise Signing and Encryption standard specified in [RFC8152], which is used to sign released DSCAs and to build and maintain a Merkle tree that functions as the append-only ledger for DSCAs. The format and verification process for ledger-based transparency receipts are described in [I-D.birkholz-scitt-receipts].

1.1. Requirements Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Use Cases

This section presents representative and solution-agnostic use cases to illustrate the scope of SCITT and the processing of Digital Supply Chain Artifacts.

2.1. Software Bill of Materials (SBOM)

As the ever increasing complexity of large software projects requires more modularity and abstractions to manage them, keeping track of their full Trusted Computing Base (TCB) is becoming increasingly difficult. Each component may have its own set of dependencies and libraries. Some of these dependencies are binaries, which means their TCB depends not only on their source, but also on their build environment (compilers and tool-chains). Besides, many source and binary packages are distributed through various channels and
repositories that may not be trustworthy.

Software Bills of Materials (SBOM) help the authors, packagers, distributors, auditors and users of software understand its provenance and who may have the ability to introduce a vulnerability that can affect the supply chain downstream. However, the usefulness of SBOM in protecting end users is limited if supply chain actors cannot be held accountable for their contents. For instance, consider a package repository for an open source operating system distribution. The operator of this repository may decide to provide a malicious version of a package only to users who live in a specific country. They can write two equivocal SBOMs for the honest and backdoored versions of the package, so that nobody outside the affected country can discover the malicious version, but victims are not aware they are being targeted.

2.2. Confidential Computing

Confidential Computing can leverage hardware-protected trusted execution environments (TEEs) to operate cloud services that protect the confidentiality of data that they process. It relies on remote attestation, which allows the service to prove to remote users what is the hash of its software, as measured and signed by the hardware.

For instance, consider a speech recognition service that implements machine learning inference using a deep neural network model. The operator of the service wants to prove to its users that the service preserves the user’s privacy, that is, the submitted recordings can only be used to detect voice commands but no other purpose (such as storing the recordings or detecting mentions of brand names for advertisement purposes). When the user connects to the TEE implementing the service, the TEE presents attestation evidence that includes a hardware certificate and a software measurement for their task; the user verifies this evidence before sending its recording.

But how can users verify the software measurement for their task? And how can operators update their service, e.g., to mitigate security vulnerabilities or improve accuracy, without first convincing all users to update the measurements they trust?

A supply chain that maintains a transparent record of the successive software releases for machine-learning models and runtimes, recording both their software measurements and their provenance (source code, build reports, audit reports,...) can provide users with the information they need to authorize these tasks, while holding the service operator accountable for the software they release for them.
2.3. Cold Chains for Seafood

Once seafood is caught, its quality is determined -- amongst other criteria -- via the integrity of a cold chain that ensures a regulatory perspective freshness mandating a continuous storing temperature between 1 °C and 0 °C (or -18 °C and lower for frozen seafood). The temperature is recorded by cooling units adhering to certain compliance standards automatically. Batches of seafood can be split or aggregated before arriving in a shelf so that each unit can potentially have a potentially unique cold chain record whose transparency impacts the accuracy of the shelf-life associated with it. Especially in early links of the supply chain, Internet connection or sophisticated IT equipment are typically not available and sometimes temperature measurements are recorded manually and digital records are created in hindsight.

3. Terminology

The terms defined in this section have special meaning in the context of Supply Chain Integrity, Transparency, and Trust throughout this document. When used in text, the corresponding terms are capitalized. To ensure readability, only a core set of terms is included in this section.

Artifact: the physical or non-physical item that is moving along the supply chain.

Statement: any serializable information about an Artifact. To help interpretation of Statements, they must be tagged with a media type (as specified in [RFC6838]).

Claim: an identifiable and non-repudiable Statement about an Artifact made by an Issuer. In SCITT, Claims are encoded as COSE signed objects; the payload of the COSE structure contains the Statement.

Issuer: creator of Claims submitted to a Transparency Service for Registration. The Issuer may be the owner or author of the Artifact, or a completely independent third party.

Envelope: the metadata added to the Statement by the Issuer to make it a Claim. It contains the identity of the Issuer and other information to help Verifiers identify the Artifact referred in the Statement. A Claim binds the Envelope to the Statement. In COSE, the Envelope consists of protected headers.

Feed: An identifier chosen by the Issuer for the Artifact. For
every Issuer and Feed, the Ledger on a Transparency Service contains a sequence of Claims about the same Artifact. In COSE, Feed is one header attributes in the protected header of the Envelope.

Ledger: the verifiable data structure that stores Claims in a transparency service. SCITT supports multiple Ledger formats to accommodate different transparency service implementations, such as historical Merkle Trees and sparse Merkle Trees.

Transparency Service: the entity that maintains and extends the Ledger, and endorses its state. A Transparency Service can be a complex distributed system, and SCITT requires the TS to provide many security guarantees about its Ledger. The identity of a TS is captured by a public key that must be known by Verifiers in order to validate Receipts.

Receipt: a Receipt is a special form of COSE countersignature for Claims that embeds cryptographic evidence that the Claim is recorded in the Ledger. It consists of a Ledger-specific inclusion proof, a signature by the Transparency Service of the state of the Ledger, and additional metadata (contained in the countersignature protected headers) to assist in auditing.

Registration: the process of submitting a Claim to a Transparency Service, applying its registration policy, storing it in the Ledger and producing the Receipt returned to the submitter.

Transparent Claim: a Claim that is augmented with a Receipt of its registration. A Transparent Claim remains a valid Claim (as the Receipt is carried in the countersignature), and may be registered again in a different TS.

Verifier: the entity that consumes Transparent Claims, verifying their proofs and inspecting their Statements, either before using their Artifacts, or later to audit their supply chain.

4. Definition of Transparency

In this document, we use a definition of transparency built over abstract notions of Ledgers and Receipts. Existing transparency systems such as Certificate Transparency are instances of this definition.

A Claim is an identifiable and non-repudiable Statement made by an Issuer. The Issuer selects additional metadata and attaches a proof of endorsement (in most cases, a signature) using the identity key of the Issuer that binds the Statement and its metadata. Claims can be
made transparent by attaching a proof of Registration by a TS, in the form of a Receipt that countersigns the Claim and witnesses its inclusion in the Ledger of a TS. By extension, we may say an Artifact (e.g. a firmware binary) is transparent if it comes with one or more Transparent Claims from its author or owner, though the context should make it clear what type of Claim is expected for a given Artifact.

Transparency does not prevent dishonest or compromised Issuers, but it holds them accountable: any Artifact that may be used to target a particular user that checks for Receipts must have been recorded in the tamper-proof Ledger, and will be subject to scrutiny and auditing by other parties.

Transparency is implemented by a Ledger that provides a consistent, append-only, publicly available record of entries. Implementations of TS may protect their Ledger using a combination of trusted hardware, replication and consensus protocols, and cryptographic evidence. A Receipt is an offline, universally-verifiable proof that an entry is recorded in the Ledger. Receipts do not expire, but it is possible to append new entries that subsume older entries.

Anyone with access to the Ledger can independently verify its consistency and review the complete list of Claims registered by each Issuer. However, the Ledgers of separate Transparency Services are generally disjoint, though it is possible to take a Claim from one Ledger and register it again on another (if its policy allows it), so the authorization of the Issuer and of the Ledger by the Verifier of the Receipt are generally independent.

Reputable Issuers are thus incentivized to carefully review their Statements before signing them into Claims. Similarly, reputable TS are incentivized to secure their Ledger, as any inconsistency can easily be pinpointed by any auditor with read access to the Ledger. Some Ledger formats may also support consistency auditing through Receipts, that is, given two valid Receipts the TS may be asked to produce a cryptographic proof that they are consistent. Failure to produce this proof can indicate that the TS operator misbehaved.

5. Architecture Overview
The SCITT architecture consists of a very loose federation of Transparency Services, and a set of common formats and protocols for issuing, registering and auditing Claims. In order to accommodate as many TS implementations as possible, this document only specifies the format of Claims (which must be used by all Issuers) and a very thin wrapper format for Receipts, which specifies the TS identity and the Ledger algorithm. Most of the details of the Receipt’s contents are specific to the Ledger algorithm. The [I-D.birkholz-scitt-receipts] document defines two initial Ledger algorithms (for historical and sparse Merkle Trees), but other Ledger formats (such as blockchains, or hybrid historical and indexed Merkle Trees) may be proposed later.

In this section, we describe at a high level the three main roles and associated processes in SCITT: Issuers and the Claim issuance process, transparency Ledgers and the Claim Registration process, and Verifiers and the Receipt validation process.

5.1. Claim Issuance
5.1.1. Issuer Identity

Before an Issuer is able to produce Claims, it must first create its decentralized identifier (https://www.w3.org/TR/did-core) (also known as a DID). A DID can be _resolved_ into a _key manifest_ (a list of public keys indexed by a _key identifier_) using many different DID methods.

Issuers MAY choose the DID method they prefer, but with no guarantee that all TS will be able to register their Claim. To facilitate interoperability, all Transparency Service implementations SHOULD support the did:web method from [https://w3c-ccg.github.io/did-method-web/]. For instance, if the Issuer publishes its manifest at https://sample.issuer/user/alice/did.json, the DID of the Issuer is did:web:sample.issuer:user:alice.

Issuers SHOULD use consistent decentralized identifiers for all their Artifacts, to simplify authorization by Verifiers and auditing. They MAY update their DID manifest, for instance to refresh their signing keys or algorithms, but they SHOULD NOT remove or change any prior keys unless they intend to revoke all Claims issued with those keys. This DID appears in the Issuer header of the Claim’s Envelope, while the version of the key from the manifest used to sign the Claim is written in the kid header.

5.1.2. Naming Artifacts

Many Issuers issue Claims about different Artifacts under the same DID, so it is important for everyone to be able to immediately recognize by looking at the Envelope of a Claim what Artifact it is referring to. This information is stored in the Feed header of the Envelope. Issuers MAY use different signing keys (identified by kid in the resolved key manifest) for different Artifacts, or sign all Claims under the same key.

5.1.3. Claim Metadata

Besides Issuer, Feed and kid, the only other mandatory metadata in the Claim is the type of the Payload, indicated in the cty Envelope header. However, this set of mandatory metadata is not sufficient to express many important Registration policies. For example, a Ledger may only allow a Claim to be registered if it was signed recently. While the Issuer is free to add any information in the payload of the Claim, the TS (and most of its auditor) can only be expected to interpret information in the Envelope.
Such metadata, meant to be interpreted by the TS during Registration policy evaluation, should be added to the reg_info header. While the header MUST be present in all Claims, its contents consist of a map of named attributes. Some attributes (such as the Issuer’s timestamp) are standardized with a defined type, to help uniformize their semantics across TS. Others are completely customizable and may have arbitrary types. In any case, all attributes are optional so the map MAY be empty.

5.2. Transparency Service (TS)

The role of TS can be decomposed into several major functions. The most important is maintaining a Ledger, the verifiable data structure that records Claims, and enforcing a Registration policy. It also maintains a service key, which is used to endorse the state of the Ledger in Receipts. All TS MUST expose standard endpoints for Registration of Claims and Receipt issuance, which is described in Section 8.1. Each TS also defines its Registration policy, which MUST apply to all entries in the Ledger.

The combination of Ledger, identity, Registration policy evaluation, and Registration endpoint constitute the trusted part of the TS. Each of these components SHOULD be carefully protected against both external attacks and internal misbehavior by some or all of the operators of the TS. For instance, the code for policy evaluation, Ledger extension and endorsement may be protected by running in a TEE; the Ledger may be replicated and a consensus algorithm such as Practical Byzantine Fault Tolerance (pBFT [PBFT]) may be used to protect against malicious or vulnerable replicas; threshold signatures may be use to protect the service key, etc.

Beyond the trusted components, Transparency Services may operate additional endpoints for auditing, for instance to query for the history of Claims made by a given Issuer and Feed. Implementations of TS SHOULD avoid using the service identity and extending the Ledger in auditing endpoints; as much as practical, the Ledger SHOULD contain enough evidence to re-construct verifiable proofs that the results returned by the auditing endpoint are consistent with a given state of the Ledger.

5.2.1. Service Identity, Remote Attestation, and Keying

Every TS MUST have a public service identity, associated with public/private key pairs for signing on behalf of the service. In particular, this identity must be known by Verifiers when validating a Receipt.
This identity should be stable for the lifetime of the service, so that all Receipts remain valid and consistent. The TS operator MAY use a distributed identifier as their public service identity if they wish to rotate their keys, if the Ledger algorithm they use for their Receipt supports it. Other types of cryptographic identities, such as parameters for non-interactive zero-knowledge proof systems, may also be used in the future.

The TS SHOULD provide evidence that it is securely implemented and operated, enabling remote authentication of the hardware platforms and/or software TCB that run the TS. This additional evidence SHOULD be recorded in the Ledger and presented on demand to Verifiers and auditors.

For example, consider a TS implemented using a set of replicas, each running within its own hardware-protected trusted execution environments (TEEs). Each replica SHOULD provide a recent attestation report for its TEE, binding their hardware platform to the software that runs the Transparency Service, the long-term public key of the service, and the key used by the replica for signing Receipts. This attestation evidence SHOULD be supplemented with transparency Receipts for the software and configuration of the service, as measured in its attestation report.

5.2.2. Registration Policies

*Editor’s note*

The initial version of this document assumes Registration policies are set for the lifetime of the Ledger, and that they apply to all Issuers and Feeds uniformly. There is an ongoing discussion on how to make the design more flexible to allow per-Issuer and per-Feed Registration policies, and whether such policies should be updatable or if a policy change requires a Feed change. Please contribute your comments to the SCITT mailing list.

Each TS is initially configured with a set of Registration policies, which will be applied for the lifetime of the Ledger. A Registration policy represents a predicate that takes as input the current Ledger and the Envelope of a new Claim to register (including the reg_info header which contains customizable additional attributes), and returns a Boolean decision on whether the Claim should be included on the Ledger or not. A TS MUST ensure that all its Registration policies return a positive decision before adding a Claim to the Ledger.
While Registration policies are a burden for Issuers (some may require them to maintain state to remember what they have signed before) they support stronger transparency guarantees, and they greatly help Verifiers and auditors in making sense of the information on the Ledger. (This is particularly relevant for parties that verify Receipts on their own, without accessing the Ledger.) For instance, if a TS doesn’t apply any policy, Claims may be registered in a different order than they have been issued, and old Claims may be replayed, which makes it difficult to understand the logical history of an Artifact, or to prevent rollback attacks.

There are two kinds of Registration policies: (1) named policies have standardized semantics that are uniform across all implementations of SCITT Transparency Services, while (2) custom policies are opaque and may contain pointers to (or even inlined) policy descriptions (declarative or programmable).

Transparency services MUST advertise the Registration policies enforced by their service, including the list of reg_info attributes they require, both to minimize the risk of rejecting Claims presented by Issuers, and to advertise the properties implied by Receipt verification. Implementations of Receipt Verifiers SHOULD persist the list of Registration policies associated with a service identity, and return the list of Registration policies as an output of Receipt validation. Auditors MUST re-apply the Registration policy of every entry in the Ledger to ensure that the Ledger applied them correctly.

Custom policies may use additional information present in the Ledger outside of Claims. For instance, Issuers may have to register on the TS before Claims can be accepted; a custom policy may be used to enforce access control to the Transparency Service. Verifying the signature of the Issuer is also a form of Registration policy, but it is globally enforced in order to separate authentication and authorization, with policy only considering authentic inputs.

Table 1 defines an initial set of named policies that TS may decide to enforce. This may be evolved in future drafts.
<table>
<thead>
<tr>
<th>Policy Name</th>
<th>Required attributes</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeLimited</td>
<td>register_by: uint</td>
<td>Returns true if now () &lt; register_by. The Ledger MUST store the Ledger time at Registration along with the Claim, and SHOULD indicate it in Receipts</td>
</tr>
<tr>
<td>Sequential</td>
<td>sequence_no: uint</td>
<td>First, lookup in the Ledger for Claims with the same Issuer and Feed. If at least one is found, returns true if and only if the sequence_no of the new Claim is the highest sequence_no in the existing Claims incremented by one. Otherwise, returns true if and only if sequence_no = 0.</td>
</tr>
<tr>
<td>Temporal</td>
<td>issuance_ts: uint</td>
<td>Returns true if and only if there is no Claim in the Ledger with the same Issuer and Feed with a greater issuance_ts</td>
</tr>
<tr>
<td>NoReplay</td>
<td>None</td>
<td>Returns true if and only if the Claim doesn’t already appear in the Ledger</td>
</tr>
</tbody>
</table>

Table 1: An Initial Set of Named Policies

5.2.3. Ledger Security Requirements

There are many different candidate verifiable data structures that may be used to implement the Ledger, such as chronological Merkle Trees, sparse/indexed Merkle Trees, full blockchains, and many other variants. We only require the Ledger to support concise Receipts (i.e. whose size grows at most logarithmically in the number of entries in the Ledger). This does not necessarily rule out blockchains as a Ledger, but may necessitate advanced Receipt schemes that use arguments of knowledge and other verifiable computing techniques.

Since the details of how to verify a Receipt are specific to the data structure, we do not specify any particular Ledger format in this document. Instead, we propose two initial formats for Ledgers in
5.2.3.1. Finality

The Ledger is append-only: once a Claim is registered, it cannot be modified, deleted, or moved. In particular, once a Receipt is returned for a given Claim, the Claim and any preceding entry in the Ledger become immutable, and the Receipt provides universally-verifiable evidence of this property.

5.2.3.2. Consistency

There is no fork in the Ledger: everyone with access to its contents sees the same sequence of entries, and can check its consistency with any Receipts they have collected. TS implementations SHOULD provide a mechanism to verify that the state of the Ledger encoded in an old Receipt is consistent with the current Ledger state.

5.2.3.3. Replayability and Auditing

Everyone with access to the Ledger can check the correctness of its contents. In particular,

* the TS defines and enforces deterministic Registration policies that can be re-evaluated based solely on the contents of the Ledger at the time of registraton, and must then yield the same result.

* The ordering of entries, their cryptographic contents, and the Ledger governance may be non-deterministic, but they must be verifiable.

* The TS SHOULD store evidence about the resolution of distributed identifiers into manifests.

* The TS MAY additionally support verifiability of client authentication and access control.
5.2.3.4. Governance and Bootstrapping

The TS needs to support governance, with well-defined procedures for allocating resources to operate the Ledger (e.g., for provisioning trusted hardware and registering their attestation materials in the Ledger) and for updating its code (e.g., relying on Transparent Claims about code updates, secured on the Ledger itself, or on some auxiliary TS).

Governance procedures, their auditing, and their transparency are implementation specific. The TS SHOULD document them.

* Governance may be based on a consortium of members that are jointly responsible for the TS, or automated based on the contents of an auxiliary governance TS.

* Governance typically involves additional records in the Ledger to enable its auditing. Hence, the Ledger may contain both Transparent Claims and governance entries.

* Issuers, Verifiers, and third-party auditors may review the TS governance before trusting the service, or on a regular basis.

5.3. Verifying Transparent Claims

For a given Artifact, Verifiers take as trusted inputs:

1. the distributed identifier of the Issuer (or its resolved key manifest),

2. the expected name of the Artifact (i.e. the Feed),

3. the list of service identities of trusted TS.

When presented with a Transparent Claim for the Artifact, they verify its Issuer identity, signature, and Receipt. They may additionally apply a validation policy based on the protected headers present both in the Envelope or in the countersignature and the Statement itself, which may include security-critical Artifact-specific details.

Some Verifiers may systematically resolve the Issuer DID to fetch their latest DID document. This strictly enforces the revocation of compromised keys: once the Issuer has updated its document to remove a key identifier, all Claims signed with this kid will be rejected. However, others may delegate DID resolution to a trusted third party and/or cache its results.
Some Verifiers may decide to skip the DID-based signature verification, relying on the TS’s Registration policy and the scrutiny of other Verifiers. Although this weakens their guarantees against key revocation, or against a corrupt TS, they can still keep the Receipt and blame the Issuer or the TS at a later point.

6. Claim Issuance, Registration, and Verification

This section details the interoperability requirements for implementers of Claim issuance and validation libraries, and of Transparency Services.

6.1. Envelope and Claim Format

The formats of Claims and Receipts are based on CBOR Object Signing and Encryption (COSE). The choice of CBOR is a trade-off between safety (in particular, non-malleability: each Claim has a unique serialization), ease of processing and availability of implementations.

At a high-level that is the context of this architecture, a Claim is a COSE single-signed object (i.e. COSE_Sign1) that contains the correct set of protected headers. Although Issuers and relays may attach unprotected headers to Claims, Transparency Services and Verifiers MUST NOT rely on the presence or value of additional unprotected headers in Claims during Registration and validation.

All Claims MUST include the following protected headers:

* algorithm (label: 1): Asymmetric signature algorithm used by the Claim Issuer, as an integer, for example -35 for ECDSA with SHA-384, see COSE Algorithms registry (https://www.iana.org/assignments/cose/cose.xhtml);

* Issuer (label: TBD, to be registered): DID (Decentralized Identifier, see W3C Candidate Recommendation (https://www.w3.org/TR/did-core/) of the signer, as a string, for example did:web:example.com;

* Feed (label: TBD): the Issuer’s name for the Artifact, as a string;

* payload type (label: 3): Media type of payload as a string, for example application/spdx+json

* Registration policy info (label: TBD): a map of additional attributes to help enforce Registration policies;
* DID key selection hint (label: TBD): a DID method-specific selector for the signing key, as a bytestring.

Additionally, Claims MAY carry the following unprotected headers:

* Receipts (label: TBD, to be registered): Array of Receipts, defined in [I-D.birkholz-scitt-receipts]

In CDDL [RFC8610] notation, the Envelope is defined as follows:

```
SCITT_Envelope = COSE_Sign1_Tagged

COSE_Sign1_Tagged = #6.18(COSE_Sign1)

COSE_Sign1 = [
  protected : bstr .cbor Protected_Header,
  unprotected : Unprotected_Header,
  payload : bstr,
  signature : bstr
]

Reg_Info = {
  ? "register_by": uint,
  ? "sequence_no": uint,
  ? "issuance_ts": uint,
  * tstr => any
}
```

; All protected headers are mandatory, to protect against faulty implementations of COSE
; that may accidentally read a missing protected header from the unprotected headers.

```
Protected_Header = {
  1 => int ; algorithm identifier
  3 => tstr ; payload type
  258 => tstr ; DID of Issuer
  259 => tstr ; Feed
  260 => Reg_Info ; Registration policy info
  261 => bstr ; key selector
}
```

```
Unprotected_Header = {
  ? 257 => SCITT_Receipt / [+ SCITT_Receipt]
}
```
6.2. Claim Issuance

There are many types of Statements (such as SBOMs, malware scans, audit reports, policy definitions) that Issuers may want to turn into Claims. The Issuer must first decide on a suitable format to serialize the Statement, such as: - JSON-SPDX - CBOR-SPDX - SWID - CoSWID - CycloneDX - in-toto - SLSA

Once the Statement is serialized with the correct content type, the Issuer should fill in the attributes for the Registration policy information header. From the Issuer's perspective, using attributes from named policies ensures that the Claim may only be registered on Transparency Services that implement the associated policy. For instance, if a Claim is frequently updated, and it is important for Verifiers to always consider the latest version, Issuers SHOULD use the `sequence_no` or `issuer_ts` attributes.

Once all the Envelope headers are set, the Issuer MAY use a standard COSE implementation to produce the serialized Claim (the SCITT tag of COSE_Sign1_Tagged is outside the scope of COSE, and used to indicate that a signed object is a Claim).

6.3. Registering Signed Claims

The same Claim may be independently registered in multiple TS. To register a Claim, the service performs the following steps:

1. Client authentication. This is implementation-specific, and MAY be unrelated to the Issuer identity. Claims may be registered by a different party than their Issuer.

2. Issuer identification. The TS MUST store evidence of the DID resolution for the Issuer protected header of the Envelope and the resolved key manifest at the time of Registration for auditing. This MAY require that the service resolve the Issuer DID and record the resulting document, or rely on a cache of recent resolutions.

3. Envelope signature verification, as described in COSE signature, using the signature algorithm and verification key of the Issuer DID document.

4. Envelope validation. The service MUST check that the Envelope has a payload and the protected headers listed above. The service MAY additionally verify the payload format and content.
5. Apply Registration policy: for named policies, the TS should check that the required Registration info attributes are present in the Envelope and apply the check described in Table 1. A TS MUST reject Claims that contain an attribute used for a named policy that is not enforced by the service. Custom Claims are evaluated given the current Ledger state and the entire Envelope, and MAY use information contained in the attributes of named policies.

6. Commit the new Claim to the Ledger

7. Sign and return the Receipt.

The last two steps MAY be shared between a batch of Claims recorded in the Ledger.

The service MUST ensure that the Claim is committed before releasing its Receipt, so that it can always back up the Receipt by releasing the corresponding entry in the Ledger. Conversely, the service MAY re-issue Receipts for the Ledger content, for instance after a transient fault during Claim Registration.

6.4. Validation of Transparent Claims

This section provides additional implementation considerations, the high-level validation algorithm is described in Section 5.3, with the Ledger-specific details of checking Receipts are covered in [I-D.birkholz-scitt-receipts].

Before checking a Claim, the Verifier must be configured with one or more identities of trusted Transparency Services. If more than one service is configured, the Verifier MUST return which service the Claim is registered on.

In some scenarios, the Verifier already expects a specific Issuer and Feed for the Claim, while in other cases they are not known in advance and can be an output of validation. Verifiers SHOULD offer a configuration to decide if the Issuer’s signature should be locally verified (which may require a DID resolution, and may fail if the manifest is not available or if the key is revoked), or if it should trust the validation done by the TS during Registration.

Some Verifiers MAY decide to locally re-apply some or all of the Registration policies if they have limited trust in the TS. In addition, Verifiers MAY apply arbitrary validation policies after the signature and Receipt have been checked. Such policies may use as input all information in the Envelope, the Receipt, and the payload, as well as any local state.
Verifiers SHOULD offer options to store or share Receipts in case they are needed to audit the TS in case of a dispute.

7. Federation

We explain how multiple, independent Transparency Services can be composed to distribute supply chains without a single transparency authority trusted by all parties.

Multiple SCITT instances, governed and operated by different organizations.

For example, - a small, simple SCITT instance may keep track specifically of the software used for operating SCITT services. - an air-gapped data center may operate its own SCITT Ledger to retain full control and auditing of its software supplies.

How? - Policy-based. Within an organization, local Verifiers contact an authoritative SCITT that records the latest policies associated with classes of Artifacts; these policies indicate which Issuers and Ledgers are trusted for verifying signed Transparent Claims for these Artifacts.

* Other federation mechanisms?

We’d like to attach multiple Receipts to the same signed Claims, each Receipt endorsing the Issuer signature and a subset of prior Receipts. This involves down-stream Ledgers verifying and recording these Receipts before issuing their own Receipts.

8. Transparency Service API

Editor’s Note: this may be moved to appendix.

8.1. Messages

8.1.1. Register Signed Claims

8.1.1.1. Request

POST <Base URL>/entries

Body: SCITT COSE_Sign1 message

8.1.1.2. Response

One of the following:
* HTTP Status 201 - Registration was tentatively successful pending service consensus.

* HTTP Status 400 - Registration was unsuccessful.
  - Error code AwaitingDIDResolutionTryLater
  - Error code InvalidInput

[TODO] Use 5xx for AwaitingDIDResolutionTryLater

The 201 response contains the x-ms-ccf-transaction-id HTTP header which can be used to retrieve the Registration Receipt with the given transaction ID. [TODO] this has to be made generic

[TODO] probably a bad idea to define a new header, or is it ok? can we register a new one? https://www.iana.org/assignments/http-fields/http-fields.xhtml

The 400 response has a Content-Type: application/json header and a body containing details about the error:

```json
{ "error": { "code": "<error code>" , "message": "<message>" } }
```

AwaitingDIDResolutionTryLater means the service does not have an up-to-date DID document of the DID referenced in the Signed Claims but is performing or will perform a DID resolution after which the client may retry the request. The response may contain the HTTP header Retry-After to inform the client about the expected wait time.

InvalidInput means either the Signed Claims message is syntactically malformed, violates the signing profile (e.g. signing algorithm), or has an invalid signature relative to the currently resolved DID document.

8.1.2. Retrieve Registration Receipt

8.1.2.1. Request

```
GET <Base URL>/entries/<Transaction ID>/receipt
```

8.1.2.2. Response

One of the following:

* HTTP Status 200 - Registration was successful and the Receipt is returned.
* HTTP Status 400 - Transaction exists but does not correspond to a Registration Request.
  - Error code TransactionMismatch

* HTTP Status 404 - Transaction is pending, unknown, or invalid.
  - Error code TransactionPendingOrUnknown
  - Error code TransactionInvalid

The 200 response contains the SCITT_Receipt in the body.

The 400 and 404 responses return the error details as described earlier.

The retrieved Receipt may be embedded in the corresponding COSE_Sign1 document in the unprotected header, see TBD.

[TODO] There's also the GET <Base URL>/entries/<Transaction ID> endpoint which returns the submitted COSE_Sign1 with the Receipt already embedded. Is this useful?

9. Privacy Considerations

Unless advertised by the TS, every Issuer should treat its Claims as public. In particular, their Envelope and Statement should not carry any private information in plaintext.

10. Security Considerations

On its own, verifying a Transparent Claim does not guarantee that its Envelope or contents are trustworthy---just that they have been signed by the apparent Issuer and counter-signed by the TS. If the Verifier trusts the Issuer, it can infer that the Claim was issued with this Envelope and contents, which may be interpreted as the Issuer saying the Artifact is fit for its intended purpose. If the Verifier trusts the TS, it can independently infer that the Claim passed the TS Registration policy and that has been persisted in the Ledger. Unless advertised in the TS Registration policy, the Verifier should not assume that the ordering of Transparent Claims in the Ledger matches the ordering of their issuance.

Similarly, the fact that an Issuer can be held accountable for its Transparent Claims does not on its own provide any mitigation or remediation mechanism in case one of these Claims turned out to be misleading or malicious---just that signed evidence will be available to support them.
Issuers SHOULD ensure that the Statements in their Claims are correct and unambiguous, for example by avoiding ill-defined or ambiguous formats that may cause Verifiers to interpret the Claim as valid for some other purpose.

Issuers and Transparency Services SHOULD carefully protect their private signing keys and avoid these keys for any purpose not described in this architecture. In case key re-use is unavoidable, they MUST NOT sign any other message that may be verified as an Envelope.

11. IANA Considerations

See Body Section 4.

12. References

12.1. Normative References


12.2. Informative References
Appendix A. Attic

Not ready to throw these texts into the trash bin yet.

Authors’ Addresses

Henk Birkholz
Fraunhofer SIT
Rheinstrasse 75
64295 Darmstadt
Germany
Email: henk.birkholz@sit.fraunhofer.de

Antoine Delignat-Lavaud
Microsoft Research
21 Station Road
Cambridge
CB1 2FB
United Kingdom
Email: antdl@microsoft.com

Cedric Fournet
Microsoft Research
21 Station Road
Cambridge
CB1 2FB
United Kingdom
Email: fournet@microsoft.com
Countersigning COSE Envelopes in Transparency Services
draft-birkholz-scitt-receipts-00

Abstract

A transparent and authentic ledger service in support of a supply chain’s integrity, transparency, and trust requires all peers that contribute to the ledgers operations to be trustworthy and authentic. In this document, a countersigning variant is specified that enables trust assertions on merkle-tree based operations for global supply chain ledgers. A generic procedure how to produce payloads for signing and validation is defined and leverages solutions and principles from the Concise Signing and Encryption (COSE) space.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 8 September 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document.
1. Introduction

This document defines a method for issuing and verifying countersignatures on COSE_Sign1 messages included in an authenticated data structure such as a Merkle Tree.

We adopt the terminology of architecture (pointer) for Claim, Envelope, Transparency Service, Ledger, Receipt, and Verifier.
[TODO] Do we need to explain or introduce them here? We may also define Tree (our shorthand for authenticated data structure), Root (a succinct commitment to the Tree, e.g., a hand) and use Issuer instead of TS.

From the Verifier’s viewpoint, a Receipt is similar to a countersignature V2 on a single signed message: it is a universally-verifiable cryptographic proof of endorsement of the signed envelope by the countersigner.

Compared with countersignatures on single COSE envelopes, Receipts countersign the envelope in context, providing authentication both of the envelope and of its logical position in the authenticated data structure. Receipts are proof of commitment to the whole contents of the data structure, even if the Verifier knows only some of its contents. Receipts can be issued in bulk, using a single public-key signature for issuing a large number of Receipts.

1.1. Requirements Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Common Parameters

Verifiers are configured by a collection of parameters to identify a Transparency Service and verify its Receipts. These parameters MUST be fixed for the lifetime of the Transparency Service and securely communicated to all Verifiers.

At minimum, these parameters include:

* a Service identifier: An opaque identifier (e.g. UUID) that uniquely identifies the service and can be used to securely retrieve all other Service parameters.

* The Tree algorithm used for issuing receipts, and its additional global parameters, if any. This document creates a registry (see Section 9.2.1) and describes an initial set of tree algorithms.

[TODO] The architecture also has fixed TS registration policies.
3. Generic Receipt Structure

A Receipt represents a countersignature issued by a Transparency Service.

The Receipt structure is a CBOR array with two items, in order:

* service_id: The service identifier as tstr.

* contents: The proof as a CBOR structure determined by the tree algorithm.

Receipt = [service_id: tstr, contents: any]

Each tree algorithm MUST define its contents type and procedures for issuing and verifying a receipt.

4. COSE_Sign1 Countersigning

While the tree algorithms may differ in the way they aggregate multiple envelopes to compute a digest to be signed by the TS, they all share the same representation of the individual envelopes to be countersigned (intuitively, their leaves).

This document uses the principals and structure definitions of COSE_Sign1 countersigning V2 ([I-D.ietf-cose-countersign]). Each envelope is authenticated using a Countersign_structure array, recalled below.

Countersign_structure = [
  context: "CounterSignatureV2",
  body_protected: empty_or_serialized_map,
  sign_protected: empty_or_serialized_map,
  external_aad: bstr,
  payload: bstr,
  other_fields: [signature: bstr]
]

The body_protected, payload, and signature fields are copied from the COSE_Sign1 message being countersigned.
The sign_protected field is provided by the TS, see Section 4.1 below. This field is included in the Receipt contents to enable the Verifier to re-construct Countersign_structure, as specified by the tree algorithm.

By convention, the TS always provides an empty external_aad: a zero-length bytestring.

Procedure for reconstruction of Countersign_structure:

1. Let Target be the COSE_Sign1 message that corresponds to the countersignature. Different environments will have different mechanisms to achieve this. One obvious mechanism is to embed the Receipt in the unprotected header of Target. Another mechanism may be to store both artifacts separately and use a naming convention, database, or other method to link both together.

2. Extract body_protected, payload, and signature from Target.

3. Create a Countersign_structure using the extracted fields from Target, and sign_protected from the Receipt contents.

4.1. Countersigner Header Parameters

The following parameters MUST be included in the protected header of the countersigner (sign_protected in Section 4):

* Issued At (label: TBD): The time at which the countersignature was issued as the number of seconds from 1970-01-01T00:00:00Z UTC, ignoring leap seconds.

5. CCF 2 Tree Algorithm

The CCF 2 tree algorithm specifies an algorithm based on a binary Merkle tree over the sequence of all ledger entries, as implemented in the CCF version 2 framework (see [CCF_Merkle_Tree]).

5.1. Additional Parameters

The algorithm requires that the TS define additional parameters:

* Hash Algorithm: The hash algorithm used in its Merkle Tree (see Section 9.2.2).

* Signature Algorithm: The signature algorithm used (see Section 9.2.3).
* Service Certificate: The self-signed X.509 certificate used as trust anchor to verify signatures generated by the transparency service using the Signature Algorithm.

All definitions in this section use the hash algorithm set in the TS parameters (see Section Section 5.1). We write HASH to refer to this algorithm, and HASH_SIZE for the fixed length of its output in bytes.

5.2. Cryptographic Components

Note: This section is adapted from Section 2.1 of [RFC9162], which provides additional discussion of Merkle trees.

5.2.1. Binary Merkle Trees

The input of the Merkle Tree Hash (MTH) function is a list of n bytestrings, written D_n = \{d[0], d[1], ..., d[n-1]\}. The output is a single HASH_SIZE bytestring, also called the tree root hash.

This function is defined as follows:

The hash of an empty list is the hash of an empty string:

\[ MTH(\{} = HASH(). \]

The hash of a list with one entry (also known as a leaf hash) is:

\[ MTH(\{d[0]\}) = HASH(d[0]). \]

For \( n > 1 \), let \( k \) be the largest power of two smaller than \( n \) (i.e., \( k \) < \( n \) <= 2\( k \)). The Merkle Tree Hash of an \( n \)-element list \( D_n \) is then defined recursively as:

\[ MTH(D_n) = HASH(MTH(D[0:k])) \, || \, MTH(D[k:n])), \]

where:

* || denotes concatenation
* : denotes concatenation of lists
* \( D[k1:k2] = D’_{(k2-k1)} \) denotes the list \( \{d’[0] = d[k1], d’[1] = d[k1+1], \ldots, d’[k2-k1-1] = d[k2-1]\} \) of length \( k2 - k1 \).
5.2.2. Merkle Inclusion Proofs

A Merkle inclusion proof for a leaf in a Merkle Tree is the shortest list of intermediate hash values required to re-compute the tree root hash from the digest of the leaf bytestring. Each node in the tree is either a leaf node or is computed from the two nodes immediately below it (i.e., towards the leaves). At each step up the tree (towards the root), a node from the inclusion proof is combined with the node computed so far. In other words, the inclusion proof consists of the list of missing nodes required to compute the nodes leading from a leaf to the root of the tree. If the root computed from the inclusion proof matches the true root, then the inclusion proof proves that the leaf exists in the tree.

5.2.2.1. Verifying an Inclusion Proof

When a client has received an inclusion proof and wishes to verify inclusion of a leaf_hash for a given root_hash, the following algorithm may be used to prove the hash was included in the root_hash:

```python
recompute_root(leaf_hash, proof):
    h := leaf_hash
    for [left, hash] in proof:
        if left
            h := HASH(hash || h)
        else
            h := HASH(h || hash)
    return h
```

5.2.2.2. Generating an Inclusion Proof

Given the MTH input $D_n = \{d[0], d[1], ..., d[n-1]\}$ and an index $i < n$ in this list, run the MTH algorithm and record the position and value of every intermediate hash concatenated and hashed first with the digest of the leaf, then with the resulting intermediate hash value. (Most implementations instead record all intermediate hash computations, so that they can produce all inclusion proofs for a given tree by table lookups.)

5.3. Encoding Signed Envelopes into Tree Leaves

This section describes the encoding of signed envelopes and auxiliary ledger entries into the leaf bytestrings passed as input to the Merkle Tree function.

Each bytestring is computed from three inputs:
* internal_hash: a string of HASH_SIZE bytes;

* internal_data: a string of at most 1024 bytes; and

* data_hash: either the HASH of the CBOR-encoded
  Countersign_structure of the signed envelope, using the CBOR
  encoding described in Section 6, or a bytestring of size HASH_SIZE
  filled with zeroes for auxiliary ledger entries.

as the concatenation of three hashes:

LeafBytes = internal_hash || HASH(internal_data) || data_hash

This ensures that leaf bytestrings are always distinct from the
inputs of the intermediate computations in MTH, which always consist
of two hashes, and also that leaf bytestrings for signed envelopes
and for auxiliary ledger entries are always distinct.

The internal_hash and internal_data bytestrings are internal to the
CCF implementation. Similarly, the auxiliary ledger entries are
internal to CCF. They are opaque to receipt Verifiers, but they
commit the TS to the whole ledger contents and may be used for
additional, CCF-specific auditing.

5.4. Receipt Contents Structure

The Receipt contents structure is a CBOR array. The items of the
array in order are:

* signature: the signature over the Merkle tree root as bstr.

* node_certificate: a DER-encoded X.509 certificate for the public
  key for signature verification. This certificate MUST be a valid
  CCF node certificate for the service; in particular, it MUST form
  a valid X.509 certificate chain with the service certificate.

* inclusion_proof: the intermediate hashes to recompute the signed
  root of the Merkle tree from the leaf digest of the envelope.

  - The array MUST have at most 64 items.

  - The inclusion proof structure is an array of [left, hash] pairs
    where left indicates the ordering of digests for the
    intermediate hash computation. The hash MUST be a bytestring of
    length HASH_SIZE.
leaf_info: auxiliary inputs to recompute the leaf digest included in the Merkle tree: the internal hash, the internal data, and the protected header of the countersigner.

- internal_hash MUST be a bytestring of length HASH_SIZE;
- internal_data MUST be a bytestring of length less than 1024.

The inclusion of an additional, short-lived certificate endorsed by the TS enables flexibility in its distributed implementation, and may support additional CCF-specific auditing.

The CDDL fragment that represents the above text follows.

```cddl
ReceiptContents = [
  signature: bstr,
  node_certificate: bstr,
  inclusion_proof: [+ ProofElement],
  leaf_info: LeafInfo
]

ProofElement = [
  left: bool
  hash: bstr
]

LeafInfo = [
  internal_hash: bstr,
  internal_data: bstr,
  sign_protected: empty_or_serialized_map
]
```

5.5. Receipt Verification

Given the TS parameters, a signed envelope, and a Receipt for it, the following steps must be followed to verify this Receipt.

1. Verify that the Receipt Content structure is well-formed, as described in Section 5.4

2. Construct a Countersign_structure as described in Section 4, using sign_protected from the leaf_info field of the receipt contents.

3. Compute LeafBytes as the bytestring concatenation of the internal hash, the hash of internal data, and the hash of the CBOR-encoding of Countersign_structure, using the CBOR encoding described in Section 6.
LeafBytes := internal_hash || HASH(internal_data) || HASH(cbor(Countersign_structure))

4. Compute the leaf digest.

LeafHash := HASH(LeafBytes)

5. Compute the root hash from the leaf hash and the Merkle proof using the Merkle Tree Hash Algorithm found in the service’s parameters (see Section 5.1):

   root := recompute_root(LeafHash, inclusion_proof)

6. Verify the certificate chain established by the node certificate embedded in the receipt and the fixed service certificate in the TS parameters (see Section 5.1) using the Issued At time from sign_protected to verify the validity periods of the certificates. The chain MUST enable the use of the public key in the receipt certificate for signature verification with the Signature Algorithm of the TS parameters.

7. Verify that signature is a valid signature value of the root hash, using the public key of the receipt certificate and the Signature Algorithm of the TS parameters.

The Verifier SHOULD apply additional checks before accepting the countersigned envelope as valid, based on its protected headers and payload.

5.6. Receipt Generation

This document provides a reference algorithm for producing valid receipts, but it omits any discussion of TS registration policy and any CCF-specific implementation details.

The algorithm takes as input a list of entries to be jointly countersigned, each entry consisting of internal_hash, internal_data, and an optional signed envelope. (This optional item reflects that a CCF ledger records both signed envelopes and auxiliary entries.)

1. For each signed envelope, compute the Countersign_structure as described in Section 4.

2. For each item in the list, compute LeafBytes as the bytestring concatenation of the internal hash, the hash of internal data and, if the envelope is present, the hash of the CBOR-encoding of Countersign_structure, using the CBOR encoding described in Section 6, otherwise a HASH_SIZE bytestring of zeroes.
3. Compute the tree root hash by applying MTH to the resulting list of leaf bytestrings, keeping the results for all intermediate HASH values.

4. Select a valid node_certificate and compute a signature of the root of the tree with the corresponding signing key.

5. For each signed envelope provided in the input,

   * Collect an inclusion_proof by selecting intermediate hash values, as described above.
   
   * Produce the receipt contents using this inclusion_proof, the fixed node_certificate and signature, and the bytestrings internal_hash and internal_data provided with the envelope.
   
   * Produce the receipt using the Service Identifier and this receipt contents.

6. CBOR Encoding Restrictions

   In order to always regenerate the same byte string for the "to be signed" and "to be hashed" values, the core deterministic encoding rules defined in Section 4.2.1 of [RFC8949] MUST be used for all their CBOR structures.

7. Privacy Considerations

   TBD

8. Security Considerations

   TBD

9. IANA Considerations

9.1. Additions to Existing Registries

9.1.1. New Entries to the COSE Header Parameters Registry

   IANA is requested to register the new COSE Header parameters defined below in the "COSE Header Parameters" registry.

9.1.1.1. COSE_Sign1 Countersign receipt

   Name: COSE_Sign1 Countersign receipt

   Label: TBD
Value Type: Receipt / [+ Receipt]

Description: A COSE_Sign1 Countersign Receipt to be embedded in the unprotected header of the countersigned COSE_Sign1 message.

9.1.1.2. Issued At

Name: Issued At
Label: TBD
Value Type: uint

Description: The time at which the signature was issued as the number of seconds from 1970-01-01T00:00:00Z UTC, ignoring leap seconds.

9.2. New SCITT-Related Registries

IANA is asked to add a new registry "TBD" to the list that appears at https://www.iana.org/assignments/.

The rest of this section defines the subregistries that are to be created within the new "TBD" registry.

9.2.1. Tree Algorithms

IANA is asked to establish a registry of tree algorithm identifiers, named "Tree Algorithms", with the following registration procedures: TBD

The "Tree Algorithms" registry initially consists of:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Tree Algorithm</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCF-2</td>
<td>CCF 2 tree algorithm</td>
<td>This document</td>
</tr>
</tbody>
</table>

Table 1: Initial content of Tree Algorithms registry

The designated expert(s) should ensure that the proposed algorithm has a public specification and is suitable for use as [TBD].

9.2.2. Hash Algorithms

IANA is asked to establish a registry of hash algorithm identifiers, named "Hash Algorithms", with the following registration procedures: TBD
The "Hash Algorithms" registry initially consists of:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Hash Algorithm</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-256</td>
<td>SHA-256</td>
<td>[RFC6234]</td>
</tr>
</tbody>
</table>

Table 2: Initial content of Hash Algorithms registry

The designated expert(s) should ensure that the proposed algorithm has a public specification and is suitable for use as a cryptographic hash algorithm with no known preimage or collision attacks. These attacks can damage the integrity of the ledger.

9.2.3. Signature Algorithms

IANA is asked to establish a registry of signature algorithm identifiers, named "Signature Algorithms", with the following registration procedures: TBD

The "Signature Algorithms" registry initially consists of:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Signature Algorithm</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES256</td>
<td>Deterministic ECDSA (NIST P-256) with HMAC-SHA256</td>
<td>[RFC6979]</td>
</tr>
<tr>
<td>ED25519</td>
<td>Ed25519 (PureEdDSA with the edwards25519 curve)</td>
<td>[RFC8032]</td>
</tr>
</tbody>
</table>

Table 3: Initial content of Signature Algorithms registry

The designated expert(s) should ensure that the proposed algorithm has a public specification and is suitable for use as a cryptographic signature algorithm.

10. References

10.1. Normative References
10.2. Informative References

[CCF_Merkle_Tree]

[I-D.ietf-cose-countersign]
Authors’ Addresses

Henk Birkholz
Fraunhofer SIT
Rheinstrasse 75
64295 Darmstadt
Germany
Email: henk.birkholz@sit.fraunhofer.de

Maik Riechert
Microsoft
United Kingdom
Email: Maik.Riechert@microsoft.com

Antoine Delignat-Lavaud
Microsoft
United Kingdom
Email: antdl@microsoft.com

Cedric Fournet
Microsoft
United Kingdom
Email: fournet@microsoft.com