

Workgroup: TBD

Internet-Draft:

draft-birkholz-scitt-receipts-02

Published: 24 October 2022

Intended Status: Standards Track

Expires: 27 April 2023

Authors: H. Birkholz      M. Riechert      A. Delignat-Lavaud

Fraunhofer SIT      Microsoft      Microsoft

C. Fournet

Microsoft

**Countersigning COSE Envelopes in Transparency Services**

## Abstract

A transparent and authentic Transparent Registry service in support of a supply chain's integrity, transparency, and trust requires all peers that contribute to the Registry 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 registries. A generic procedure for producing payloads to be signed and validated 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 27 April 2023.

## 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. 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](#)
  - [1.1. Requirements Notation](#)
- [2. Common Parameters](#)
- [3. Generic Receipt Structure](#)
- [4. COSE\\_Sign1 Countersigning](#)
  - [4.1. Countersigner Header Parameters](#)
- [5. Receipt Verification](#)
- [6. CCF Tree Algorithm](#)
  - [6.1. Additional Parameters](#)
  - [6.2. Cryptographic Components](#)
    - [6.2.1. Binary Merkle Trees](#)
    - [6.2.2. Merkle Inclusion Proofs](#)
  - [6.3. Encoding Signed Envelopes into Tree Leaves](#)
  - [6.4. Receipt Contents Structure](#)
  - [6.5. Receipt Contents Verification](#)
  - [6.6. Receipt Generation](#)
- [7. CBOR Encoding Restrictions](#)
- [8. Privacy Considerations](#)
- [9. Security Considerations](#)
- [10. IANA Considerations](#)
  - [10.1. Additions to Existing Registries](#)
    - [10.1.1. New Entries to the COSE Header Parameters Registry](#)
  - [10.2. New SCITT-Related Registries](#)
    - [10.2.1. Tree Algorithms](#)
    - [10.2.2. Signature Algorithms](#)
- [11. References](#)
  - [11.1. Normative References](#)
  - [11.2. Informative References](#)
- [Authors' Addresses](#)

## 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 the Supply Chain Integrity, Transparency, and Trust (SCITT) architecture document (An Architecture for Trustworthy and Transparent Digital Supply Chains,

see [[I-D.birkholz-scitt-architecture](#)]): Claim, Envelope, Transparency Service, Registry, 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 parameters, if any. This document creates a registry (see [Section 10.2.1](#)) 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:

\*protected: The protected header of the countersigner.

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

```
Receipt = [  
  protected: bstr .cbor {  
    * label => value  
  },  
  contents: any  
]  
label = tstr / int  
value = 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 principles 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](#)):

\*Service ID (label: TBD): The Service identifier, as defined in the Transparency Service parameters.

\*Tree Algorithm (label: TBD): The Tree Algorithm used for issuing the receipt, as defined in the Transparency Service parameters.

\*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. Receipt Verification

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

1. Decode the protected header of the Receipt and look-up the TS parameters using the Service ID header parameter.
2. Verify that the Tree Algorithm parameter value in the receipt protected header matches the one in the TS parameters.
3. Construct a Countersign\_structure as described in [Section 4](#), using the protected header of the Receipt as sign\_protected.
4. CBOR-encode Countersign\_structure as To-Be-Included, using the CBOR encoding described in [Section 7](#).
5. Invoke the Tree Algorithm receipt verification procedure with the TS parameters and To-Be-Included as inputs.

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

## 6. CCF Tree Algorithm

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

### 6.1. Additional Parameters

The algorithm requires that the TS define additional parameters:

\*Signature Algorithm: The ECDSA signature algorithm used to sign the Merkle tree root (see [Section 10.2.2](#)).

\*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 required by the signature algorithm set in the TS parameters (see [Section 6.1](#)). We write HASH to refer to this algorithm, and HASH\_SIZE for the fixed length of its output in bytes.

## 6.2. Cryptographic Components

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

### 6.2.1. Binary Merkle Trees

The input of the Merkle Tree Hash (MTH) function is a list of `n` bytestrings, written  $D_n = \{d[0], d[1], \dots, d[n-1]\}$ . 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:

$MTH(\{\}) = \text{HASH}()$ .

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

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

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

$MTH(D_n) = \text{HASH}(MTH(D[0:k]) \parallel MTH(D[k:n]))$ ,

where:

$\parallel$  denotes concatenation

$*$ : denotes concatenation of lists

$*D[k_1:k_2] = D'_{(k_2-k_1)}$  denotes the list  $\{d'[0] = d[k_1], d'[1] = d[k_1+1], \dots, d'[k_2-k_1-1] = d[k_2-1]\}$  of length  $(k_2 - k_1)$ .

### 6.2.2. 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.

#### 6.2.2.1. 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:

```
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
```

#### 6.2.2.2. Generating an Inclusion Proof

Given the MTH input  $D_n = \{d[0], d[1], \dots, d[n-1]\}$  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.)

### 6.3. 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 7](#), 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.

#### 6.4. Receipt Contents Structure

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

- \*signature: the ECDSA signature over the Merkle tree root as bstr. Note that the Merkle tree root hash is the prehashed input to ECDSA and is not hashed twice.

- \*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 and the internal data.

  - 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.

```

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

```

```

ProofElement = [
    left: bool
    hash: bstr
]

```

```

LeafInfo = [
    internal_hash: bstr,
    internal_data: bstr
]

```

## 6.5. Receipt Contents Verification

Given the To-Be-Included bytes (see [Section 5](#)) and the TS parameters, the following steps must be followed to verify the Receipt contents.

1. Verify that the Receipt Content structure is well-formed, as described in [Section 6.4](#).
2. Compute LeafBytes as the bytestring concatenation of the internal hash, the hash of internal data, and the hash of the To-Be-Included bytes.

```
LeafBytes := internal_hash || HASH(internal_data) || HASH(To-Be-Include
```

3. Compute the leaf digest.

```
LeafHash := HASH(LeafBytes)
```

4. 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 6.1](#)):

```
root := recompute_root(LeafHash, inclusion_proof)
```

5. Verify the certificate chain established by the node certificate embedded in the receipt and the fixed service certificate in the TS parameters (see [Section 6.1](#)). TBD needs more details

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

## 6.6. 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, create the countersigner protected header and 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 7](#), 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 countersigner protected header and this receipt's contents.

## 7. CBOR Encoding Restrictions

In order to always regenerate the same byte string for the "to be included" 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.

## **8. Privacy Considerations**

TBD

## **9. Security Considerations**

TBD

## **10. IANA Considerations**

### **10.1. Additions to Existing Registries**

#### **10.1.1. 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.

##### **10.1.1.1. COSE\_Sign1 Countersign receipt**

Name: COSE\_Sign1 Countersign receipt

Label: TBD (temporary: 394, see also [[I-D.birkholz-scitt-architecture](#)])

Value Type: [+ Receipt]

Description: One or more COSE\_Sign1 Countersign Receipts to be embedded in the unprotected header of the countersigned COSE\_Sign1 message.

##### **10.1.1.2. 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.

### **10.2. 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.

#### 10.2.1. 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:

Identifier	Tree Algorithm	Reference
CCF	CCF tree algorithm	This document

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].

#### 10.2.2. 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:

Identifier	Signature Algorithm	Reference
ES256	Deterministic ECDSA (NIST P-256) with HMAC-SHA256	[ <a href="#">RFC6979</a> ]

Table 2: 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.

## 11. References

### 11.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234,

DOI 10.17487/RFC6234, May 2011, <<https://www.rfc-editor.org/info/rfc6234>>.

[RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <<https://www.rfc-editor.org/info/rfc6979>>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <<https://www.rfc-editor.org/info/rfc8032>>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <<https://www.rfc-editor.org/info/rfc8949>>.

[RFC9162] Laurie, B., Messeri, E., and R. Stradling, "Certificate Transparency Version 2.0", RFC 9162, DOI 10.17487/RFC9162, December 2021, <<https://www.rfc-editor.org/info/rfc9162>>.

## 11.2. Informative References

[CCF\_Merkle\_Tree] Microsoft Research, "CCF - Merkle Tree", n.d., <[https://microsoft.github.io/CCF/main/architecture/merkle\\_tree.html](https://microsoft.github.io/CCF/main/architecture/merkle_tree.html)>.

[I-D.birkholz-scitt-architecture] Birkholz, H., Delignat-Lavaud, A., Fournet, C., and Y. Deshpande, "An Architecture for Trustworthy and Transparent Digital Supply Chains", Work in Progress, Internet-Draft, draft-birkholz-scitt-architecture-01, 7 September 2022, <<https://www.ietf.org/archive/id/draft-birkholz-scitt-architecture-01.txt>>.

[I-D.ietf-cose-countersign] Schaad, J. and R. Housley, "CBOR Object Signing and Encryption (COSE): Countersignatures", Work in Progress, Internet-Draft, draft-ietf-cose-countersign-10, 20 September 2022, <<https://www.ietf.org/archive/id/draft-ietf-cose-countersign-10.txt>>.

## Authors' Addresses

Henk Birkholz  
Fraunhofer SIT

Rheinstrasse 75  
64295 Darmstadt  
Germany

Email: [henk.birkholz@sit.fraunhofer.de](mailto:henk.birkholz@sit.fraunhofer.de)

Maik Riechert  
Microsoft  
United Kingdom

Email: [Maik.Riechert@microsoft.com](mailto:Maik.Riechert@microsoft.com)

Antoine Delignat-Lavaud  
Microsoft  
United Kingdom

Email: [antdl@microsoft.com](mailto:antdl@microsoft.com)

Cedric Fournet  
Microsoft  
United Kingdom

Email: [fournet@microsoft.com](mailto:fournet@microsoft.com)