Coalition for
Content Provenance
and Authenticity

Attestation in the C2PA Framework

1.0, 2023-09-14: Release

Table of Contents

Chapter 1. Introduction

Attestation is a platform security feature that allows software to prove its identity and the identity and/or security characteristics of the device or platform on which it is running.

The current C2PA trust model is primarily designed to enable persons or organizations to make claims about digital assets and their associated metadata. Attestation extends the C2PA trust model to allow relying parties to determine the software and devices that are involved in the creation or processing of a digital asset. These additional trust signals can supplement or replace the person/organization-based trust signals that are currently defined.

This specification describes how to add attestation signals and measurements to C2PA-marked assets. C2PA seeks feedback on this proposal, especially from implementers.

Attestation is a rapidly evolving field. At the time of writing, there are several widely deployed but non-interoperable technologies and protocols. There are also several initiatives to improve interoperability, but all current proposals involve new trust-broker-style services, which which are under development. This specification takes a pragmatic approach: it defines general-purpose data structures that support current technologies and known upcoming standards and systems.

1.1. Outline

This document is organized as follows.

1. Chapter 2 contains a glossary of attestation-related terms used in this specification. This document also uses terms defined in the C2PA specification C2PA Glossary.
2. Chapter 3 and Chapter 4 describe the existing C2PA trust model and provide an introduction to attestation.
3. Chapter 5 describes some motivating scenarios using different types of attestation technology.
4. Chapter 6 and Explicit Attestation are the main normative parts of the specification. These chapters describe two different approaches to adding attestation to C2PA data structures. The architecture and data structures described in these chapters are attestation-technology-agnostic. Appendix A provides information on how the currently supported attestation technologies can be used, and how the resulting attestations are encoded.
5. Chapter 8 contains considerations for how C2PA Trust Roots should be supplemented to enable interoperable attestation creation and verification.
6. Chapter 9 provides guidance for selecting an attestation technology and choosing how to encode the attestation evidence.

This specification assumes a working understanding of the C2PA architecture and data structures: in particular, Claims, Assertions, Claim Signatures and how these data objects are packaged into C2PA manifests. This specification also assumes some understanding of attestation technologies.

Chapter 2. Glossary

This chapter contains a glossary of terms used in this document. Additional C2PA-related terms are defined in C2PA Glossary.

Attestation

Attestation (verb) is the process of making claims about the security properties of a device, program, or execution environment. In the context of this specification, attestation is built on digital signatures from trusted attestation providers. An attestation (noun) is security claim about a program or device.

Attestation Provider

A trustworthy local or remote entity that generates Attestations. A local attestation provider is typically a Root of Trust on a device. A remote Attestation Provider is called a Trust Broker in this specification.

Attestation Evidence

An attestation generated by a local Attestation Provider - i.e., a Root of Trust on the device. Attestation Evidence can be directly consumed by C2PA Validators or or by Trust Brokers.

Attestation Verdict.

An Attestation Verdict is the information returned by a Trust Broker, e.g., a RATS Verifier, after the appraisal of Attestation Evidence. In RATS terminology it is known as Attestation Results.

EAT

An Entity Attestation Token (EAT) is a claims set that describes state and characteristics of a hardware/software component. EATs are used by relying parties, servers or services to determine how much they wish to trust the entity. An EAT is either a CBOR Web Token (CWT) or JSON Web Token (JWT) with attestation-oriented claims. For more detail see: The Entity Attestation Token.

Explicit Attestation

Explicit Attestations are signatures over user-provided data and claims about the security state of the platform or application, combined to form an evidence. Platforms doing explicit attestation contain a Root of Trust with a signing key that signs data structure containing application-provided data (for example, the hash of a C2PA Claim) together with measurements/statements of the application and the platform environment where it is running.

Implicit Attestation (Key Attestation)

Implicit Attestations are digital signatures using keys that are only available to trusted devices and applications. Because only trusted applications can use these keys, a digital signature using one of the keys *implies* the identity of the application processed the Claim. The keys used for implicit attestation can be embedded during manufacture or can be created/certified in the field. In the latter case, Explicit Attestations are often used to prove that Implicit Attestation keys are being managed by trusted software on known or authorized hardware.

Quote

A Quote is the name given to the raw attestation evidence provided by TPMs, SGX Enclaves, and many other attestation providers. Quotes are digital signatures over user-supplied data (a Claim-hash, in this setting) as well as measurements of the application/execution environment that requested the quote-operation.

Sealing

Sealing is a security primitive that allows an application to limit the application and devices that can access the sealed data. In this specification, Sealing can be used to ensure that only authorized applications can access keys used for Implicit Attestation.

Remote Attestation procedureS (RATS)

The Remote ATtestation ProcedureS (RATS) is a framework and architecture for unifying and simplifying the use of attestation in distributed systems. RATS defines a trusted intermediary, called a RATS Verifier (A Trust Broker, in the terminology of this specification) to hide the complexity of attestation primitives and policies from relying parties. RATS is defined in: RATS Architecture.

Root of Trust

A Root of Trust is a hardware or software component that is trusted to perform a critical security function. Roots of Trust are typically used to vouch for the device or software running on the device: i.e., to generate attestations.

TPM / TPM 2.0

A Trusted Platform Module (TPM, also known as ISO/IEC 11889) is a widely adopted international standard for a Root of Trust. TPMs can be implemented as discrete components, can be integrated into SoCs, or can be implemented in firmware running in a protected mode or environment on an application processor. See: TPM 2.0 Library | Trusted Computing Group

Trust Broker

Trust Broker is the generic term for an attestation verification service. IETF RATS Verifiers and Google Play Integrity services are examples of Trust Brokers. Trust Brokers evaluate Attestation Evidence produced by Local Attestation Providers and return Attestation Verdicts which are incorporated into C2PA Manifests.

Chapter 3. C2PA Trust Model Overview

A C2PA Claim Signature is a simple digital signature over a CBOR-serialized C2PA Claim. This signature allows relying parties to verify the signer, and that the claim (and referenced assertions, and the asset) have not been modified. In the current version of the specification, Claim Signers-keys are supported by an X.509-based PKI. C2PA does not specify trust roots: these are defined by other entities.

C2PA also supports optional time-stamping. A C2PA Timestamp is a countersignature over the CBOR-serialized C2PA Claim: effectively, the Claim is signed twice: once by the Claim Signer, and once by the time-stamping service on behalf of the Claim Signer.

In the current version of the C2PA main specification, Claim Signing keys and their associated certificates can be associated with people, organizations or devices. This specification describes how the existing C2PA Claim Signing keys can be managed so that the resulting signature conveys information about the Claim Signer and characteristics of the security environment in which it is running. This is called *Implicit Attestation* (Chapter 6).

This specification also describes how evidence from attestation providers can be added to a Claim with a normal Claim signature (Explicit Attestation).

Chapter 4. Attestation Overview

A well-formed C2PA Claim Signature demonstrates that the Claim Creator had access to the Claim Signing key when the Manifest was created. If the Claim Signing key can only be used by a single entity, then the Claim Signature demonstrates that the Claim Creator was that entity. In most cases, access to Claim Signing Keys (or any key that is used to create digital signatures) is carefully controlled. For example, an enterprise might only allow certain trusted employees to sign assets. This allows relying parties to infer that a properly signed asset was published by that organization.

Attestation, in the context of C2PA, seeks to refine this Trust Model to allow relying parties to infer more information about the *device or application* that created Claim and Manifest. The additional information might include:

1. The specific device that created the Claim.
2. The *class* of device that created the Claim, without revealing the specific device (to preserve privacy.)
3. The application that created the Claim. This might be a specific application identified by a cryptographic digest, a family of applications identified by a signing key or certificate, or a publisher, identified by a unique name in the context of an app-store.
4. The security properties of the environment in which the app is running. This might include the operating system that is hosting the application as well as its security configuration (e.g. non-rooted or no admin-access.) In some cases, the security environment is provided by the CPU vendor with little or no additional software, in which case the security properties might included whether the "enclave" is debuggable.

This additional information may supplement or replace the information conventionally conveyed by a Claim Signature (i.e., the organization or person that created the Claim.)

As previously noted, this specification describes two approaches to conveying attestation signals, called *Implicit* and *Explicit* attestations.

An Implicit Attestation is a conventional Claim Signature, but using a key that is only accessible to authorized platforms and applications. This is described in Chapter 6. [1]

An Explicit Attestation is a special sort of signature that conveys evidence about the current security posture of the platform or application that requested the signature. Explicit Attestation-capable platforms contain a special sort of signing key, usually called an attestation key. Attestation keys cannot be used to generate simple signatures; instead they sign a combination of user/application-supplied data, together with measurements (or other sort of description) of the application that requested the signature and information about the environment in which it is running. Most platforms that provide attestation capabilities refer to these specialized signatures as *quotes* and call the signing operation *quoting*.

Explicit Attestation works slightly differently than Implicit Attestation does. Attestation-capable platforms provide a special sort of signing key, usually called an attestation key, that can be used by more than one application/operating system. However, if the attestation key is used to sign data, then the digital signature also conveys measurements of the application that requested the signature. Most platforms that provide attestation capabilities refer to these specialized signature as *quotes* and call the signing operation *quoting*.

For example:

- TPM quote-operations are a signature over some application-provided data together with an encoding of current Platform Configuration Registers (PCR) values. Compliant platforms record measurements of the booting software into the PCRs, so the TPM quote reports reflects the software running on the platform.
- Android Play Integrity is a system/cloud-service that allows app-developers to generate signed statements indicating the package that requested the integrity verdict, as well information about the security posture of the requesting device - e.g., the running operating system.
- Intel’s Software Guard Extensions (SGX) provides a reporting/quoting feature that is a signature over the measurement of the code that was loaded into the enclave, as well as other critical security settings.
- ARM’s Platform Security Architecture (PSA) provides the calling application with attestation tokens that contain security claims about the underlying platform.

Attestations can be directly consumed by relying parties or can be forwarded to a Trust Broker (a Verification Service) where they are translated into an easier-to-understand form, sometime called an Attestation Verdict. For example, Android Play Integrity verdicts use a Google-provided Trust Broker service.

This specification allow Explicit Attestations in the form of quote-style signatures to be embedded directly into Claims. This is described in Explicit Attestation.

Keys for Implicit Attestation may be provisioned at the time of device manufacture, but more commonly, quote-style attestations are used to enrol and certify Implicit Attestation keys. The advantage of this is that new keys can be enrolled when the device is in the field. This specification does not define suitable protocols, but general considerations secure key enrolment/provisioning are described in Section 6.2.

4.1. Trust Brokers and RATS

RATS, or Remote Attestation Procedures, is an IETF specification for a Trust Broker that seeks to improve interoperability between differing attestation systems (https://datatracker.ietf.org/doc/html/rfc9334). In RATS, the trusted service is called The *Verifier*, and acts as a trusted intermediary between The *Attester* (the device or service that creates a C2PA asset) and the *Relying Party* that seeks to make a trust assessment about a program or environment.

C2PA specifications use the terms *Validator* and *Verifier* for relying parties. To avoid confusion, this specification uses the term *Trust Broker* to indicate a RATS Verifier or similar service such as Google’s PlayIntegrity.

Trust Brokers are an important building block for enabling an open ecosystem of attestation-enabled asset creators and attestation-aware relying parties. Trust Brokers must be aware of a diverse set of platform types, applications, operating systems, and security policies. But if their policy requirement are met, then they can translate the raw-attestations into much simpler claims for C2PA validators (e.g., "Trusted Camera App running on Android 11.")

The expectation is that RATS-style Trust Brokers will simplify adoption of attestation technologies by open ecosystems. However, at the time of writing, the RATS ecosystem is still in its infancy. Because of this, current specification supports both raw attestation evidence (e.g., "quotes") as well as RATS-style "verdicts."

[1] Implicit Attestations can also be encoded in assertions, as described in Section 7.5

Chapter 5. Scenarios

This specification allows for a wide variety of attestation technologies to be used on a range of platforms for a wide variety of scenarios. However, the specification present two use-cases that informed the architecture and design of this specification.

5.1. Trusted Camera Application on an Android Phone

Sophisticated and easy to use media editing tools as well as synthetic image generation technologies are widely available. The existence of these tools degrades the probative value of all digital imagery.

Attestation can be used to distinguish imagery captured by a trusted camera and imagery created or manipulated by other applications and services. While no system is foolproof and immune from security threats, for some scenarios an attestation can markedly increase the likelihood that an image is of a real scene.

For example, a Trusted Camera application can use attestation to

1. Indicate the application that captured the image,
2. Implicitly prove that an image was obtained from the sensor,
3. Indicate that the phone operating system is up-to-date and configured securely, and
4. Indicate the type of phone or other device that was used.

Android Play Integrity is a Google service for obtaining such attestations.

5.2. AI/ML Models running in an Enclave or Isolated Virtual Machine

Machine learning models are increasingly responsible for security-critical tasks. Training sets, models, and inferences will all benefit from C2PA-style provenance signals and protection. In some cases, ML models execute in environments with poor or unknown security characteristics. In these cases, attestations can be added to demonstrate that a model or inference was trained or run in a secure environment.

For example, consider an ML-based image recognition model. The model performs recognition tasks and then outputs a C2PA-signed version of the asset with manifest metadata that identifies the type and location of the recognized objects.

If the image-recognition model can prove that it is running in a secure environment, then its outputs will be more trustworthy than the same model running in an open environment with poor security characteristics. This type of evidence can be included in C2PA manifests using the attestation technology defined in this specification.

Chapter 6. Implicit Attestation

This section describes the design considerations and normative requirements for incorporating implicit attestations into the C2PA trust model. The chapter than follows - Explicit Attestation - describes Explicit Attestation.

Implicit Attestation (sometimes called Key Attestation), in the context of this specification, involves the use of a signing key that can only be used by an authorized application running on an authorized device. If the signing key can only be used by the authorized application on the authorized devices, then the presence of a well-formed signature in a manifest *implies* that the authorized application signed the Claim. I.e., the Claim-Signing application has been *implicitly attested*. Keys with these security properties are called Implicit Attestation keys, or *IA-keys*.

The IA key is used to sign a Claim in exactly the same way as Claims are normally signed. I.e., the signature is calculated over the serialized Claim, and the resulting Claim Signature is packaged in the manifest identically to other Claim Signatures.

This specification also allows an IA-key signature to be embedded in an attestation assertion as an alternative to incorporating it as a Claim Signature. This is described in Section 7.5.

6.1. Implicit Attestation Requirements

The current (1.3) version of the C2PA specification supports the creation and validation of Implicit Attestation Claim Signatures without changes:

> Identity of Signers
>
> The identity of a signatory is not necessarily a human actor, and the identity presented may be a pseudonym, completely anonymous, or pertain to a service or trusted hardware device with its own identity, including an application running inside such a service or trusted hardware.

IA-keys and certificates are generally associated with devices, as opposed to people or organizations. If required, Claim Creators may use the `CreativeWork` assertion to encode a person or organization that was involved in creating a Claim. Alternatively, the conventional Claim Signature (associated with the person or organization) can be retained, and the Implicit Attestation can be encoded in an assertion, as described in Section 7.5.

6.2. Issuing / Certifying Keys for Implicit Attestation

This specification provides guidance but no normative requirements for the issuance and lifetime management of Implicit Attestation keys. Implementers should consult platform-specific documentation for detailed protocol descriptions.

Devices that support attestation provide cryptographic building blocks that can be used in protocols to prove the identity of the device and running software to relying parties. One common cryptographic primitive is usually called

Quote. Quotes are digital signatures over user-supplied data and platform/quote-engine-supplied data that describes the running program and the security properties of the environment in which it is executing.

Quotes (and related primitives) can be used to create Explicit Attestations (Explicit Attestation) but can also be used in a protocol to provision or certify an Implicit Attestation key. A commonly used protocol fragment is that an application creates a key-pair and “Quotes” the resulting public key. The public key and quote is sent to a trusted service that decodes the quote to check that the device is known or in good-standing, and that the program measurements conveyed in the quote are in policy. If the checks succeed, the service creates a certificate for the newly created Implicit Attestation public key.

(Note that this description is simplified: implementers should consult platform documentation for recommended provisioning protocols for Implicit Attestation keys.)

Once provisioned, the Claim Creator uses the corresponding private key to sign the serialized Claim (this chapter) or sign a serialize Partial Claim (Section 7.5).

6.3. IA-Key Certificates

The certificate for the implicit attestation claim signing key should comply with the requirements in the main C2PA specification. Optionally, the certificate may contain additional fields that convey additional information about the application or security posture of the device to which it was provisioned.

6.4. Protecting Implicit Attestation Signing Keys

Most platforms that provide attestation capabilities also provide security primitives to allow a system to protect stored keys and other data so that the data is only accessible to the attesting application, or other applications explicitly authorized by the attesting application. This operation is commonly called *sealing*.

Implementers should consult platform documentation for usage and the expected strength of protection. If suitable sealing primitives are not available, applications should not persist IA-keys: instead they should re-provision on every app-startup (or use Explicit Attestation).

6.5. Timeline for Provisioning and Use of Implicit Attestation Signing Keys

Summarizing the previous sections in the form of a timeline, the following steps are required when provisioning and using a key for Implicit Attestation.

1. It is assumed that devices are provisioned with a platform key and certificate (or a platform key and a pre-established trust relationship with a server.) This step is usually performed during manufacture.
2. At some point in the life of the device, the C2PA-compliant Claim Creator application is installed.
3. During installation (or later), the Claim Creator will interact with a trusted service to create a certified key known

as "certified IA-key" for signing Claims. The certified key is derived from platform key as its root. The Claim Creator uses an Attestation Protocol to prove that it is an authorized application (permitted to use the certified key) running on a trusted device. Hence the Application is now directly or indirectly associated to the platform key in (1).

4. The certified IA-key and certificate can be used immediately to sign Claims or Partial Claims.
5. If the platform provides suitable *sealing* facilities, the IA-private key can be persisted in such a way that only the authorized application can retrieve it.
    a. If this is the case, then the IA-key can be stored and protected so that the application retain the same key, during multiple restarts of the same application.
    b. If this is not the case, then the IA-key should not be persisted: instead, the application should follow step (3) to obtain a new IA-key on each startup.

6.6. Validation of Claims Using Implicit Attestation

Claims signed using Implicit Attestation Keys are validated identically to Claims signed using keys associated with people or organizations. I.e. the validation procedure is exactly as described in the main C2PA specification, although a different set of Trust Roots may be required or desired.

Chapter 7. Explicit Attestation

This section describes how an attestation can be created and embedded as an assertion. This contrast with the architecture described in the Implicit Attestation section where the (implicit) attestation is conveyed as a Claim Signature. Implementation considerations for choosing Implicit vs. Explicit Attestation is discussed in section Section 7.1.

The current C2PA trust model is built using two signatures:

1. A signature over a CBOR-serialized Claim. The signature is encoded in the Manifest, and the certificate chain for the signing key is usually also included.
2. Optionally, a countersignature from an RFC 3161-compliant time stamping service.

Attestations, in the context of this document, are also digital signatures. This section describes options for how the additional attestation signature(s) can be added to the two that are already defined. Note that the discussion in this section is simplified: later sections present the complete design and normative requirements.

In the following, when a statement say *"attestation over data X"* it implies that the attestation is a signature over the hash of the serialization of X, and also attestation-engine-supplied measurements or claims about the platform and/or application that requested the attestation.

The simplest option for adding an attestation would be to include it as a second countersignature over the C2PA Claim (as data X above). However, with this design, the Claim Signature or the attestation can be independently removed and replaced. Whether these attacks are troublesome will depend on scenario (and the sophistication of the C2PA Relying Party), which makes a thorough security analysis difficult. However, the following proposal prevents the attestation and claim signatures being independently replaced with only modest complexity.

To achieve the Claim-Signature/attestation binding, the following procedure is used:

1. An attestation over the serialized *partial C2PA Claim* is performed first and then the claim signature is performed over the complete C2PA Claim that contains attestation. This prevents the attestation being removed or replaced independently of the Claim Signature, because any modification of the attestation will invalidate the Claim Signature
2. To prevent the Claim Signature being replaced, the attestation can optionally contain the public key of the subsequent Claim Signer. This means that validators can detect and reject a Claim Signature if the Claim Signature key specified in the attestation does not match the keys used for the actual Claim Signature.

To summarize: The Claim Signer signs the attestation, which prevents tampering of the attestation. The attestation contains the public key of the Claim Signer, so the Claim Signature cannot be replaced with a signature using a different key.

NOTE There are no cryptographic protections against both the attestation and Claim Signature being stripped or replaced together.

The binding is illustrated below.

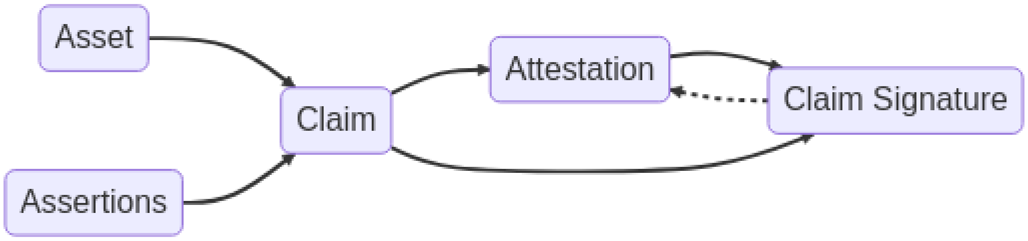


Figure 1. Cryptographic dependencies

The above figure shows cryptographic dependencies for an Asset, Claim, Attestation and Claim Signature. An arrow pointing from A to B means that B cryptographically depends on A, so A is effectively integrity protected. (In subsequent sections Partial Claims will be introduced, which changes the diagram but not the dependencies.)

7.1. What is Attested?

Claim Signatures are calculated over the serialization of a Claim, and the Claim itself is cryptographically linked to its embedded Assertions. Two types of Assertion are important here: Assertions that contain metadata, and Assertions that bind a Claim to an Asset with a hash or hashes of an Asset or Asset fragment (a "hard binding"). By signing the Claim, the Claim Creator vouches for both the asset (via the binding Assertion) and the metadata.

Similar bindings exists for the explicit attestation signature: i.e., the attestation vouches for the Asset and the metadata. Note that most or all scenarios benefit from binding to the Asset (e.g., for the Trusted Camera Application, *“this image was captured by this program running on this device.”*) Some scenarios will benefit from binding to metadata assertions (e.g., for a Trusted Camera Application *“the GPS coordinates obtained from the radio when this image was captured”* or for the ML scenario, *“the following objects were recognized in the image.”*) However, note that attestation will *not* improve trust for all types of assertion data: for example, user-input is hearsay as far as the attesting application is concerned.

The approach in this specification is that the attestation is over the serialization of the entire Claim – i.e., the same data structure that the Claim Generator signs (with one exception, which is described below). As noted above, not all Assertions/metadata gain security benefit from the attestation, but signing all assertions provides the most flexibility for future scenarios. For future advanced use cases, the specification defines a field that an attestation generator can use to indicate which assertions are meaningfully “attested.” However, the use and interpretation of this field is beyond the current scope of C2PA.

7.2. Embedding an Explicit Attestation in a Manifest

Explicit attestations are similar to Claim Signatures and could be encoded in a manifest similarly. However, there is also an additional security requirement noted earlier: i.e., that the Claim Signer should also sign the attestation result.

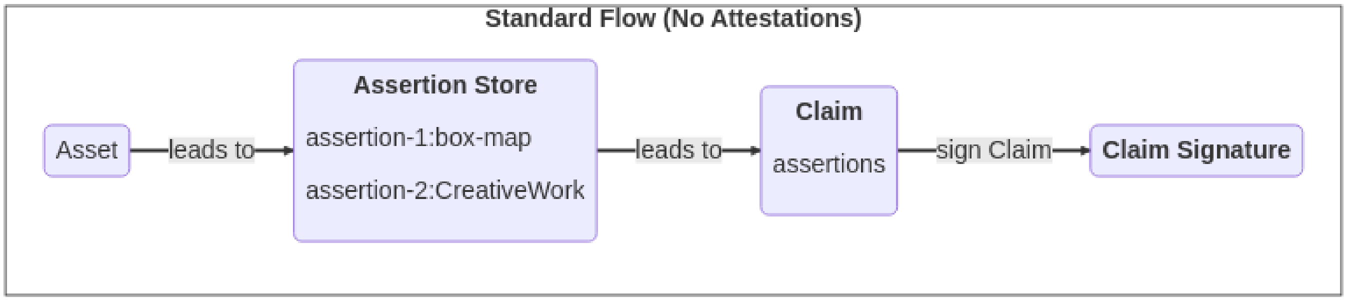
A natural way of encoding the attestation so that it is signed by the Claim Signer is to embed it as an assertion in the Claim. However with this method, then the attestation will naturally be signed in the same way as any other Assertion.

However, naïve attempts at implementing this method will fail, as it introduces a circular dependency: the Claim cannot be finalized before the attestation is created, but the attestation requires a finalized Claim in order to calculate the Claim hash.

The specification break this dependency by specifying that the attestation is created over the hash of the serialized claim but **omitting the attestation assertion** (which has not been created yet.) Once the attestation has been created, it is embedded in the Manifest and Claim identically to any other assertion and will subsequently be signed by the Claim Creator. The Claim with one or more attestation assertions elided is called a *Partial Claim*.

This architecture is attractive because normal Claim creation and Claim Validation are unaffected. It is also attractive because an attestation-enhanced claim can be processed by an attestation-unaware validator without changes (the attestation assertion can be treated like any other third-party or unrecognized assertion.)

The (simplified) logical work flow for creating a manifest with and without an attestation is illustrated below.



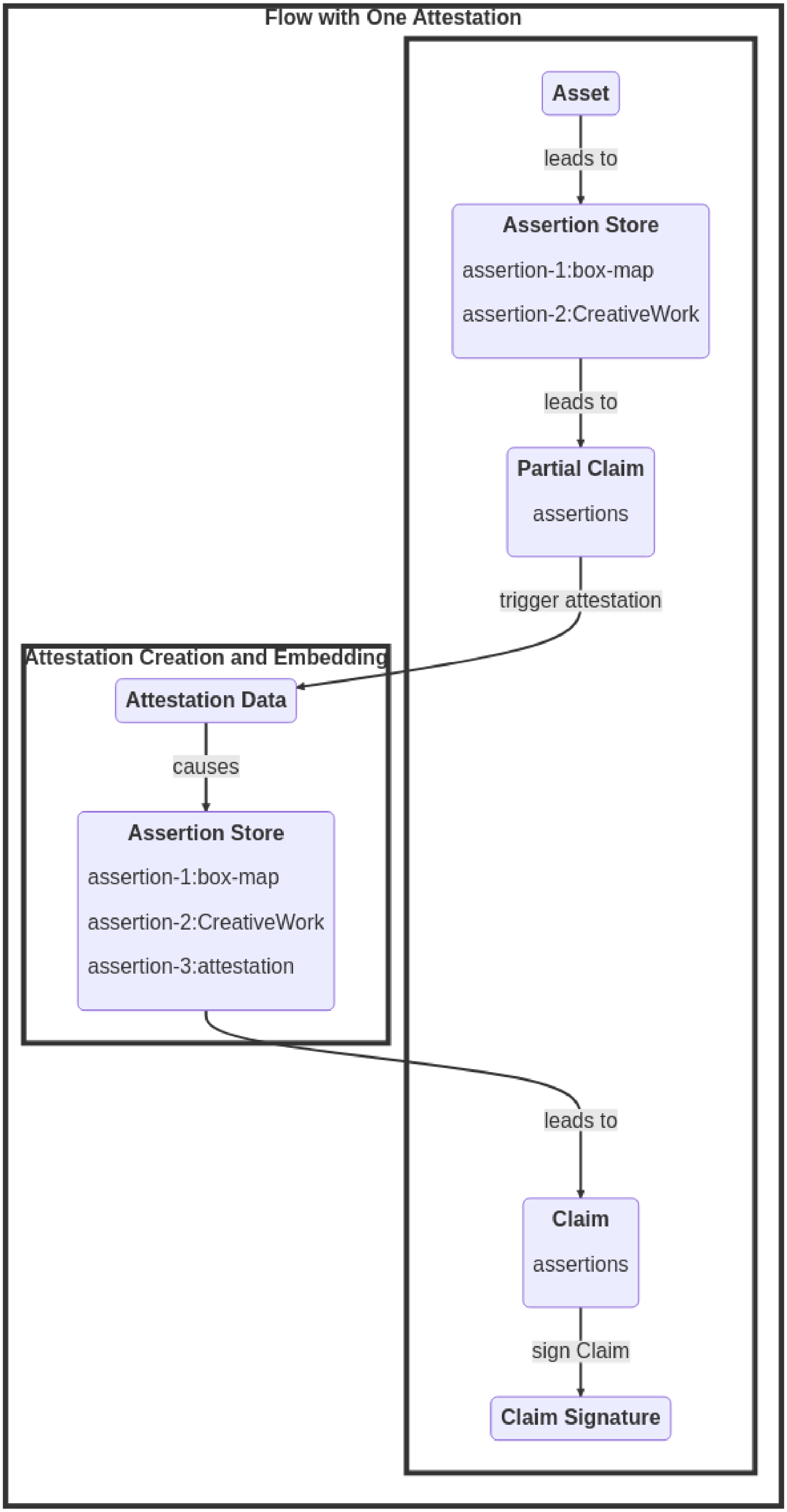


Figure 2. Simplified Steps

The above figure shows simplified steps for creating a C2PA Manifest containing an Attestation Assertion. In the no-attestation case, Assertions are created and stored in the Assertion Store. Next, a Claim is then prepared, with the `assertions` array set to the location and hash of the referenced assertions. The Claim is then serialized, hashed, and signed by the Claim Generator, yielding the Claim Signature. The Assertion Store, Claim, and Claim Signature are then packaged as a C2PA manifest (not shown.)

To support explicit attestations, the additional steps in the shaded box are required. As before, an Assertion Store is populated with the desired assertions, but it is referred to as a Partial Claim because one additional Assertion will be added before the Claim is finalized. The Partial Claim is then serialized and the Partial-Claim's hash (which is generated using the same methodology as a standard Claim) is then attested using the appropriate platform attestation service. Next, the Attestation is packaged as an Attestation Assertion and added to the Assertion Store to create a finalized Assertion Store. Finally, the complete Claim (with the embedded Attestation Assertion) is signed by the Claim Generator.

This architecture requires that attestation-aware validators perform additional steps: Specifically, validators must internally modify the Claim to remove any attestation assertions to produce a Partial Claim that can then be used to validate the bindings in the attestation assertion. This tradeoff is considered to be acceptable: the rewriting cost is borne by the creators and consumers of attestations, but the creation and validation workflow for non-attestation-aware entities is unchanged.

Additional details are included in the normative parts of the specification below.

7.3. Cryptographic Dependencies for the Asset, Partial Claim, and Claim Signature

The diagram below is a more detailed version of dependencies using Partial Claims

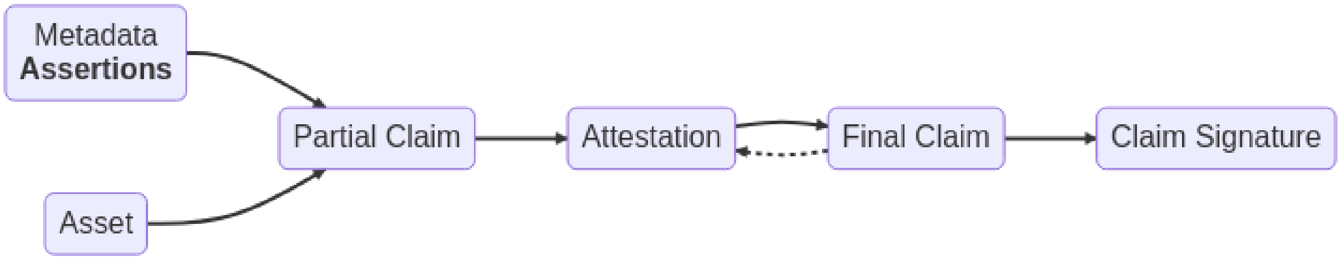


Figure 3. Cryptographic dependencies

The above figure shows cryptographic dependencies for the asset, Partial Claim, and Claim.

As before, an arrow from A to B indicates that A cannot be changed without invalidating B and hence the manifest. Practically, this means that the attestation and Claim Signature can both be stripped and replaced, but neither can be changed independently.

7.4. Multiple Explicit Attestations

This specification supports more than one attestation for a Claim – for example, there may be one attestation for code running in an enclave, and a second for the overall platform (the “rich OS.”)

If more than one attestation is required then they are ordered and each subsequent attestation is over the Partial Claim containing the prior attestations. For example, if two attestations are required, the first attestation to be added will be over the Partial Claim with no attestation assertions. The first attestation assertion is then added to the Claim, and the resulting data structure becomes the user data input (partial Claim) for the second attestation. Of course, the second attestation assertion is then added, and the resulting finalized Claim is signed by the Claim Creator.

The cryptographic dependencies are illustrated below. The manifest is built from the top to the bottom. An arrow from A to B indicates that B has a cryptographic dependence on A.

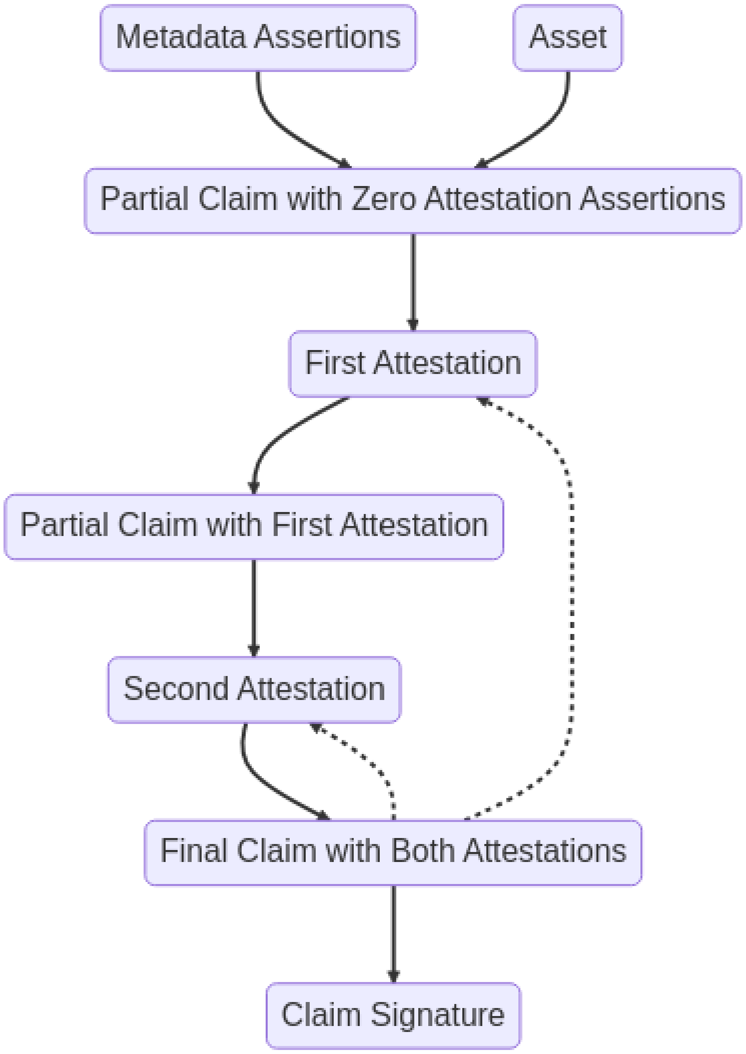

Metadata Assertions
Asset
Partial Claim with Zero Attestation Assertions
First Attestation
Partial Claim with First Attestation
Second Attestation
Final Claim with Both Attestations
Claim Signature

Figure 4. Cryptographic dependencies.

The above figure shows cryptographic dependencies for a manifest with two explicit attestations.

Attesters that wish to include multiple attestations are free to decide the order that they are created and embedded, but since later attestations include earlier attestations, the validation order must be the same as the creation order. To ensure that that this occurs, this specification requires that attestations are embedded in the assertions array in the order that they are created.

7.5. Implicit Attestation Assertions

Chapter 6 describes Implicit Attestation in detail, but briefly, an Implicit Attestation is a simple signature (i.e., not a *"quote"*) using a key that is only accessible to authorized applications running on authorized devices. If the key can only be used by trusted applications, then the presence of a well-formed Claim Signature *implies* that the trusted app signed the Claim.

Section 6.1 describes how Implicit Attestations can be encoded as Claim Signatures. This is simple and straightforward, but C2PA does not support multiple Claim Signatures in a Manifest so may not be suitable when the Claim Creator needs to convey both the (organization- or human-) Claim Signer *and* the application+device on which the Manifest was created.

To remedy this, the specification also allows Implicit Attestations to be encoded in Attestation Assertions, in exactly the same way that Explicit Attestations are supported. When using this option, rather than using a quote-style signature, the signature is created using an Implicit Attestation key: e.g., a key that is associated with a trusted application running on a trusted device.

Attestation Assertions for Implicit Attestations are distinguished using the `type` field. Implicit Attestations use the attestation type `c2pa.embedded-implicit` and the actual signature is embedded in the `attestation-info-map.att-result` field. Details are provided in Appendix A.

7.6. Normative Requirements for Explicit Attestations

7.6.1. Data Structures

7.6.1.1. Partial Claim Definition

A Partial Claim is a C2PA `claim-map` but omitting any Attestation Assertions. (Note: for simplicity it is stated that an assertion is “included” or “omitted”, but what is actually added or omitted from the Claim is actually a `hashed-uri-map` link in the `assertions` array). Partial Claims use the same `claim-map` data structure as a standard Claim: a Partial Claim only differs in that one or more Attestation Assertions are removed from the `assertions` array.

Attestation Assertions can be included at any location in the assertion array, although placing them last in the assertions array simplifies processing and is therefore preferred.

During Claim creation, the attestations are calculated then embedded one at a time. For example, the Partial Claim without attestation assertions is created, serialized, and then the first attestation is gathered. The resulting attestation result is then encoded as an `attestation-info-map` assertion and then added to the end of the `assertions` array in the Partial Claim form the finalized Claim.

If a second attestation is required, the Partial Claim with the first Attestation Assertion included is serialized and the second attestation is performed. The second Attestation Assertion is encoded in a `attestation-info-map` then added to the end of the `assertions` array to form the final Claim, which is then signed by the Claim Creator.

This specification does not restrict the location in the `assertions` array where the Attestation Assertions are added,[1] although, as previously noted, incorporating them last in the array is preferred. However, the order in the `assertions` array is important when more than one Attestation Assertion is incorporated. In this case, higher-index Attestation Assertions are performed after lower-index entries.

7.6.1.2. Attestation "To-Be-Signed" Definition

The `attestation-tbs-map` (attestation to-be-signed map) is an envelope data structure for the information that is to be serialized, hashed, and signed by the attestation machinery. The most important field in `attestation-tbs-map` is the `partial-claim-hash`: essentially the hash of the Asset and its referenced Assertions.

```
; Container data structure for the information "attested"

attestation-tbs-map =
{
    “partial-claim-hash”: bstr,     ;hash of partial claim
    “alg”: tstr                     ;hash algorithm used in part-claim-hash
    ? “pub-key”: bstr,              ;claim signer public key
    ? “created”: tdate,             ;UTC when this map was created
    ? “other-tbs-info”: tstr        ;Optional parameters
    ? “other-tbs-info-2”: tstr      ;Optional parameters
}
```

Field	Description
`partial-claim-hash`	The hash of a Partial Claim - i.e., a Claim that omits all or some of the Attestation Assertions. If the final Claim contains a single Attestation Assertion, then `partial-claim-hash` will be the hash of the CBOR-serialized Claim omitting the Attestation Assertion - i.e., usually the final entry in the `assertions` array. If the claim contains *n* Attestation Assertions, then *n* Partial Claim hashes are defined, with zero to *n-1* embedded Attestation Assertions.
`alg`	The hash algorithm used to compute `partial-claim-hash`.
`pub-key`	(Optional) The DER-encoded public key and algorithm of the Claim Signer (the Subject Public Key as used in the X.509 certificate) as specified in RFC-5480 and RFC-8017.
`created`	(Optional) The time and data when the `attestation-tbs-map` was created. Can be set to reduce the risk that the same claim signer can later add a different claim signature.
`other-tbs-info`	(Optional) Additional information that the Claim Creator wishes to associate with the attestation.

Field	Description
`other-tbs-info-2`	(Optional) Additional information that the Claim Creator wishes to associate with the attestation.

7.6.1.3. Attestation Assertion Definition

`attestation-info-map` is the envelope data structure that contains the attestation data/signature and related information. It is used to embed an Attestation Assertion in the Assertion Store.

`attestation-info-maps` are boxed with label `c2pa.attestation` (or `c2pa.attestation`, `c2pa.attestation_001`, `c2pa.attestation_002`, if more than one attestation is required.)

Attestation Assertions can be placed at any index in the `assertions` array but adding them as the final entries in the order that they are created is preferred.

The `attestation-info-map` definition presented here can encode many types of attestation. See Appendix A for currently defined attestation encodings.

```
; Encoding of an attestation into an assertion

attestation-info-map = {
  "att-type":tstr,                          ; type of attestation
  “attestation-tbs”: attestation-tbs-map,   ; The attestation-tbs-map
  "attestation-results": bstr,              ; attestation result/signature
  ? "certificates" : tstr,                  ; PEM-encoded certificate or certificate chain
  ? “created”: tdate                        ; attestation creation time
  ? “other-info”: bstr                      ; other info
  ? “other-info-2”: bstr                    ; other info
  ? “pad”: bstr                             ; padding
  ? “pad2”: bstr                            ; secondary padding
  ? "metadata": $assertion-metadata-map,    ; additional information about the assertion
}
```

Field	Description
`att-type`	The attestation type. E.g. `c2pa.SGX`, See Appendix A for currently defined types. If non-standard attestations are encoded, the same convention for non-standard Assertion labels should be used. E.g., `com.litware.custom-assertion`
`attestation-tbs`	The `attestation-tbs-map` used when the assertion was obtained.
`attestation-results`	Binary encoding of the attestation measurement.
`certificates`	(Optional) String version of the PEM encoded certificate of certificate chain for the attesting key. The root certificate should not be included.
`created`	(Optional) The date-time when the attestation was created.
`other-info`	(Optional) Any additional information that the Claim Creator wishes to associate with the attestation.

Field	Description
`other-info-2`	(Optional) Any additional information that the Claim Creator wishes to associate with the attestation.
`pad`	(Optional) Zero-filled byte string used for filling up space. See Section 9.7 for background on the necessity of padding in this structure.
`pad2`	(Optional) Second zero-filled byte string used for filling up space. See the note in Section 11.4.5 of the C2PA spec on the necessity of both `pad` and `pad2` fields.

7.7. Creating a Claim Containing One or More Explicit Attestations

These are the detailed steps for creating an embedding an Attestation into a Claim. In the 1.3 version of the C2PA specification, these steps should be performed immediately prior to *“11.3.2.4. Signing a Claim”*. Following the steps listed here, the Claim should be signed as described in the main specification.

1. Create a copy of the `claim-map` containing all non-attestation assertions.
2. Calculate the hash of the serialized (Partial) `claim-map` to form *hash(claim-map)*.
3. Create an `attestation-tbs-map` data structure, including *hash(claim-map)*, the time, and (optionally) the public key of the Claim Signer. Add any desired optional data to `attestation-tbs-map`. CBOR-serialize and hash to form *hash(attestation-tbs-map)*.
4. Use the appropriate platform service to obtain an attestation over *hash(attestation-tbs-map)* to form attestation *attest(hash(`attestation-tbs-map`))*
5. Create a new `attestation-info-map`, then:
   a. Set `att-type` to the type-selector for the attestation performed (Appendix A)
   b. Set `attestation-tbs` to `attestation-tbs-map`
   c. Set `attestation-results` to *attest(hash(attestation-tbs-map))* obtained in the previous step. Note that defined types and encodings are specified in Appendix A.
6. Create a JUMBF attestation container with label `c2pa.attestation` (or `c2pa.attestation_00n`, if more than one attestation must be included) containing the `attestation-info-map` being prepared.
7. Add the `attestation-info-map` to the Assertion Store.
8. Add a new `hashed-uri-map` to the `assertions` array in `claim-map`, referencing the newly added Attestation Assertion.
9. If more than one attestation is required, go back to (2), but start with the current Partial `claim-map` containing the attestations calculated thus far.

At this point, the final claim has been created, and processing can continue according to *“11.3.2.4. Signing a Claim”* using the `claim-map` with all embedded Attestation Assertions.

Note that this generic flow does not specify how an attestation is encoded into `attestation-results`. This is scheme-specific and is described in Appendix A.

7.8. Validating a Claim Containing Explicit Attestations

This section describes how Claims containing explicit attestations should be validated. This section also describes how non-attestation aware validators should behave: this text will be replicated in the main C2PA technical specification.

7.8.1. Attestation Aware Validator

NOTE This section is preliminary: it will be refined based on prototyping and feedback.

An attestation aware validator should perform all of the steps documented in the Validation section of the main C2PA technical specification. This will include checks that the attestation assertion(s) hash is correctly specified in the manifest.

If all currently defined checks defined in the main specification succeed, then an attestation-aware validator will perform the following additional steps. If any of the checks specified here fail then the Claim should be considered invalid.

1. To perform attestation validation, first all Attestation Assertions are removed from the `assertions` array in the Claim to form the (first) Partial Claim.[2]
2. If there is only one Attestation Assertion, then that Attestation Assertion becomes the Current Attestation Assertion. If the Claim contains more than one Attestation Assertion, then the first Attestation Assertion becomes the Current Attestation Assertion.
3. The Current Attestation Assertion is then checked using the following steps:
   a. The `att-type` field is checked to see if it understood and expected.
   b. The `attestation-tbs-map` is extracted from the `attestation-info-map` and the following checks are performed:
      i. If the `alg` field is present, then the Validator should check that the algorithm is supported. If `alg` is not present, then the hash algorithm defined by the enclosing claim should be used.
      ii. The `hash` field is checked to see if it matches the hash of the current Partial Claim.
      iii. If present, the `pub-key` field must match the public key associated with the Claim Signature.
   c. The `attestation-results` is validated. How this validation is performed will depend on the attestation technology used. The following steps are generic. See Appendix A for scheme-specific detail.
      i. `attestation-results` encodes the attestation measurement. The external data hash that was used when the attestation was created should match the hash of the `attestation-tbs-map` that is encoded in this assertion.
      ii. `attestation-results` is typically a signature that chains up to an Attestation Trust Root. If

`certificates` is present, Validators should check that the certificate chain is valid and chains up to a known Attestation Trust Root. If `certificates` is not present, then the Validator should check that the attestation measurement is signed by a known Attestation Trust Root.

iii. `attestation-results` also typically encodes measurements of the device, application, and/or security environment when the Attestation was created. The policy for evaluating these measurements is not in scope for this specification. General considerations for managing attestation measurements are discussed in Chapter 3.

4. If the preceding checks succeed then this Attestation Assertion has been validated.
5. If the Claim contains additional Attestation Assertions, the `attestation-info-map` that was just processed is added back to the `assertions` array in Partial Claim in the place that it was removed forming the next Partial Claim, and processing continues at Step 3 using the next Attestation Assertion as the Current Attestation Assertion.

When all Attestation Assertions have been processed successfully, the Claim is considered valid.

7.8.2. Non-Attestation Aware Validator

> (This text belongs in the main specification.)

This text will be added to the current main-specification validation section. It ensures that assets containing attestation assertions compliant with this specification are not rejected. Practically, this means that non-attestation aware validators will ignore attestation assertions, and will not reject claims containing them.

“For validators that do not consume attestations, any assertion with label starting with `c2pa.attestation` should be ignored. Third party unrecognized attestations, including third-party attestation assertions, are ignored as specified elsewhere.”

[1] This specification does not demand that attestation assertions appear last in the attestation array, because doing so would limit the future evolution of the C2PA standard.

[2] Note that when an Attestation Assertion is removed, the Claim is serialized using normal CBOR rules. For example, if a claim included two standard assertions and one Attestation Assertion, then the Partial Claim will contain an `assertions` array containing two elements. If a Partial Claim contains two standard assertions and two Attestation Assertions, then two Partial Claims are defined: one omitting just the last Attestation Assertion in the `attestations` array, and one omitting both Attestation Assertions. Of course, the order of the Attestation Assertions must be preserved as the Partial Claim is created and validated.

Chapter 8. Trust Roots for Attestation

Attestation introduces new Trust Roots. Devices that support attestation are typically certified by the device vendor. Sometimes these keys are used directly in attestations and sometimes these keys are used to certify additional attestation keys using additional trust roots.

If RATS Verifiers are employed, then C2PA Validators must also be aware of the trust roots used by the RATS server/Verifier.

In either case, Validators must manage additional trust roots for PKIs that support device attestations.

Attestations also (indirectly or directly) convey measurements of the Claim Creator and the hosting platform, so Validators must be aware of the measurements associated with trustworthy attesting applications and platforms.

Conceptually, sets of acceptable measurements are associated with specific Trust Roots, and it is anticipated that the Trust Roots and measurements of trusted application will be maintained and distributed similarly.

Chapter 9. Implementation Considerations

This chapter contains implementation notes and some tradeoffs to consider when choosing an attestation technology and embedding scheme. A thorough working knowledge of the relevant attestation technologies is assumed.

9.1. Attestation Technology

A variety of hardware-based attestation technologies are available. Most platforms offer just one choice, although some platforms have two. For example, many platforms contain a TPM that can attest the "rich OS", and some contain an additional attestation root for reporting the software running in a protected enclave.

System architects may consider the following factors when choosing an attestation technology:

1. What is available? The platform may only offer one attestation technology. For example, a TPM may be the only attestation technology available on a platform. In this case, the choice is made for the system architect.
2. Which System Services need to be "trusted?" Some aspects of Claim Creation can be performed without use of external services. For example, creating asset-bindings (hashes) can be implemented entirely by the Claim Creator without the use of OS services. However, some scenarios benefit from attesting that data was securely obtained from IO devices such as a camera sensor or GPS receiver. If attestation is to increase trust in data "passed in" from outside the Claim Creator application, then the device or OS that provides this data must be attested, be immutable, or be authenticated. This is generally simplest when the whole OS is attested.
3. Online or Offline? Some attestation technologies (PlayIntegrity, RATS) require an online connection to a server create an attestation. Others can create attestations offline.

9.2. Implicit or Explicit Attestation

Explicit Attestations are created at the time of Claim Creation and are bundled in the manifest. Explicit attestations supplement a conventional Claim Signature and are bound to it so that neither the attestation or the Claim Signature can be independently stripped or replaced.

Implicit Attestation is a Signature with a key that was provisioned for the Claim Creator using attestation primitives, but reflects some past security configuration for the platform or Claim Creator.

System architects may consider the following factors when choosing between Implicit and Explicit attestations:

9.2.1. Availability of Suitable Sealing Primitives

Implicit attestation requires that the Implicit Attestation Key (IA-Key) be protected (i.e., only accessible to the authorized Claim Creator) from the point of provisioning to the point of use.

Most platforms that support attestation also support sealing primitives that can be used to protect the IA-Key when the Claim Creator is not running. For example, the TPM supports sealing to PCR values that define the security configuration, and Intel SGX supports sealing to enclave state.

Implicit attestation works best when suitable sealing primitives are available. If no suitable sealing primitives are available, then short-lived IA-keys may be used. For example, each time the attesting Claim Creator is launched it could obtain a new certified Implicit Attestation key and never persist it to storage.

9.2.2. Online or Offline Claim Creation

When suitable sealing primitives are available, Implicit Attestation can generally be performed offline: it is just a simple signature using the IA-key.

Whether Explicit Attestation can be performed offline will depend on the attestation technology used. For example, a "raw" TPM quote can be obtained offline, while a PlayIntegrity attestation requires an online connection to a PlayIntegrity server. Similarly, a RATS attestation requires an online connection to a RATS server when the attestation is created.

System architects should consider whether online connections at the time of attestation creation are available and acceptable when selecting an attestation technology and embedding scheme.

9.3. RATS or Native Attestations

RATS (Remote Attestation Procedures) is a suite of technologies and protocols that allow a third-party verifier to translate a diverse set of "raw" attestations into simpler and easier-to-understand trust assessments. I.e., RATS will generally relieve the Claim Validator of the need to understand the details of the attestation technology and acceptable attestation measurements used by the Claim Creator.

This is generally a good thing, but it comes at the cost of:

1. The availability of a RATS server that is trusted by the Claim Verifier, and
2. An online connection to a RATS server at the time of Claim Creation.

Note that RATS also defines protocols for "key attestation." These techniques can be used to enroll keys for Implicit Attestation.

9.4. Claim-Creator Implicit Attestation or Embedded Implicit Attestation

This specification supports Claim-Signer implicit attestation, where the signature with the IA-key **is** the Claim Signature, and Embedded Implicit Attestation, where the IA-key signature is embedded in an attestation assertion and the resulting Claim is signed with a separate Claim Signature.

Embedded Implicit attestation is favoured when Claim Creators wish to create manifests that are associated with both a person/organization *and* the identity of the an application or device.

9.5. Space and Time Considerations for Attestation Creation

Implementation complexity and cost for Claim Signer Implicit Attestation is similar to normal Claim Creations: it just differs in the type of key used sign the Claim.

Explicit attestation requires additional steps, with implications for both Claim Creation time and memory usage. Explicit attestation requires that a Partial Claim be constructed, attested, and bundled/embedded as an assertion. (This may be repeated if more than one attestation is required.) Once the final Claim is available, it is signed as usual.

The Claim Creator must protect the Claim as it passes from a Partial Claim to a finalized Claim with a Claim Signature. Obviously, this is easiest if the Partial Claim can be held in Claim-Creator-private RAM. If this is not possible, then the Claim Creator may use cryptography or suitably protected external storage during processing.

9.6. Space and Time Considerations for Attestation Verification

Implicit Attestations are processed identically to normal Claims.

Embedded Explicit Attestations are processed by non-attestation aware Claim Validators in the normal way, but they will ignore the attestation assertion.

Embedded Explicit Attestations processed by attestation-aware Claim Validators require additional steps. The Claim Signature will be validated, as usual, but then the the Partial Claim must be created and the attestation must be evaluated. This requires additional data structure manipulation, an additional signature verification, and and additional checks that the attestation measurements are within policy.

9.7. Multiple-Step Processing With a Data Hash Assertion

When a C2PA Manifest uses a `c2pa.hash.data` assertion for the hard binding to the media, there is an interdependency between the hard binding assertion and the attestation assertion. This interdependency can be summarized by highlighting two key steps in the process:

1. The `c2pa.attestation` assertion contains a hash of the Partial Claim, which in turn contains a hash of the `c2pa.hash.data` assertion.
2. The `c2pa.hash.data` assertion contains an `exclusions` field in it, which implicitly encodes the size of the C2PA Manifest. But the size of the C2PA Manifest will change when the `c2pa.attestation` assertion is finalized.

The contents of the `c2pa.attestation` assertion depends on the contents of the `c2pa.hash.data` assertion, which depends on the size of the `c2pa.attestation` assertion. The solution to the problem is to use padding. For example, the `pad` and `pad2` fields of the `attestation-info-map` can be used to add padding to the `c2pa.attestation` assertion before finalizing the `exclusions` field of the `c2pa.hash.data`. A procedure for accomplishing this is described in Section 11.4 of the C2PA specification.

> **NOTE**
>
> This issue does not apply to other hard binding assertions that don’t contain the manifest size in the assertion. For example, `c2pa.hash.bmff.v2` and `c2pa.hash.boxes` are both immune to this problem, and so don’t require padding in the `c2pa.attestation` assertion.
>
> Also, for certain implementations it might be possible to predict the size of the `c2pa.attestation` assertion based on known characteristics of the signature and hashing algorithm(s) used. In this case, padding may not be required.

9.8. Considerations for C2PA Salt in the Partial Claim

In Section 8.3.1.3, the C2PA specification describes a method for adding salt to assertions before constructing the Claim. Since this salt is part of the JUMBF structure, and not part of the assertion data, some implementations might not store this salt in internal (deserialized) structures, and might just generate it as part of the serialization to JUMBF process.

In such implementations, it is possible that the salt value could change between the process of serializing the JUMBF structure for the Partial Claim, vs. serializing the JUMBF structure for the full Claim that will be signed by the C2PA Signature. If the salt changes between these two steps, then the Claim’s hashes for each assertion will also change, and the `c2pa.attestation` assertion will fail verification.

It is important that the assertion salt values (if any) do not change between generation of the Partial Claim and the full Claim.

Appendix A: Defined Attestation Schemes and Encodings

This appendix describes the currently defined attestation schemes and how attestation information is incorporated into a C2PA manifest.

In all cases, attestations should be calculated 'over' the hash of the serialization of the Partial Claim.

The documentation for some of the schemes presented here are incomplete. C2PA seeks feedback and proposals for how these (or other) schemes and technologies should be incorporated.

A.1. Android Key Attestation

Android Key Attestation provides a way to determine if an asymmetric key pair is hardware-backed, what the properties of the key are, and what constraints are applied to its usage. See the following Android references for details:

- Key and ID Attestation
- Verifying hardware-backed key pairs with Key Attestation

A.1.1. Other Android Attestation Methods

Key Attestation is one method of performing attestation on Android devices. Other methods include:

- **SafetyNet Attestation API** - This API has been deprecated and replaced by the Play Integrity API.
- **Play Integrity API** - Attestations received via this API are primarily intended to verify "internal" communications between collaborative application services, and are not designed to be ingested by a 3rd party. There are two problems associated with Play Integrity attestations that limit their applicability for C2PA attestation:
    - The *integrity verdict* provided by Play Integrity is wrapped in JWE (JSON Web Encryption). It can only be decrypted by an entity in possession of the application’s decryption key, which should not be shared publicly, and probably should not even be stored or accessed locally on the Android device.
    - Once the *integrity verdict* is decrypted, it contains a JWS (JSON Web Signature) which must be verified with the proper public key. Google provides this public key to developers without a backing certificate, and it is not clear how long-lived this key is and how relying parties should handle key rotations.

A.1.2. Key Attestation Overview

When adding an Android Key Attestation assertion to a C2PA claim, the Claim Generator will create a keypair in a StrongBox trusted execution environment. The private key will be used to sign over the `attestation-tbs-map`, and the public key is presented in a certificate that indicates it was generated in StrongBox, and has a chain-of-trust that traces back to a root signed by Google.

The end-entity certificate (the one containing the public key portion of the keypair) also has an x509v3 extension that contains data about the environment in which the keypair was created. A validator can optionally parse this extension data to gain increased confidence about the state of the device where the claim was generated.

A.1.3. Creating the Assertion

1. Generate the `attestation-tbs-map` as described in Section 7.6.1.2.
2. Hash this `attestation-tbs-map`. This hash will be used as the *challenge* value for the Key Attestation. The hashing algorithm used should be the same algorithm that was used when creating the `partial-claim-hash` parameter of the `attestation-tbs-map` (the algorithm reported in the `alg` parameter of the `attestation-tbs-map`).
3. Request a keypair from the Android keystore, passing the *challenge* value in a call to `setAttestationChallenge()` as described here. This will create a private key, secured in the keystore, and a certificate chain containing all certificates from the Google root to the end-entity certificate, which provides the public key for this keypair.
4. Sign the previously generated `attestation-tbs-map` with the private key in the keystore.
5. Place the signature, along with the certificate chain (minus the root certificate) into the `attestation-info-map`, as described in Section 7.6.1.3.
6. Store the `attestation-info-map` in the C2PA manifest’s Assertion Store as a new `c2pa.attestation` assertion.
7. Continue with the normal C2PA claim signing process.

A.1.4. Definition of Fields in `attestation-info-map` for Android Key Attestation

Field	Value
`att-type`	`c2pa.AndroidKeyAttestation`
`attestation-results`	The signature digest created by signing over the `attestation-tbs-map` with the private key.
`certificates`	All of the non-root certificates provided by the `KeyStore` when the keypair is created. This includes the end-entity certificate (with the *attestation extension* in it) but excludes the root certificate.
`other-info`	The signature algorithm used to generate the `attestation-results` must be placed in this parameter as a null-terminated ASCII byte string. The list of valid algorithms is provided in the Signature Algorithms section of the C2PA specification.

A.1.5. Validating the Assertion

1. Obtain Google’s Hardware Attestation Root certificates. Since various devices will provide a trust chain against

any one of these roots, the validator should attempt to verify the assertion's certificate chain against each of them.

2. Check each certificate in the chain for revocation using the certificate revocation status list (CRL) maintained by Google.
3. The assertion is validated according to the procedure described in Section 7.8. The `attestation-results` signature should be verified by checking the signature digest with the public key in the end-entity certificate from the `certificates` field. The signing algorithm is provided in the `other-info` field, as described above.
4. If this validation succeeds, then extra steps may optionally be performed to provide increased confidence in the integrity of the signing device.
    a. From the end-entity certificate, find and extract the x509v3 extension with OID of `1.3.6.1.4.1.11129.2.1.17`. This is the *attestation extension*. It is an ASN.1 structure adhering to this schema.

    b. In the *attestation extension*, check the `attestationSecurityLevel` and the `keymasterSecurityLevel`. Both should be set to a value of `2` (`StrongBox`).

    c. In the *attestation extension*, check the `attestationChallenge` value. It should match a hash of the `attestation-tbs` parameter from the `attestation-info-map`.

    d. In the `teeEnforced` structure of the *attestation extension*, in the `rootOfTrust` structure, check the `verifiedBootState` parameter. It should be set to a value of `0` (`Verified`).

A.2. Intel SGX

Intel SGX Attestation provides evidence that the application is running on an Intel SGX enabled platform, inside of a properly instantiated enclave, on a system with a known security configuration.

The high-level flow of an Intel SGX ECDSA attestation is:

1. The Intel SGX workload contacts the relying party and requests access to a service or resource.
2. The relying party responds by issuing a challenge: it asks the Intel SGX workload to identify itself and provide proof that its credentials are valid.
3. To satisfy the challenge, the Intel SGX workload generates a quote, which is a cryptographic measurement of the instantiated enclave. The quote is signed using the attesttion collateral that's stored in the data center caching service.
4. The quote is sent it to the relying party over the secure communications channel.
5. The relying party verifies the quote. It fetches the attestation collateral associated with the quote from the data center caching service and uses it to verify the signature.
6. Assuming the quote is valid, it examines the quote metadata and the trusted compute base (TCB) level that is associated with the signing key. The service then applies its security policy and decides whether it should trust the enclave.

Intel does not define a specific protocol for communications between the Intel SGX workload and the relying party. How the two systems establish their secure communications channel and exchange information is up to the solution provider.

A.2.1. Obtaining an Attestation

A.2.1.1. Generate the Quote

Generate a quote and write it to a file named quote.dat

```
QuoteGenerationSample$ ./app
sgx_qe_set_enclave_load_policy is valid in in-proc mode only and it is optional: the default
enclave load policy is persistent:
set the enclave load policy as persistent:succeed!

Step1: Call sgx_qe_get_target_info:succeed!
Step2: Call create_app_report:succeed!
Step3: Call sgx_qe_get_quote_size:succeed!
Step4: Call sgx_qe_get_quote:succeed!cert_key_type = 0x5
sgx_qe_cleanup_by_policy is valid in in-proc mode only.

 Clean up the enclave load policy:succeed!
QuoteGenerationSample$ ls -l quote.dat
-rw-r--r-- 1 sgxtest users 4594 Jan 28 14:31 quote.dat
```

A.2.1.2. Transfer the quote to the relying party

Copy this file to the system that acts as a relying party. One can use `scp` tool, but any secure file transfer option will yield similar result.

```
$ scp quote.dat relyingparty:.
sgxtest@relyingparty's password:
quote.dat                                 100% 4594     100.8KB/s   00:00
```

A.2.1.3. Verify the quote

Run the code, using the -quote option to provide the path to your quote.dat file.

```
QuoteVerificationSample$ ./app -quote ~/quote.dat
Info: ECDSA quote path: /home/sgxtest/quote.dat

Trusted quote verification:
    Info: get target info successfully returned.
    Info: sgx_qv_set_enclave_load_policy successfully returned.
    Info: sgx_qv_get_quote_supplemental_data_size successfully returned.
    Info: App: sgx_qv_verify_quote successfully returned.
    Info: Ecall: Verify QvE report and identity successfully returned.
    Info: App: Verification completed successfully.

===========================================

Untrusted quote verification:
    Info: sgx_qv_get_quote_supplemental_data_size successfully returned.
    Info: App: sgx_qv_verify_quote successfully returned.
    Info: App: Verification completed successfully.
```

The sample application performs two verifications. The first, labeled “Trusted quote verification”, is done in an Intel SGX enclave using Intel’s reference quote verification library and quote verification enclave. This approach requires that Intel SGX be enabled on the platform. The second attestation, labeled “Untrusted quote verification”, also uses Intel’s reference quote verification library, but it’s done in untrusted memory and does not require Intel SGX.

A.2.1.4. Inspect the quote

With the quote verified, inspect the quote and examine its contents. Remember that quote verification merely tells whether the quote is valid, and whether there are issues with the remote enclave’s TCB. Trusting the quote is not the same as trusting the enclave: to make a trust decision about the enclave, its metadata must be examined and, at minimum, answer the following questions:

- Is this enclave one that can be recognized? Specifically, can verification recognize MRSIGNER, MRENCLAVE, and the vendor-specific identifiers?
- Is the debug attribute set?
- Is the quoting enclave one that can be identified? And is it up to date?
- Is the enclave’s software version up to date? Note that this check assumes the software that generates the SGX image maintains a proper SVN process.

A production service may have even more stringent requirements. Ultimately, these are all policy matters; it is up to the service provider to decide which enclaves their service will trust and how that service should arrive at its decisions.

Here it is not intended to do exhaustive examination of `quote`, but some key fields have been examined that can answer a subset of the questions above. The quote structure is defined in the header file sgx_quote_3.h, and the xxd Linux tool can be used to extract required fields.

Here’s the MRENCLAVE value:

```
~$ xxd -s 112 -g 0 -l 16 quote.dat
00000070: da77d1e3e10c61e405442f117ae10f0e .w....a..D/.z...
```

And the MRSIGNER:

```
~$ xxd -s 176 -g 0 -l 16 quote.dat
000000b0: d412a4f07ef83892a5915fb2ab584be3 ....~.8..._..XK.
```

These two values indicate (a) the enclave identity (which enclave it is) and (b) the enclave signer (i.e the entity that signed the enclave). As a rule, a relying party should never trust an enclave or a signer that it doesn’t recognize!

Additional information on the quote structure can be obtained at Overview of Intel® Software Guard Extensions Instructions and Data Structures and Github: SGX data Center Attestation.

A.2.1.5. Definition of Fields in `attestation-info-map` for Intel SGX.

Field	Value
`att-type`	`c2pa.SGX`
`attestation-results`	UTF-8-encoding of the null-terminated attestation result
`certificates`	
`other-info`	(Optional)

A.3. TPM 2.0

A.3.1. Obtaining an Attestation

Applications should create C2PA attestations using the `TPM2_Quote` operation with the `qualifyingData` set to the digest of the Partial Claim using the same hash algorithm as Platform Configuration Registers (PCRs) that are used.

The key used for signing should be a **restricted signing key**, and applications should specify a set PCRs that adequately represent the security configuration of the host platform.

Depending on scenario, it may be necessary or desireable to also include/embed a certificate chain for the quoting key.

A.3.2. Definition of Fields in `attestation-info-map` for TPM 2.0

Field	Value
`att-type`	`c2pa.TPM2.0`
`attestation-results`	Raw binary `TPMT_SIGNATURE` obtained from the TPM without padding.
`certificates`	base-64 encoded PEM-encoded certificate chain for the quoting key.
`other-info`	Raw binary `TPM2B_ATTEST` data obtained from the TPM without padding.

A.4. IETF RATS

IETF RATS (Remote Attestation Procedures) defines a suite of protocols that enable attesters (devices/applications) to perform an attestation exchange with a RATS Verifier (a Trust Broker) and receive an attestation result/verdict in a RATS compliant format. In this revision of the document, the RATS Passport based topology is followed and the received attestation verdict (known as the Passport) is encoded as a C2PA attestation assertion, which is then embedded in a Manifest.

The remainder of the sub-sections describe how the desired attestation can be obtained and interpreted from a device complying with the ARM Platform Security Architecture (PSA) which supports the IETF RATS based attestation protocol. Additional architectures will be defined in the future.

The Attestation Evidence is a set of signed platform-specific claims that is generated from the ARM PSA device. The Attestation Evidence is a specific profile of an Entity Attestation Token (EAT) documented here. The RATS trust broker which performs the Attestation Verification and provides the Attestation Verdict is built on Open Source componentry called Veraison

A.4.1. Obtaining an Attestation

The ARM PSA Attestation Architecture API is documented here.

In order to comply with C2PA requirement of binding user-supplied data (the hash of the to-be-signed Partial Claim) to the platform attestation AND ease of integration of the ARM PSA attestation with the C2PA claim signing application, a simple binding layer enhancement to the base PSA Implementation on the device is used. This enhancement is documented here.

With this enhancement in place, the following sequence should be used to create the attestation:

1. The C2PA Claim-Signer Application invokes the binding layer API `uint8_t request_attestation(uint8_t* data_ptr, uint8_t data_size, uint8_t* result_data_ptr, uint8_t* result_size)`. Here `data_ptr` points to the hash of `attestation-tbs-map` and `data_size` is the size of the hash, in bytes. The remaining arguments are return values that are set if the function call completes successfully.
2. Upon receipt of the call, an attestation protocol handshake is performed between the binding layer and the Veraison RATS Trust Broker Service. The exact message exchange follows the RATS Challenge-Response Protocol, which is described here. The high level flow of RATS Verification steps performed by a RATS Trust Broker Service can be found here.
3. If the attestation protocol completes successfully, the function returns `0` and the `result_data_ptr` receives the attestation verdict encoded as a signed Jason Web Token (JWT) or CBOR Web Token (CWT) object.
4. In the case of JWT, `attestation-results` in `attestation-info-map` is set to the UTF-8-encoded JWT, as described in Section 7.6.1.3. In the case of CWT, `attestation-results` in `attestation-info-map` is set to the CBOR-encoded token without further encoding. The media type of the `attestation-results` payload is used to deduce the encoding. For example, a JWT serialization would use: `application/eat-jwt; eat_profile="tag:github.com,2023:veraison/ear"` and CWT would use `application/eat-cwt; eat_profile="tag:github.com,2023:veraison/ear"`

A.4.2. Verifying the Attestation Assertion

A RATS Attestation Result is based on the EAT Attestation Result format serialized as a JWT or CWT. The semantic core is defined by the AR4SI trustworthiness vector i.e. a collection of well-defined appraisal categories that uses a standardized "scoring" system to present a normalized, yet rich view of the appraised attester. The basic purpose of the trustworthiness vector is to decouple a broad range of attesters/devices from the Relying Parties so that the latter can take a fine-grained policy decisions without having any specific knowledge about the device they are interacting with.

An attestation-aware C2PA Validator shall perform the following steps for attestation verification.

1. Extract the Attestation Assertion
2. Extract the `att-result` field to obtain the CWT or JWT, then verify the signature using the public key of the Verifier. The Verifier public key can be either pre-fetched from the Verifier or obtained at the time of Verification using the defined public interface, here
3. Upon signature verification, decode the CWT or JWT, for example, using the EAT Attestation Result(EAR) Library here.
4. From the decoded content, extract the `user data` from the EAR Veraison specific extension
5. Compute the hash of the Partial Claim and match it with `user data` claim from the EAR
6. Decode the `att-result` to get the structure of the Attestation Results as given in EAT Attestation Result or using equivalent "C" based decoder here
7. Decode the Appraisal structure to obtain the `Trustworthiness Vector`
8. Ascertain that the Status value is `TrustTierAffirming`. A Verifier which receives Status value as `TrustTierContraindicated` can conclude that the underlying platform is not trustworthy
9. Verify that the Hardware and Executables trustworthiness claim are set to `TrustTierAffirming`. This is an indicator of a genuine hardware running a trusted executable platform

A.4.3. Definition of Fields in `attestation-info-map` for IETF RATS

Field	Value
`att-type`	`c2pa.RATS`
`att-result`	The EAT Attestation Result (EAR) message, either a UTF-8 encoded JWT or the binary CBOR Web Token.

A.5. Implicit Attestations Encoded in Attestation Assertions

A.5.1. Obtaining an Attestation

The Implicit Attestation signature is a signature over the hash of the Partial Claim as described in section Section 7.6.1.1.

A.5.2. Definition of Fields in `attestation-info-map` for Embedded-Implicit Attestation

Field	Value
`att-type`	`c2pa.embedded-implicit`
`attestation-results`	Signature over Partial Claim using one of the signature encoding allowed by RFC 5280 (https://www.rfc-editor.org/rfc/rfc5280).
`certificates`	(Optional) base-64 encoded PEM-encoded certificate chain for the Implicit Attestation key
`other-info`	TBD