UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
E scola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

Specification and implementation of metadata for secure image provenance information

Master Thesis
submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by

Nikolaos Fotos

In partial fulfillment
of the requirements for the master in
Cybersecurity

Advisor: Jaime M. Delgado
Barcelona, 7th of July 2022

Contents

List of Figures

Listings

List of Tables

Revision history and approval record

Revision	Date	Purpose
0	27/05/2022	Document creation
1	14/06/2022	Document revision
2	20/06/2022	Document revision
3	27/06/2022	Final Document revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Nikolaos Fotos	nikolaos.fotos@estudiantat.upc.edu
Jaime M. Delgado	jaime.delgado@upc.edu

Written by:		Reviewed and approved by:	
Date	28 /06/2022	Date	01/07/2022
Name	Nikolaos Fotos	Name	Jaime M. Delgado
Position	Project Author	Position	Project Supervisor

Abstract

The booming of AI tools capable of modifying images has equipped fake media producers with strong tools in their arsenal. Complementary to the efforts of implementing fake media detectors, research organizations are designing a standardized way of describing the modification history of digital media in a cryptographically secure way, ensuring that this information cannot be tampered with. This thesis proposes a specification which focuses on JPEG images and specifies a data model based on the JPEG Universal Metadata Box Format (JUMBF) standard. Furthermore, it proposes the encryption of a subset of provenance metadata that could pose privacy-related risks to the users. Along with the specification, a library has been developed to manage provenance information of JPEG images. To that extent, a set of libraries that handle JUMBF information is required to be implemented. These libraries have been submitted as a proposed reference software contributing to the JUMBF standard.

1 Introduction

With the digital transformation of information sharing, the world has become connected more than ever. With the proliferation of smart devices and the improvement of Internet connections media sharing and consumption has become a relatively cheap task. On top of that, the abundant use of social media has changed the way people share experiences, educate and keep up with the latest news. All these actions pose various challenges to the users and solutions need to be proposed.

First and foremost, recent advances in media creation and modification allow the production of near realistic (e.g. deepfakes) media assets that are almost indistinguishable from original assets to the human eye. Fake media along with the emerging phenomenon of deepfakes can proliferate the problems of disinformation and misinformation. If used accurately, it could guide public opinion towards a specific side or lead to social unrest. To address this problem, [1] discusses a set of state-of-the-art tools that try to detect such media.

At the same time, social media users have the tendency to disclose their personal moments with their social media followers. They capture a picture of a scene using their mobile phone and upload it to social media platforms without taking into consideration the privacy risks that this might cause. Metadata about GPS coordinates or information about the capturing device could be leveraged by adversarial users. Users depend solely on the devices and social media platforms to protect (e.g. strip) these pieces of metadata from their media before sharing; otherwise they are at risk of disclosing information that they didn't intend to.

The aforementioned challenges could be mitigated in a proactive manner. Specifically, it would make sense to annotate media creation and modifications in a clear and transparent way. This could be considered a crucial element in many usage scenarios bringing trust to the users. To ensure trust, these annotations should be related to the media in a cryptographically secure way to prevent them from being compromised. With this solution it is possible to keep track of the provenance history of a media asset.

Provenance refers to the basic, trustworthy facts about the origins of a piece of digital content (image, video, audio recording, document). It may include information such as who created it and how, when, and where it was created or edited. The content author always has control over whether provenance data is included as well as what data is included. Included information can be removed in later edits.

1.1 Background

Regarding the privacy concerns for media sharing, [2] has worked on developing mechanisms for protecting regions of interest of a JPEG image as well its metadata. In addition, [3] has provided a mechanism to protect JPEG images using privacy policies that could be either embedded in the image or completely managed by an external service.

In parallel, the annotation challenge has already triggered various organizations to develop a wide range of mechanisms that can detect and/or annotate modified media assets when

they are shared. These annotations should be attached to the media in a secure way to deter them from being altered. Apart from JPEG [4], Coalition for Content Provenance & Authenticity [5] have launched initiatives to design standards focusing towards this directive.

1.2 Problem

Although there is a set of standards and specifications focusing on securely annotating digital assets, there hasn't been yet the effort to specify those use cases where end-to-end encryption and selective access of provenance metadata is required. These use cases describe situations where specific metadata shall be protected as they might disclose Personal Identifiable Information (PII), geo-location or device information.

In fact, C2PA Technical Specification [6] introduces the ability of a user to remove a set of provenance information that could pose privacy concerns. As per C2PA, this action is defined as *redaction*. However, there is no provision about protecting sensitive metadata. Our proposed specification takes into consideration these use cases and extends the C2PA capabilities by allowing the users to control their privacy-related metadata.

1.3 Solution

Based on all the above, this project focuses on designing a technical specification that defines the model to express provenance information for JPEG images while allowing the possibility to protect a set of metadata that could pose privacy risks for the user producing it. This proposed technical specification is based on the work introduced by C2PA.

In scope of this project, a library has been developed showcasing the capabilities of the specification. Additional libraries need to be implemented that focus on expressing and handling information using the JPEG Universal Multimedia Box Format since the provenance data model is based upon it. These libraries are submitted as proposed reference software to the JUMBF standard. Eventually, the aim of this project is to submit the specification to the JPEG Call For Proposals [7] that requests for contributors who shall support the creation of a new standard addressing the Fake Media problem.

1.4 Objective

The list of objectives, that this project aims to accomplish, is listed below:

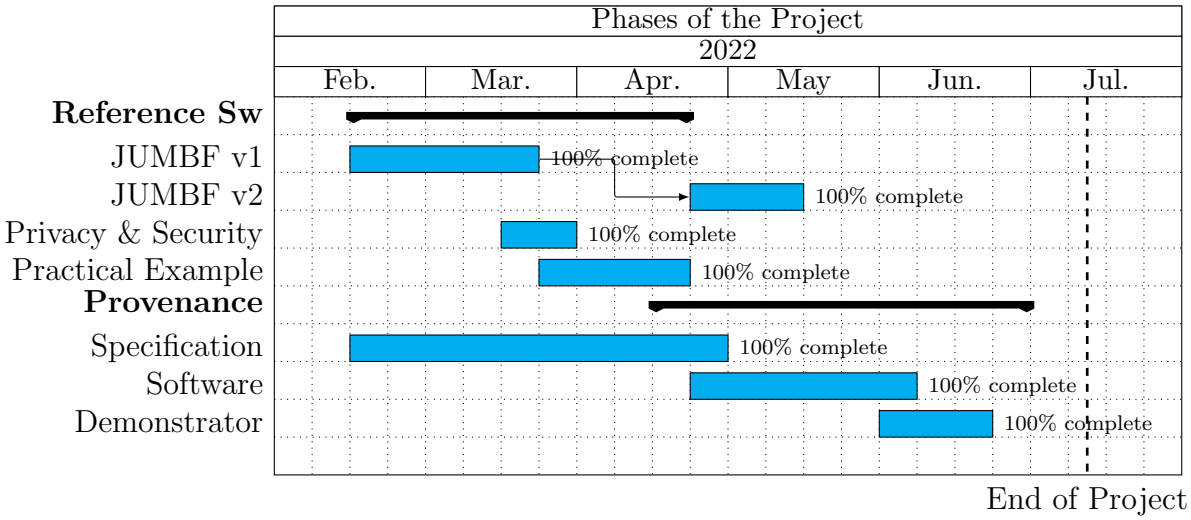
1. Design a specification that proposes a data model to express provenance information for JPEG images. This data model is based on the JPEG Universal Media Box Format (JUMBF). In addition, this specification describes the functionalities to produce and verify the integrity of the provenance structure.
2. In contrast with the existing work in the research, the proposed specification allows for protection - through encryption - of a set of assertions that could disclose personal information.

3. Implement a library with the functionality proposed in the specification
4. Implement a set of libraries to model and handle metadata in JPEG Universal Box Format (JUMBF). These libraries shall be used by the software in step 3 and they can be also used for any other project interested in handling JUMBF metadata.
5. Contribute to a set of JPEG standards. Specifically, the JUMBF-related libraries that are implemented in this project are proposed as a reference software to the JPEG standards related to JUMBF. In addition, the proposed provenance specification along with its library shall be submitted as a contribution to the JPEG Call for Proposals regarding a new standard targeting Fake Media.

1.5 Planning

The planning of the work of this project is depicted in the following Gantt diagram. Initially, effort was put to develop the libraries that model and handle JUMBF metadata inside a JPEG image. In parallel, the investigation and composition of the proposed specification for Fake Media was taking place.

Once the aforementioned tasks had finished on late April, it was time for the implementation of the library that models and operates on the proposed provenance data model and, subsequently, the effort was shifted towards the demonstrator application that showcases the functionalities that were developed throughout the project.



1.6 Organization of the document

The remaining parts of this document is structured as follows: In Chapter 2, the state-of-the-art standards and specifications related to embedding provenance - as well as other metadata - information in JPEG images are presented. Next, Chapter 3 focuses on the implementation of a proposed reference software for the JPEG Universal Metadata Box Format standard. On top of that, an application of this software is presented handling

the structures defined in JPEG Privacy & Security standard. Chapter 4 describes the proposed specification to express provenance information of a JPEG image while Chapter 5 presents the library that implements it. Chapter 6 showcases two demonstration applications related to the libraries implemented in the previous sections. Subsequently, the integration of this specification with the MIPAMS Environment is presented in Chapter 7. Chapter 8 discusses the future steps of this work. Finally, in Chapter 9 a short conclusion is presented focusing on the contributions that this work has provided in the JPEG standardization procedures.

2 State-of-the-art tools and standards

This work is based on JPEG Systems [8], a multi-part specification focusing on consolidating image formats, functionalities, and code stream syntax into one uniform system.This standard is published by ISO/IEC and is being developed by ISO/IEC JTC 1/SC 29/WG 1 (JPEG Coding of digital representations of images) [4], a joint working group of the International Standardization Organization (ISO) and the International Electrotechnical Commission (IEC). In addition, work produced from Coalition for Content Provenance and Authenticity [5] organization is studied as well. These specifications provide the mechanisms to embed metadata to an image, protect the metadata content and build provenance information that describes the modification history of an image. Finally, this thesis project takes into consideration the recent explorations of the JPEG Committee regarding the design of a new standard, namely JPEG Fake Media [9], that can facilitate a secure and reliable annotation of media modifications.

2.1 JPEG Universal Metadata Box Format (JUMBF)

One of the main focuses of JPEG Systems specification is the standardization of a universal way to embed metadata in an image. In fact, JPEG Systems Part 5 specifies the JPEG Universal Metadata Box Format (JUMBF) which provides the means to express, embed and request different types of metadata in a JPEG image. Related to this part, one International Standard has been published [10] and an updated version [11] supporting additional structures is currently under Draft International Standard (DIS) ballot status.

A JPEG image could contain metadata that is textual, image content or binary. According to JUMBF standard, to embed metadata in a JPEG image it is required that it is wrapped inside a container called box. Each of these boxes is wrapped again with a set of header values following the ISO Base Media File Format (BMFF) specification. These header values describe the length and type of the specific box. Before embedding a box to the JPEG image, these boxes are wrapped with APP 11 Marker Segments in order to follow the encoding of the entire JPEG code stream. There are different types of boxes depending on the type of metadata that needs to be expressed. The most general definition of a box in this standard is a JUMBF Box the structure of which varies according to the content of that Box.

Field	Explanation
TYPE	16-byte UUID
TOGGLES	An integer that signals which of the following fields are present
LABEL	A String that can be used to reference the JUMBF box
ID	An assigned 4-byte Id
SHA256HASH	A SHA-256 Digest
PRIVATE	An ISOBMFF box structure instide the Description box

Table 1: Description box field definition

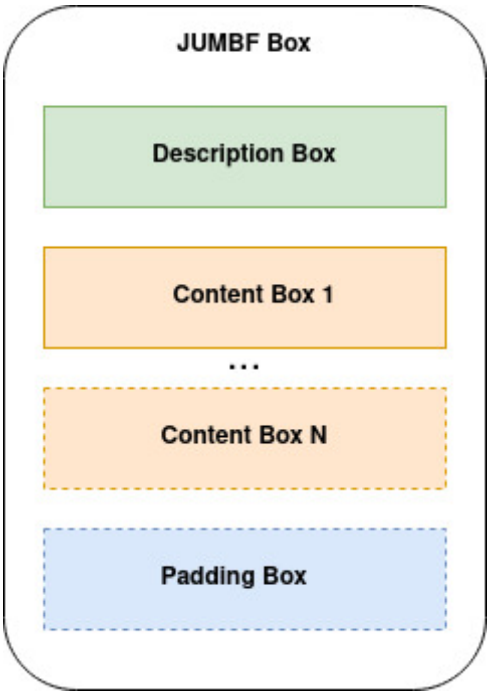


Figure 1: JUMBF Box structure

As depicted in figure 1, the structure of a JUMBF Box consists of one Description box, one ore more Content Boxes and at most one Padding box. A JUMBF Box can have another JUMBF Box as a content Box allowing for a hierarchical structure. In that case the parent JUMBF Box is also called a superbox.

A JUMBF Description box provides additional information about the behaviour and content of the JUMBF Box in which it is contained. The fields are listed in table 1.

In order to express different types of metadata, [11] has specified a set of content boxes for the JUMBF Box structure:

- XML box
- JSON box

Content Type	UUID
XML Content type JUMBF box	0x786D6C20-0011-0010-8000-00AA00389B71
JSON Content type JUMBF box	0x6A736F6E-0011-0010-8000-00AA00389B71
Codestream Content type JUMBF box	0x6579D6FB-DBA2-446B-B2AC-1B82FEEB89D1
UUID Content type JUMBF box	0x75756964-0011-0010-8000-00AA00389B71
Embedded File Content type JUMBF box	0x40CB0C32-BB8A-489D-A70B-2AD6F47F4369
CBOR Content type JUMBF box	0x63626F72-0011-0010-8000-00AA00389B71

Table 2: Supported Content Types for JUMBF Boxes

- Contiguous Codestream box (i.e. for Image codestream)
- UUID box (i.e. for vendor specific data format)
- Embedded File Description box (i.e. description box for arbitrary binary data)
- Binary Data box (i.e. for arbitrary binary data)
- CBOR box

The Content Type of a JUMBF Box describes the type and number of content boxes. Specifically, the Content Type is specified in a field inside the Description box and is expressed as a 16-byte UUID. [11] has specified 6 Content Types for a JUMBF Box but it is possible for other standards to define their own Content Type UUID and structure. The 6 basic Content Types are listed in table 2:

Except from Embedded File Content type JUMBF box, the remaining Content Types describe JUMBF Box structures that contain exactly one Content Box. For instance, a XML Content type JUMBF box contains one Description box, one XML box and at most one Padding box.

Last but not least, [11] specifies the mechanism that can be used to reference or request the content of JUMBF Boxes. In fact, a URI format has been specified to mention the contents of a specific JUMBF Box using the label field of its Description box. An example of such URI reference is shown below:

```
https://jpeg.org/image.jpg#jumbf=parentlabel/childlabel
```

Listing 1: JUMBF URI reference

There are additional standards that use JUMB Format in their applications. In particular, JPEG Systems Part 4: Privacy & Security standard [12], specifies a set of new JUMBF structures called Protection and Replacement Content type JUMBF boxes. With these two boxes a set of mechanisms is specified supporting various privacy and security features. For example, content protection could be achieved using a Protection Content type JUMBF box. Specifically, part of an image could be encrypted and placed inside a Protection Content type JUMBF box before embedded to the image. Access rules are also supported and can be embedded in a separate JUMBF Box. That way, only authorized users shall have access to the protected content.

There are more JPEG projects working around JUMBF standard (e.g. JLINK, JPEG 360, JPEG Snack). However, there is not yet a reference software showcasing JUMBF functionality or any of its applications. Working towards that direction, JPEG has recently released a new JPEG Systems part (i.e. Part 10) focusing on the implementation of such software.

2.2 Coalition for Content Provenance and Authenticity (C2PA)

The C2PA [5] is a Joint Development Foundation project to collectively build an end-to-end open technical standard to provide publishers, creators, and consumers with opt-in, flexible ways to understand the authenticity and provenance of different types of media. C2PA merges two separate initiatives, namely Adobe-led Content Authenticity Initiative (CAI) [13] and Microsoft-led Project Origin [14].

C2PA has recently produced a new technical specification [6] that describes the technical aspects of the C2PA architecture. Specifically, it defines a model for storing and accessing provenance information about various formats of digital media (i.e. digital asset). This provenance information consists of statements made by an actor which are called Assertions. Assertions are wrapped with additional information forming a claim. Then this claim is digitally signed in order to ensure tamper-evident provenance information. The aforementioned pieces of information form a C2PA Manifest. A C2PA Manifest constitutes a point in the provenance history of a digital asset. All these points related to a specific digital asset can be gathered into a C2PA Manifest Store.

Since a C2PA Manifest Store can be embedded inside the referenced digital asset, it is defined as a JUMBF superbox. In fact, all the internal structures of a C2PA Manifest Store (e.g. C2PA Manifest, Assertions, Claim, Claim Signature) constitute a specific JUMBF Box with its own Content Type. Furthermore, [6] defines the mechanisms to generate and consume provenance information. These mechanisms ensure that provenance metadata has not been tampered with. In addition, they ensure accountability in the sense that any assertion made about a digital asset is eventually related to a specific actor.

2.3 JPEG Fake Media Requirements

Working to this direction, JPEG has recently launched a new initiative for discovering use cases and requirements for addressing Fake Media [15]. The goal is to to create a new standard that can facilitate a secure and reliable annotation of media asset creation and modifications. This standard aims to cover use cases that refer to scenarios of malicious intent (e.g. Misinformation and disinformation) as well as those of good faith (e.g. Media creation and modification).

Based on the use cases identified, a set of requirements for the new standard has been released and organized in the following categories:

- Media creation and modification descriptions.
- Metadata embedding and referencing.
- Authenticity, integrity, and trust model.

Recently, a call for proposals document has been issued from JPEG requesting contributors to provide tools and information regarding the proposed use cases and requirements. For a contribution to be considered valid, it must cover a set of the requirements defined in [15].

The work proposed in this project is planned to be submitted as a contribution to this call for proposals as it covers a set of the defined requirements. In table 3, the set of requirements covered by the proposed specification in section 4 is listed. The enumeration of the requirements in table 3 is in line with the enumeration presented in [15].

In section 4.4 a more detailed discussion is provided about how these requirements are met by the proposed specification.

Ref. No.	Requirement
1.1	The standard shall provide means to describe how, by whom, where and/or when the media asset was created and/or modified.
1.8	The standard shall provide means to keep track of the provenance of media assets and/or of specific modifications.
2.3	The standard shall consider privacy of individuals and locations.
2.5	The standard shall provide means to explicitly denote anonymous, obscured, or redacted information. If the information is not provided, then it is considered anonymized.
2.7	The standard shall be viable as a self-contained structure.
2.8	The standard shall provide means to verify the integrity of the media asset by various hash (2.8.1) and signing (2.8.2) methods.
2.10	The standard shall be compliant with JPEG Privacy and Security to provide means to secure media asset metadata, including provenance information.
3.3	The standard shall provide means to verify the authenticity of media assets.
3.5	The standard shall provide meanest to verify the integrity of media assets.

Table 3: JPEG Fake Media Requirements covered by the proposed specification

3 JUMBF Reference Software

The main goal of this section is to present the development of a project that showcases the ability to model, generate and parse information expressed according to the JPEG Universal Multimedia Box Format standard [11]. The libraries presented in this section are part of the mipams-jumbf project. The entire codebase of the project is located at the following github repository URL: `https://github.com/nickft/mipams-jumbf`.

The modular architecture of the proposed software allows for easier and wider support of JUMBF Boxes defined in other standards. Currently, there are two libraries: the first one is called jumbf-core and contains all the definitions related to the JUMBF standard (i.e. ISO/IEC 19566-5) while the second one - which is called jumbf-privsec - contains the JUMBF Boxes defined in scope of Privacy and Security standard (i.e. ISO/IEC 19566-4).

The libraries presented below are written in Java and specifically using Spring Boot framework [16]. The main concept is to separate the data model definition with the classes responsible for handling (i.e. parsing/generating) JUMBF Boxes. Regarding the latter group of classes, they are implemented as Java Beans facilitating the dependency injection between these services. Furthermore, depending on the application scenario, the developer has the ability to explicitly define - in a fine-grain manner - the services (i.e. Java Beans) that are important for her specific use case. This makes the library both lightweight and robust.

The development of these two libraries have leaded to the submission of two input documents to the JPEG Systems specification. Specifically, the first input document [17] presents the jumbf-core library while the second document [18] showcases how jumbf-privsec could extend the functionality of the first library in order to support the JUMBF Boxes defined in Privacy & Security standard. These two input documents has contributed to the decision of JPEG Systems committee to issue a new part for the JPEG Systems specification, focusing on the implementation of a reference software. The aim of the reference software is to - initially - showcase the applicability of the JUMBF standard and eventually cover the JPEG Systems specification in its entirety (i.e. all the standards under JPEG Systems).

Finally, it is worth mentioning that in order to verify the correctness of our software a set of integration tests were executed against a parsing tool implemented by Content Authenticity Initiative (CAI) [13]. Basically, this codestream parser by CAI (the github repository of the tool is available here) receives a JPEG image as input and generates a report with all its contents including the parsed JUMBF metadata. Consequently, the tests consist of generating a set of images with JUMBF metadata using our software and verified that they are parsed successfully by the CAI tool.

3.1 JUMBF Core Library

The jumbf-core library provides the means to generate and parse information that is stored in JUMB Format. Furthermore, it defines an interface so that developers can extend its

functionality and support additional JUMBF Box structures as specified in other JPEG standards (e.g. Protection Box definition in ISO/IEC 19566-4).

In the first version of the library (i.e. jumbf-core-1.0), the JUMBF structure is stored to a separate file using the ISO Base Media File Format (ISOBMFF). Thus, a new file extension ".jumbf" was introduced, containing a list (i.e. concatenation) of ISOBMFF boxes. Each box corresponds to one JUMBF Box. This allows us to store the metadata information separately from the media asset itself. However, in the revised version of the library (i.e. jumbf-core-2.0), the embedding of JUMBF structure into the referenced JPEG image codestream is supported as well. This is achieved by encapsulating JUMBF metadata with APP11 segments as described in ISO/IEC 19566-5 standard. Version 2.0 of the jumbf-core library is in line with the second version of ISO/IEC 19566-5 which is currently in Draft International Standard (DIS) under ballot status.

3.1.1 Box structure

In this section the classes that have been designed in order to support the data model presented in ISO/IEC 19566-5 standard are specified. The core concept in this data model is a *Box*.

To describe a Box structure in jumbf-core library two classes need to be specified: an Entity class and a Service class. An Entity class contains the information regarding the fields that are defined in the specific Box. In addition, all the functionality which is required to parse/generate a particular box is included in its respective Service class. This allows for a better separation of concerns in our software.

Furthermore, it is important that the different Content Types of a JUMBF Box are defined. The number as well as the type of the Content Boxes inside a JUMBF Box are specified from the Content Type UUID which is located in its Description box fields. Consequently, an additional Content Type class hierarchy is designed. Each Content Type class provides the means to parse and generate the necessary Content Boxes inside a JUMBF Box.

As mentioned previously, a Box structure is wrapped with the ISOBMFF header. Specifically, before writing the contents of a Box structure to a file, a set of bytes need to be reserved in order to write the ISOBMFF header which consist of the following values

- The Length of the box (LBox): 4 Bytes
- The Type of the box (TBox): 4 Bytes
- The box length extension (XLBox): 8 Bytes

In figure 2, the entire Entity class hierarchy is depicted, as defined in jumbf-core-2.0.

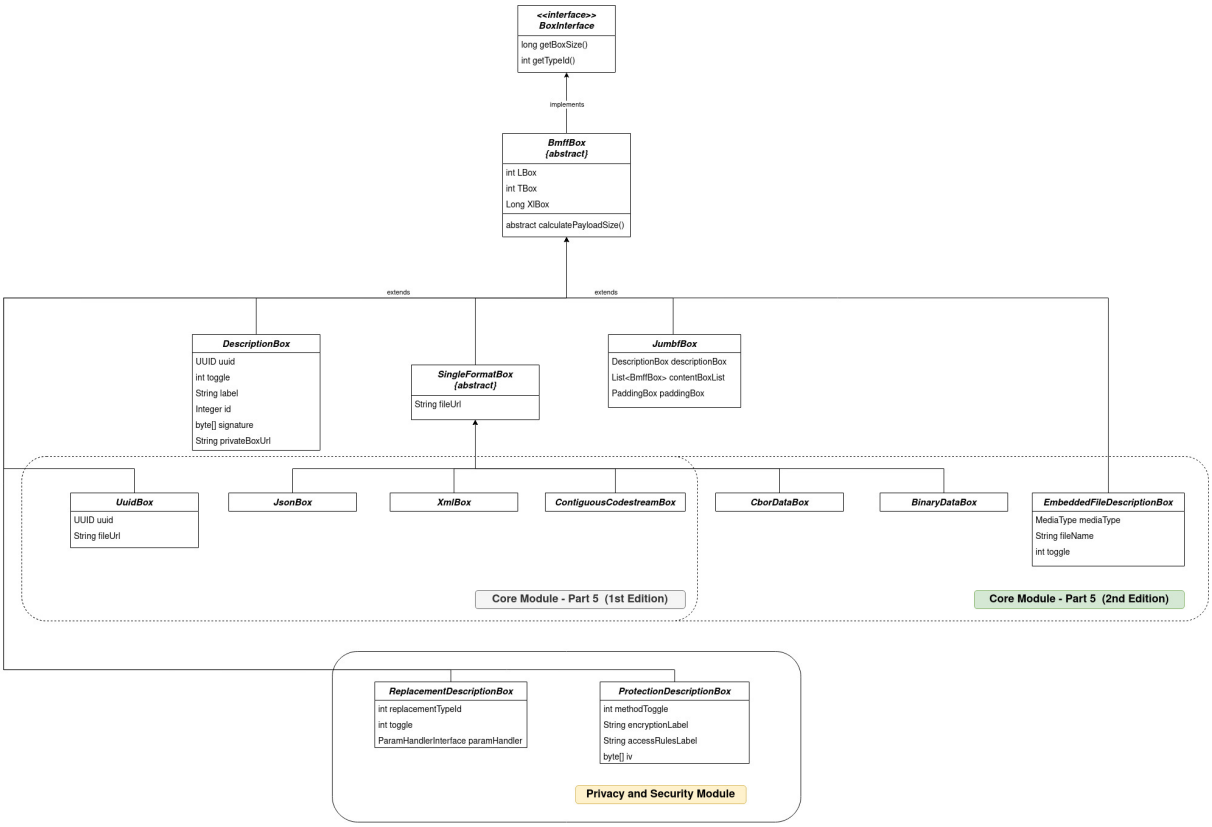


Figure 2: Entity class hierarchy in mipams-jumbf project

The core of the Entity class hierarchy is the BoxInterface interface which defines two methods, namely “Get the Box Type Id” and “Get the size of the Box”. In the next level there is a BmffBox abstract class that defines the fields which correspond to the ISOBMFF header fields. Subsequently, the Description box definition (e.g. DescriptionBox class) is presented along with the fields that are updated in scope of the second edition of ISO/IEC 19566-5. With these definitions, the JumbfBox class can be implemented as a set of fields consisting of exactly one DescriptionBox field, a list of BmffBox fields corresponding to the Content Boxes according to the Content Type specified in DescriptionBox’s uuid field and one PaddingBox field. Finally, in Figure 1, all the Content Box definitions are listed along with their respective fields. The core module level consists of the box structures defined in 1st and 2nd edition of ISO/IEC 19566-5 while the Privacy and Security module contains the box structures defined in scope of ISO/IEC 19566-4 standard (the latter boxes shall be examined in a subsequent section).

It is worth mentioning that the validity of the content of the content boxes is out of scope of the mipams-jumbf-2.0. It is up to the application that uses this library to evaluate the content of a Content Box. Based on that, an abstract class is defined, namely SingleFormatBox, that contains a single “fileUrl” field which contains the URL to the file containing the contents that should be included in a Content Box. As depicted in Figure 2, this abstract class is extended by the following Content Box classes: JSON, XML, Contiguous Codestream (JP2C), CBOR and Binary Data Content boxes.

Apart from the Entity classes defined above, the respective Service class for each Box need to be specified. Each Service should implement the methods to parse and generate only the fields that are defined in its corresponding Box. In figure 3 the entire Service class hierarchy is depicted.

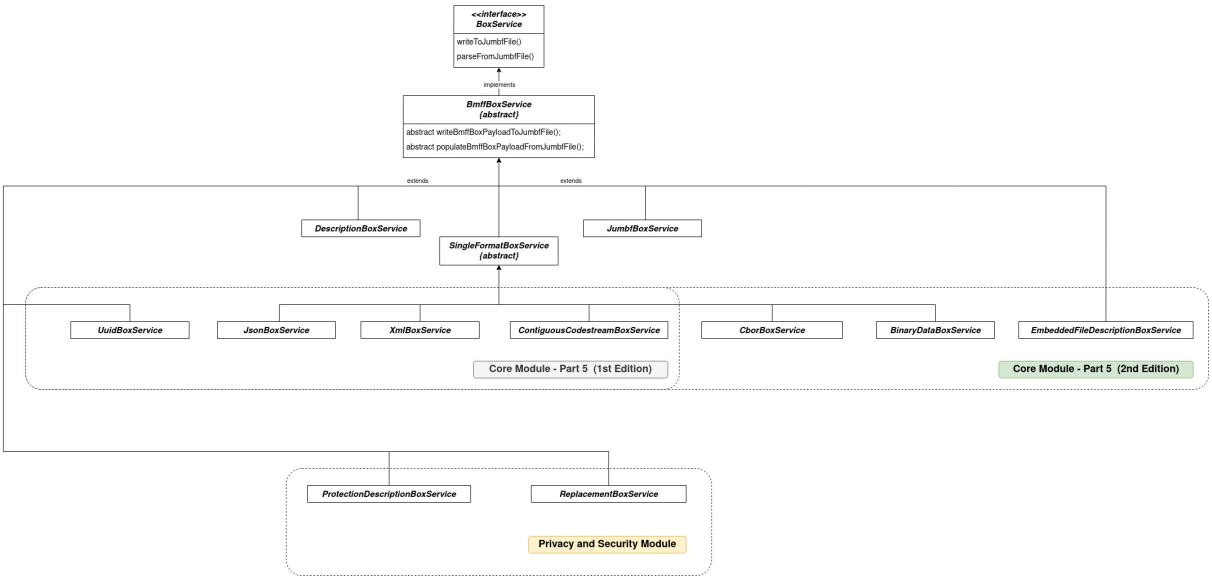


Figure 3: Service class hierarchy in mipams-jumbf project

The structure seems similar to the Entity class hierarchy. Specifically, at the top level an Interface exists, called BoxService, which specifies two methods: one for writing a Box Entity to a file and one for instantiating a Box Entity by parsing a JUMBF structure from a file. Next, the BmffBoxService abstract class contains the functionality to write an ISOBMFF structure to a file. It is evident that at this level the BmffBoxService class knows only how to handle the ISOBMFF headers of an Entity class. The way to handle the ISOBMFF payload shall be specified by the corresponding BoxService class that extends this BmffBoxService class. This is the reason BmffBoxService class marks the methods "Write ISOBMFF Box Payload" and "Populate BMFF Box Payload" as abstract classes. The remaining levels of the Core Module consist of the Services which specify how to handle each Box defined in the Entity class hierarchy.

3.1.2 Content Type class hierarchy

As mentioned in section 3.1.1, a JumbfBoxService class provides the means to handle the fields of a JUMBF Box. Notice that the DescriptionBoxService is always invoked when implementing a JumbfBoxService. However, the set of BoxService classes that are needed for the Content Box(es) depends on the Content Type of the JUMBF Box which is located in the Description Box Entity "uuid" field. For instance, for a JSON Content type JUMBF box the JumbfBoxService class will invoke - apart from the DescriptionBoxService - the JsonBoxService class. However, this is not the case for an Embedded File Content type JUMBF box which consists of two Content Boxes, namely an Embedded File Descrip-

tion Box and a Binary Data Box. In this example, the JumbfBoxService class will call the EmbeddedFileDescriptionBoxService class as well as the BinaryDataBoxService class. Consequently, in addition to the presented classes, it is required to define a family of Content Type classes that provide the means to handle the set of Content Boxes that are required according to the Content Type UUID. The Content Type hierarchy is depicted in figure 4.

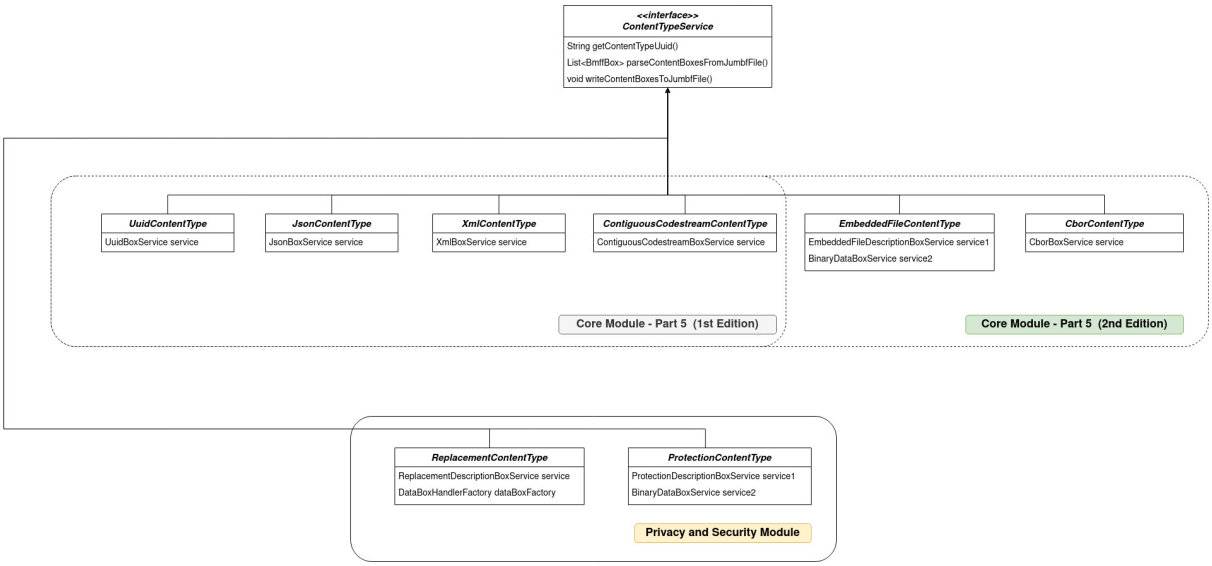


Figure 4: Content Type class hierarchy in jumbf-core-2.0

At the top level of figure 4, there is the core ContentTypeService interface which defines three important functionalities that each Content Type class shall implement: “Get the Content Type UUID”, “Parse the JUMBF Content Boxes from the JUMBF file” and “write the JUMBF Content Boxes to a JUMBF file”.

At the second level, all the Content Types supported in jumbf-core-2.0 are displayed. In each of the ContentType services the respective BoxService classe(s) required to handle the necessary Content Boxes are specified. It is worth mentioning that in scope of ISO/IEC 19566-5 standard the Embedded File Content type JUMBF box is the only Box definition that requires more than one Content Box.

3.1.3 Embedding JUMBF metadata to a JPEG image

In scope of jumbf-core-1.0 the generated JUMBF boxes were able to be stored in a separate file with .jumbf extension. However, since this is not yet standardized, it has been decided that from jumbf-core-2.0 JUMBF metadata need to be embedded in a digital asset as well. Thus, two additional services have been defined in the second version of jumbf core, namely JpegCodestreamGenerator and JpegCodestreamParser. With these two services it is possible to parse and embed JUMBF metadata inside a JPEG XT and JPEG 1 encoded images. For the scope of this project it is necessary to handle only JUMBF

metadata inside the image; the implemented parser and generation services skip do not handle the rest of the information inside the image codestream.

3.1.4 Example: Generating a JUMBF file

This section illustrates an example where an Embedded File Content type JUMBF box is generated and stored into a separate ".jumbf" file. Specifically, focus will be given on the invocations made by the reference software (i.e. jumbf-core-2.0 library) during the generation of a JUMBF file. Initially, it is assumed that there is already an instance of a JumbfBox Entity class that describes the Embedded File Content type JUMBF box. The output byte stream is logically grouped in figure 5.

Before the method invocations are listed, it is worth noting that in figure 5 it is evident which are the bytes that correspond to the ISOBMFF header and payload. Moreover, each orange box corresponds to a set of 4 bytes while the blue boxes are of variable length. The label of each box contains the Box Type that the grouped bytes are referring to as well as the exact content of each box. For example, the first box corresponds to the ISOBMFF LBox header of the generated JUMBF box (i.e. the JUMBF box type is 'jumb' or 0x6A756D62).

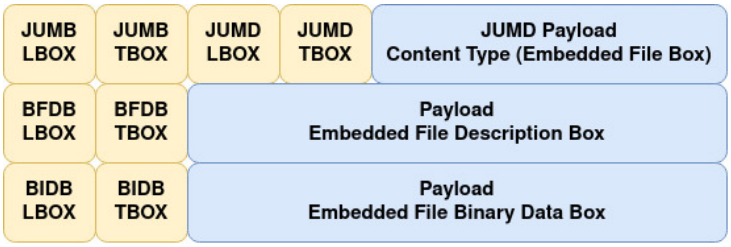


Figure 5: Embedded File Content type JUMBF box stored in a JUMBF file

The algorithm to generate a JUMBF file begins from the JUMBFBoxService class:

1. First, **JumbfBoxService** writes the ISOBMFF header bytes (JUMB LBox and TBox) through the functionality extended by the BmffBoxService class.
2. **JumbfBoxService** calls the DescriptionBoxService to proceed with its box byte generation.
3. **DescriptionBoxService** writes the ISOBMFF header bytes (JUMD LBox and TBox) through the functionality extended by the BmffBoxService class.
4. **DescriptionBoxService** writes its fields (e.g. Content Type UUID, Toggle and other optional fields). The operation returns back to the JumbfBoxService.
5. **JumbfBoxService** reads the Content Type UUID from the Description Box and chooses the corresponding BoxService to proceed with the JUMBF Content Box byte generation. In this example the correct service is the EmbeddedFileContentType.
6. **EmbeddedFileContentType** calls the EmbeddedFileDescriptionBoxService to

proceed with its box byte generation.

7. **EmbeddedFileDescriptionBoxService** writes the ISOBMFF header bytes (i.e. BFDB LBox and TBox) through the functionality extended by the BmffBoxService class.
8. **EmbeddedFileDescriptionBoxService** writes its fields, namely Content Type UUID, Toggle along with other optional fields. The operation returns back to the EmbeddedFileContentType.
9. **EmbeddedFileContentType** calls the BinaryDataBoxService to proceed with its box byte generation.
10. **BinaryDataBoxService** writes the ISOBMFF header bytes (BFDB LBox and TBox) through the functionality extended by the BmffBoxService class.
11. **BinaryDataBoxService** writes the bytes specified in the Binary Data Box structure. The operation returns back to EmbeddedFileContentType.
12. **EmbeddedFileContentType** finishes its operation which returns back to JumbfBoxService.
13. **JumbfBoxService** finishes its operation and the final .jumbf file has been generated successfully.

3.2 JUMBF Privacy & Security (privsec) Library

The primary goal of this section is to describe the JUMBF Box structures that are defined in the ISO/IEC 19566-4 standard. In addition, this section consists a proof of work that the structure defined in section 3.1 is extensible allowing Box definitions from other standards to be specified on top of the JUMBF Core library. The following sections focus on the implementation of the two Box structures defined in ISO/IEC 19566-4, namely the Protection and the Replacement boxes.

The extension of the jumbf-core library is implemented in a separate library called jumbf-privsec-1.0. This happens because the goal of jumbf-core library is to provide the foundations for other standards to implement their own JUMBF Box structures in a separate jumbf library. This allows the user (i.e. developer) to select only those jumbf libraries that define the Box structures that are necessary for her application. This is crucial both for performance reasons (i.e. less services instantiated) as well as for providing a lightweight library containing only the necessary dependencies.

3.2.1 Protection box

As per ISO/IEC 19566-4, a Protection box is defined as a JUMBF box with a new Content Type UUID consisting of two Content Boxes, namely a Protection Description box and a Binary Data box. Consequently, jumbf-core library needs to be extended in order to support the new Protection Description box Entity and Service classes and define the new Protection ContentType class.

Regarding the Protection Description box, first, it is required that its Entity class is defined by specifying the necessary fields. Since each Box structure is wrapped with the ISOBMFF headers the newly defined ProtectionDescriptionBox class needs to extend the BmffBox class as illustrated in the "Privacy & Security module" in figure 2.

Subsequently, the Service class needs to be defined. It provides the methods to parser and generates the fields that reside in a Protection Description box. Specifically, as shown in figure 3, the ProtectionDescriptionService class extends the functionality of BmffBoxService class.

Finally, all the necessary boxes are specified in order to define the new Protection Content Type. As explained in section 3.1, a ContentType class provides the means to parse/generate the content boxes of a JUMBF box. Depending on the Content Type UUID that resides in the Description box, the JumbfBoxService class decides the proper ContentType class to handle the JUMBF Box Content Boxes. The definition of the ProtectionContentType class is depicted in the "Privacy & Security module" in figure 4. As in the case of EmbeddedFileContentType class, the ProtectionContentType class contains two services corresponding to two Content Boxes: one for handling the Protection Description box and one for the Binary Data box.

3.2.2 Replacement box

This section focuses on the Replacement Content type JUMBF box definition supporting all the different types of replacement defined in ISO/IEC 19566-4. In general, the Content Box set of a Replacement Content type JUMBF box contains one Replacement Description Box and one or more Replacement Data Boxes, depending on the type of replacement. Specifically, there are four different types of replacement:

- *Box Replacement*: Replacing a box referenced by either an offset or a label with a list of one or more boxes specified in the Replacement Data Box section.
- *APP Replacement*: Replacing an APP marker segment that is referenced using the offset in the file with the contents (i.e. one or more app segments) of exactly one Replacement Data box which is a Binary Data Box.
- *ROI Replacement*: Replacing a region of interest (ROI) - as specified by the corresponding offset - in the parent image with the content of exactly one Replacement Data box which is a Contiguous Codestream box.
- *File Replacement*: Replacing the entire file where this Replacement box resides with the content of exactly one Replacement Data box which is a Contiguous Codestream box.

First of all, a new box is specified, namely the Replacement Description Box. Hence, a new class ReplacementDescriptionBox is created extending the BmffBox class as shown in the "Privacy & Security module" in figure 2. Each replacement type defines a different set of parameters. However, since the resulting Replacement Content type is the same for all replacement types (i.e. single Content Type UUID), a single ReplacementDescriptionBox class is specified and different parameter handler classes are defined, handling the param-

eters of each replacement type. Specifically, as shown at the bottom of figure 2, a field of type ParamHandlerInterface is included in the Replacement Description Box Entity class. These ParamHandlerInterface classes - apart from specifying the exact parameters of each replacement type - provide the functionality to parse and generate the specific parameter fields to bytes in order to be later included in the Replacement Description Box JUMBF structure. The class inheritance of the parameter handling classes is depicted in figure 6.

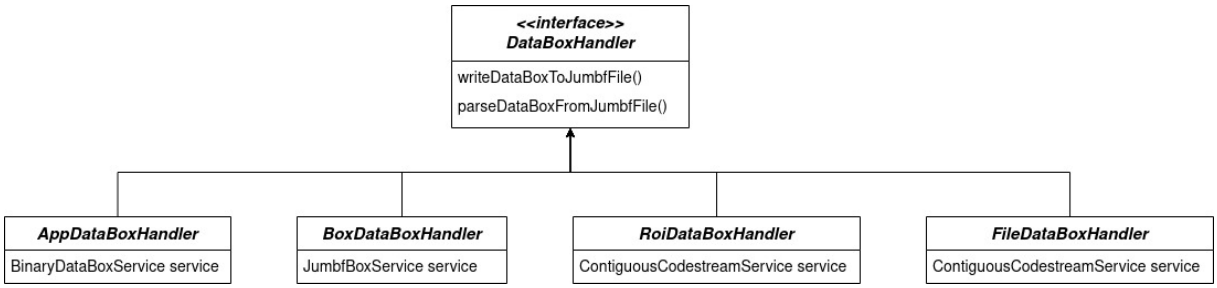


Figure 6: Replacement Description box parameter handling class inheritance

Subsequently, it is required to specify the methods that parse and generate a Replacement Description box to a JUMBF structure, namely the ReplacementDescriptionBoxService class (figure 3). Now that all the necessary boxes are implemented, the Replacement Content type can be specified as depicted in figure 4. It is worth mentioning that there are two services residing in the ReplacementContentType class. The first one is responsible for handling the Replacement Description box while the other one, namely DataBoxHandlerFactory class, provides the means to write the corresponding Replacement Data Boxes according to the replacement type.

Specifically, the DataBoxHandler interface, as described in figure 7, provides the means to parse/generate the Replacement Data Boxes of each replacement type. For the selection of the correct DataBoxHandler, a new DataBoxHandlerFactory class is defined that acts based on the replacementTypeId field provided by the Replacement Description Box.

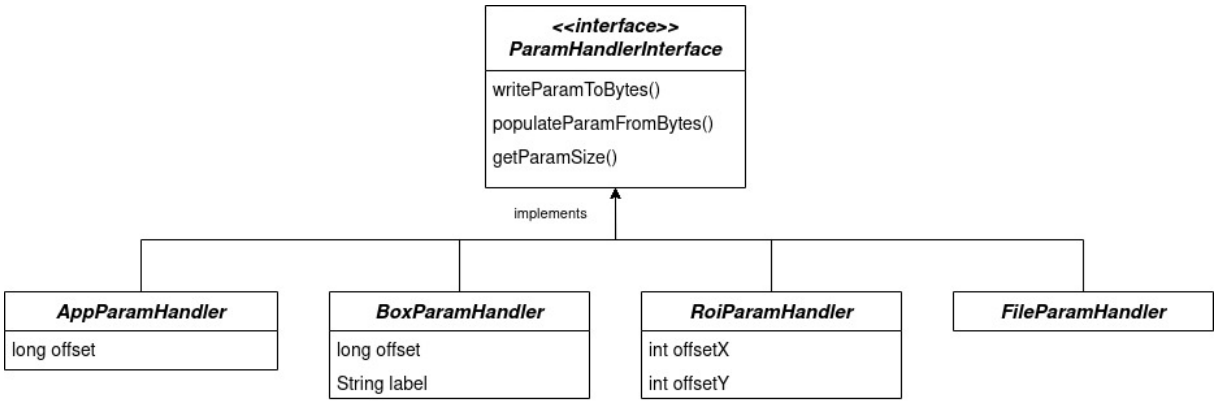


Figure 7: Replacement Data box handling class inheritance

Finally, in each DataBoxHandler class there is a service field assigned while the supported Replacement Data Boxes consist of boxes that are already defined in scope of the ISO/IEC 19566-5. Thus, the services to parse/generate the Replacement Data Boxes have already been defined in scope of the jumbf-core library.

4 Specification of JPEG Fake Media

This section presents the formalisation of the specification proposed in this project in scope of answering to the JPEG Fake Media call for proposals [7]. This specification is merely based on the technical specification introduced by C2PA [6]. By focusing the annotation of provenance metadata solely on JPEG images, our specification allows for simplified data model and operations. But more importantly, the added value that this specification suggests on top of C2PA architecture, is the provision of the use cases that require the protection of a set of metadata that might threaten the privacy of an actor. This can be achieved by including the Protection Content Type JUMBF Box as a supported Content Type to express an assertion (i.e. in C2PA, assertions are represented as one of the following Content Types: JSON, CBOR, Embedded File).

In scope of this specification, new JUMBF Box Content Types shall be defined in order to express provenance information to a digital asset. These new Content Types are defined in a library called mipams-fake-media-1.0 and depend on Box structures defined in both jumbf-core-2.0 and jumbf-privsec-1.0 libraries. The set of Content Type JUMBF Boxes are presented in table 4. In the following sections each of these structures is explained thoroughly.

Table 4: Provenance JUMBF Box definitions

Box Name	**Label**	**JUMBF Type**	**Explanation**
Manifest Store	mpms. provenance	0x6D707374-C65D-11EC-9D64-0242AC120002	The Manifest Store box shall contain at least one Manifest box.
Standard Manifest	[Manifest UUID]	0x6D70736D-C65D-11EC-9D64-0242AC120002	The *Standard Manifest* box shall contain one Claim, one Claim Signature and one Assertion Store and optionally a Credentials Store. As for the Assertion Store, a Standard Manifest box shall contain at least one content binding assertion.

Update Manifest	[Manifest UUID]	0x6D70756D-C65D-11EC-9D64-0242AC120002	The *Update Manifest* box shall contain one Claim, one Claim Signature and one Assertion Store and optionally a Credentials Store. As for the Assertion Store, an Update Manifest box shall contain exactly one ingredient assertion that: a includes a reference to that existing C2PA Manifest that is being updated and b has the value "parentOf" for the relationship field.
Claim	mpms. prov. claim	0x6D70636C-0011-0010-8000-00AA00389B71	The *Claim* box contains only one CBOR Content type box corresponding to the claim structure as described in section .
Claim Signature	mpms. prov. signature	0x6D706373-0011-0010-8000-00AA00389B71	The *Claim Signature* box contains only one CBOR Content type box to the claim signature structure as described in section .
Assertion Store	mpms. prov. assertions	0x6D706173-C65D-11EC-9D64-	The *Assertion Store* box contains a list of JUMBF super boxes corresponding to the assertion boxes as specified in section .
Credentials Store	mpms. prov. credentials	0x6D707663-0011-0010-8000-00AA00389B71	The *Credentials Store* box contains only one JSON Content Type box corresponding to the W3C credential of a user (section).

4.1 Data model

4.1.1 Assertion

An assertion is any provenance statement that an actor wants to securely include in the metadata of the JPEG image. An assertion is represented by a JUMBF Box, whose Content Type can be:

- CBOR [19],
- JSON,
- Embedded File for the case of a thumbnail assertion, or
- Protection in case an actor decides to encrypt the content of an assertion

The supported assertions needs to cover most of the provenance metadata that an actor might want to embed. These assertions range from direct modifications to a digital asset, to location metadata and digital rights. In table 5, a list of the supported assertion types is presented, originating from different standards like: C2PA, XMP, EXIF.

Table 5: Summary of supported assertions

Assertion Type	**Assertion Label**	**Origin**
Content Binding	mpms.prov.binding	C2PA
Actions	mpms.prov.actions	C2PA, XMP
Thumbnail	mpms.prov.thumbnail.auto *or* mpms.prov.thumbnail.manual	C2PA
Ingredient	mpms.prov.ingredient	C2PA
EXIF	stds.exif	EXIF

Subsequently, a graphical design of how an Assertion JUMBF Box should look like is depicted in figure 8.

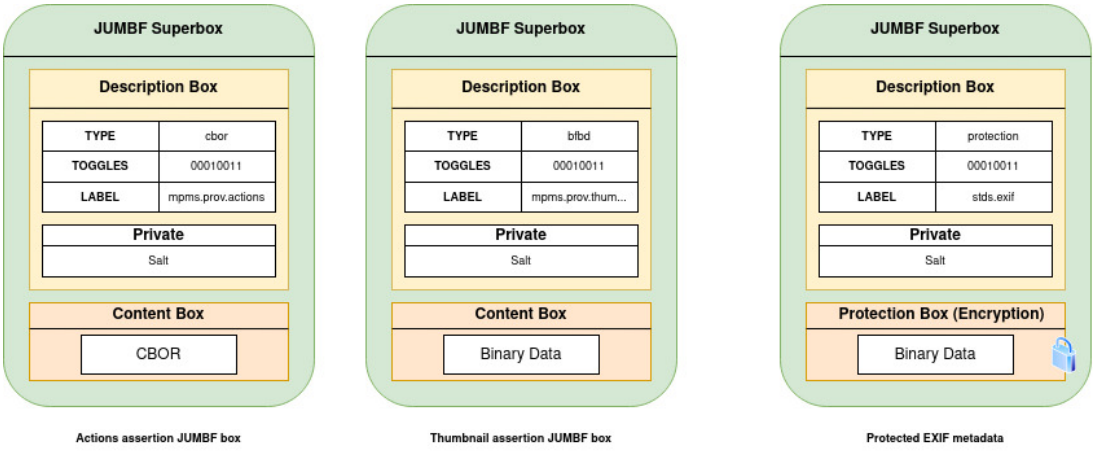


Figure 8: Assertion JUMBF Boxes

As observed in the above figure, TOGGLE is set to 00010011. According to ISO/IEC 19466-5, it means that a Private Field is present (3rd Most Significant Bit), the JUMBF Box can be requested and the label field exists in the Description box. Specifically, the label field is a string corresponding to the "Assertion Label" column presented in table 5. This label shall help the application understand the type of assertion that the Content Box contains.

Additionally, to add entropy to the hashing process a Description box's Private field is included that acts like a salt (i.e. a random 16 byte number). As defined in the subsequent subsections it is important that the hash of the Assertion JUMBF Boxes is calculated in order to verify their integrity. Thus, by using the Private field, it is likely that two assertion instances describing an identical modification in a digital asset won't have the same hash digest.

Regarding the Protection Box, it can be applied in the use cases where the producer wants to protect (i.e. encrypt) a subset of the provenance metadata before sharing a digital asset. Depending on the use case, access rules can be also applied accompanying the Protection Box.

In the following subsections all the types of assertions are described in detail.

4.1.1.1 Content Binding. This assertion describes the binding between a digital asset and its manifests. Specifically, this assertion should contain:

- A description of the hash algorithm (e.g. "sha256")
- A padding string (if needed)
- The hash of the digital asset
- A description string (e.g. "Hashing the content of the image by stripping its provenance metadata").

4.1.1.2 Actions. An *actions* assertion shall contain a list (e.g. a CBOR list) of the different actions that directly affect the digital asset. Each element of the list shall have a label specifying the type of the action. In table 6, all the supported action types are listed.

Table 6: Supported Action Types

Action Type	Label	Origin	Explanation
Convert	mpms.prov.converted	XMP	
Copy	mpms.prov.copied	XMP	
Create	mpms.prov.created	XMP	
Crop	mpms.prov.cropped	XMP	
Edit	mpms.prov.edited	XMP	
Filter	mpms.prov.filtered	XMP	
Format	mpms.prov.formatted	XMP	

Version update	mpms.prov.version_updated	XMP	
Print	mpms.prov.printed	XMP	
Publish	mpms.prov.published	XMP	
Manage	mpms.prov.managed	XMP	
Produce	mpms.prov.produced	XMP	
Resize	mpms.prov.resized	XMP	
Save	mpms.prov.saved	XMP	
Place Ingredient	mpms.prov.placed_ingredient	C2PA	Added/Placed an ingredient into the asset.
Transcode	mpms.prov.transcoded	C2PA	A direct conversion of one encoding to another, including resolution scaling, bitrate adjustment and encoding format change.
Repackage	mpms.prov.repackaged	C2PA	A conversion of one packaging or container format to another.
Unknown	mpms.prov.unknown	C2PA	Something happened, but the claim_generator cannot specify what.

Note: If any of the mp.placed, mp.transcoded, mp.repackaged, mp.unknown actions is used, it is compulsory that there exists a reference to a mp.ingredient assertion specifying which is the parent digital asset. The contents of an action assertion shall be serialized with CBOR format and are described below:

- Action: The type of the action as described in "Action Type" column in table
- Date: the time when the assertion was made. It might be different from the date where the claim is made.
- Metadata: A string providing additional information related to the action.
- Software Agent: A string descriptor related to the application that performed this type of action

4.1.1.3 Thumbnail. A thumbnail assertion provides an approximate visual representation of the asset at a specific event in the life cycle of an asset. This assertion is expressed as an Embedded File Content type JUMBF box. The information that need to be specified for this assertion is the filename of the thumbnail, the media type and the file itself which shall be embedded in the Provenance Metadata. The label of this assertion - which corresponds to the label of the Description box of the Embedded File Content type JUMBF box - is "mpms.thumbnail".

4.1.1.4 Ingredient. In many scenarios, an actor does not create a new asset from scratch; instead, she includes other existing assets to create her work - either as a derived asset, a composed asset or an asset rendition (i.e. rendering the digital asset in a different format or quality). These existing assets are called ingredients and they could carry their own provenance history as well.

An ingredient assertion should contain:

- A title for the ingredient. For instance it could be the name of the ingredient asset.
- The format of the ingredient (Media Type)
- A uniqueID that unambiguously identifies the ingredient digital asset.
- The relationship that the ingredient has with the examined digital asset. The supported relationship types are: "parentOf" and "componentOf".
- (Optional) A URI reference to the manifest of the ingredient digital asset
- (Optional) A thumbnail URI Reference to the thumbnail of the ingredient
- (Optional) Metadata regarding the ingredient itself.

4.1.1.5 EXIF metadata. The Exchangeable image file format (EXIF) is a common International standard format for storing metadata as part of an image. EXIF specification defines a set of tags related to various metadata related to assets (e.g. images, audio files). EXIF metadata categories range from image data characteristics (e.g. Color space transformation matrix coefficients) to image data structure (e.g. Artist, Copyright holder) and copyright information.

EXIF metadata can be found embedded in a digital asset. However, they can be easily stripped our modified without the permission of the image owner or even without knowing that such a modification has ever taken place. By defining an assertion type for EXIF metadata, it is possible for the provenance producers to specify the changes and ensure that some of this metadata won't be tampered with.

In addition, EXIF metadata might contain information that could raise privacy concerns. For instance, it is quite common that users of social media upload a picture, taken from their mobile phone, containing the GPS location information. Thus, the proposed specification should allow the possibility to redact or control the access of such information to the users that consume the provenance information of the shared digital asset. EXIF metadata constitutes the first assertion type introduced in the specification where both redaction and encryption functionality is supported.

EXIF metadata can be modeled as list of key-value pairs (e.g. "Exif.TagName": "ExampleTagValue"). Consequently, a JSON Content type JUMBF box could contain the EXIF metadata that the provenance producer chooses to assert in a provenance claim. In fact, it is highly recommended that the provenance producer application stores the EXIF metadata using the JSON-LD schema following the recommendations from XMP [20]. This would allow for a common structure of EXIF representation which would result in wider adoption of the proposed specification.

4.1.2 Assertion Store

Assertion Store is the JUMBF box with label 'mpms.prov.assertions' and Content Type UUID: 0x6D706173-0011-0010-8000-00AA00389B71. It should contain a list of all the Assertion JUMBF Boxes that shall be included in the claim.

In addition, a Manifest Producer has the ability to apply access rules in some Assertions. Hence, each of these Assertion JUMBF Boxes should be replaced by a Protection Box along with a JUMBF Box containing the exact access rules that shall be enforced. For instance, this external JUMBF Box should be a XML Content type JUMBF box containing the access rules using XACML [21].

Note: It is not allowed to encrypting Assertions that describe a major change in the asset. In other words, Protection Box representation is not applicable for assertions of type Action (see Section 4.1.1.2); it is applicable for EXIF metadata assertion.

4.1.3 Claim

Before defining the structure of a Claim, it is necessary that a new structure is defined. In Claim and Assertions (e.g. Ingredient Assertions) JUMBF Boxes there might be the need to reference another JUMBF box. This reference is achieved using the request definitions specified in ISO/IEC 19466-5. Specifically, the format is a URI composed of the following parts:

- A prefix stating that the URI is pointing at a JUMBF Box inside the same Manifest Store.
- The manifest unique Id number of the Manifest Content type JUMBF box that the referenced JUMBF Box resides.
- The label of each JUMBF box that wraps the referenced JUMBF box

An example of such a URI is depicted in listing 2. In this example, a reference is made to an Action assertion JUMBF Box located inside the Manifest Content type JUMBF box with label f977d216-ca0b-4e29-9298-89c8368e5cb9.

```
self#jumbf=mpms/urn:uuid:f977d216-ca0b-4e29-9298-89c8368e5cb9/mpms.
    assertions/mpms.prov.converted
```

Listing 2: Assertion URI

To ensure integrity - in the sense that the referenced JUMBF Box has not be tampered with after the point of the referencing action - a hash of the JUMBF bytestream needs to be computed. From now on, the set of the URI along with the hash of the referenced JUMBF Box and the hash algorithm used is called *URI Reference*.

A claim contains the following entities:

- a list of URI references of the assertions residing inside the Assertion Store,
- the URI reference pointing to the Claim Signature (see section 4.1.4),

- list of redacted assertions: a list of URIs mentioning the label of the Description box of those assertion JUMBF Boxes that were redacted from the provenance history of the digital asset,
- list of URIs mentioning the label of the Description Box of those assertion JUMBF Boxes that are protected (i.e. encrypted) inside the assertion store,
- Claim Generator description: a string that specifies the MIPAMS module along with the specific version where this claim is generated

A claim is represented by a JUMBF box with label 'mpms.prov.claim', Content Type UUID: 0x6D70636C-0011-0010-8000-00AA00389B71 and contains one JUMBF box of type (cbor).

4.1.4 Claim Signature

The actor (i.e. the producer), who produces a Claim, needs to digitally sign the Claim Content type JUMBF box bytestream and include the digital signature inside the Claim Signature. In addition a Claim Signature shall also include the public key that a consumer shall use in order to validate the signature.

Finally, it is of paramount importance to correspond the identity of a creator with the signature that has been generated. This can be achieved through the generation of a digital certificate for each Producer actor. This certificate should be generated using the key pair - specifically the private key - that is used by the Provenance Producer to sign the Claim. The generation of such certificates implies the need for establishment of trust anchors around the application.

In scope of RFC 8152 [22], there has been developed a standard way to structure the aforementioned information using CBOR serialization. This scheme is called CBOR Object Signing and Encryption (COSE) and this should be the standard way to express a Claim Signature content. A claim signature is represented by a JUMBF box with label 'mpms.prov.signature', Content Type UUID: 0x6D706373-0011-0010-8000-00AA00389B71 and contains one JUMBF box of type (cbor).

4.1.5 Credential Store

An application might want to correspond a specific actor with the assertions that are being registered in the provenance history of a digital asset. This doesn't mean that this actor is necessarily the one that generates the claim or owns the asset. It is a way to provide additional information for the actor that is related to the statements that are being asserted. The way to express this information is through W3C Verifiable Credentials [23] which provide the ability to specify information (i.e. statements/claims) about an actor and can be verified by a provenance consumer.

For this purpose a new JUMBF Box is introduced. A Credential Store (VCStore) is a JUMBF box that shall contain one or more JSON Content Type boxes (ISO 19566-5, Annex B.4). It shall not contain any other type of JUMBF box or superbox. It shall have

a label of ‘mpms.prov.credentials’ and a Content Type UUID: 0x6D707663-0011-0010-8000-00AA00389B71.

When storing W3C Verifiable Credentials in a VCStore, each one shall be labelled with the value of the id field of the credentialSubject of the VC itself. Since the id is guaranteed to be unique, this ensures that the URI to that credential will be unique.

4.1.6 Manifest

A manifest contains all the information fully describing a single point of the provenance history of an asset. It contains the following information

- Assertion Store: A JUMBF superbox containing all the assertion JUMBF boxes
- Claim: A JUMBF of CBOR Content type.
- Claim Signature: COSE Structure containing the signature of the actor (Timestamp is also provided)
- Optionally, the Actor’s Credentials to bind its identity with the the asset changes metadata

There are two types of Manifest entities: Standard and Update Manifests.

Standard Manifest means that the actor is modifying the asset. However, there are cases where an actor doesn’t modify the asset but simply wants to include additional metadata (e.g. EXIF metadata). In that case an Update Manifest is used. Active Manifest is defined as the latest (i.e. most recent) manifest that describes the current version of an asset.

Since there are two different types of Manifest Entities two different Content types shall be defined. The Standard Manifest shall be a JUMBF box with Content Type UUID: 0x6D70736D-0011-0010-8000-00AA00389B71. Similarly, for the case of Update Manifest, the respective JUMBF box shall have a Content Type UUID: 0x6D70756D-0011-0010-8000-00AA00389B71. In both cases, the label shall be a uuid uniquely identifying the digital asset. Its format shall be ”urn:uuid:¡uuid string¿”.

4.1.7 Manifest Store

Manifest Store is defined as the JUMBF superbox that contains a list of all the Manifest entities of a specific asset. It shall have a label ‘mpms.prov and Content Type UUID: 0x6D707374-0011-0010-8000-00AA00389B71.

A Manifest can reference other Manifest JUMBF boxes in the Manifest Store. The final structure of a Manifest Store can be depicted in the two following figure (figure 9):

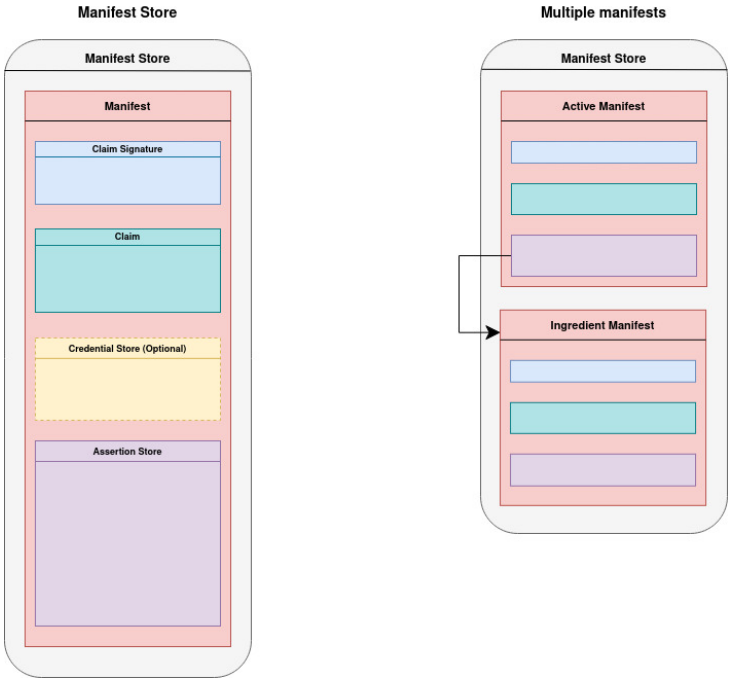


Figure 9: Component hierarchy (From Manifest Store to Assertion)

In the "Multiple manifests" example it is apparent how to express a chain of references describing the entire provenance history of an asset, where older manifests constitute ingredients of the active manifest.

4.2 Trust model

The definition of the trust signals that this specification requires are of outermost importance. In fact, none of the integrity and non-repudiation characteristics of a Manifest Content type JUMBF box have a meaning, if the identity of the signer is not assured. As specified in section 4.1.4, a digital certificate needs to be issued in order for an actor to generate provenance information. By digitally signing the claim using the same private key related to the certificate, a consumer can validate the identity of the actor who generates the assertions. This implies that as part of the operations described in the section 4.3 it is imperative to include the validity of the embedded certificates.

Thus, for any provenance application that follows this specification it is important to define the entities that are responsible for controlling/authorizing the ability to produce provenance information. These entities are called trust anchors. In this case, the trust anchors can be conceived as the Certification Authorities (CA) that provide the means to generate certificates for provenance producers as well as the interface which allows applications to query and validate the correctness of a certificate.

As clearly stated in C2PA technical specification, all the trust signals have as final destination the "Signer" which is the provenance producer. This trust relationship is depicted in figure 10

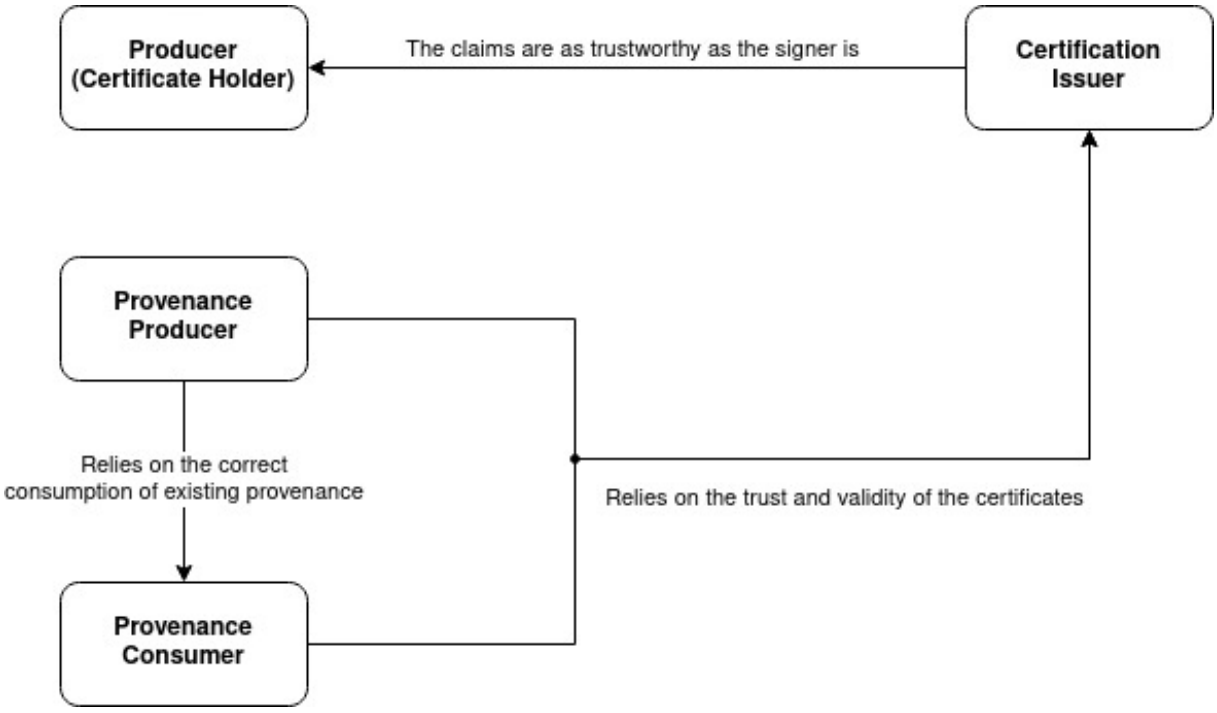


Figure 10: Trust relationship between provenance actors and entities

A simplified example of such a trust model could be applied in the case of journalism. A set of big journalist organizations around the world could play the role of trust anchors. All the international and national news agencies should be registered in at least one of them. A relationship of trust is established between them. News agencies could play the role of intermediate CAs that are authorized by the root CA to issue certificates for their entities. Eventually, this builds an entire chain of trust at the bottom of which there are the journalists.

To sum up, with these certifications chains it is possible to correspond the provenance claims with specific actors. The implementation of the specification presented in section 5 does not handle the validity check of certificates (e.g. check certificate's signature, expiration status, revocation status). It is assumed that the validation of the producers' certificates is handled by the application that uses the mipams-fake-media library.

4.3 Operations

This section describes the logic of the two core operations that a user can perform with provenance metadata. Specifically, two algorithms shall be presented, one related to consuming and one to producing provenance metadata. It is worth mentioning that in both algorithms the provenance information is provided separately to the digital asset. The goal is to support use cases and applications that embed provenance information into the digital asset as well as those that store this metadata outside of the digital asset. Regarding the first case, the specification assumes that the application shall handle the extraction and stripping of the provenance metadata from the digital asset before using

the following algorithms.

4.3.1 Consume provenance metadata

The first operation defines how to consume provenance metadata related to a digital asset. Provenance metadata is stored in the form of a Manifest Store Content type JUMBF box as defined in section 4.1. Depending on the use case the consumer application might select to consume the entire provenance history that is stored in the asset (i.e. all the Manifest Content type JUMBF boxes inside the Manifest Store) or simply the Active Manifest Content type JUMBF box assuming that the rest of the provenance chain is already validated. In general, validating a Manifest consists of validating its Claim Signature as well as the integrity of the assertions through the assertion references located in the Claim. In the case of a Standard Manifest, validation includes also the validation of the content binding assertion that verifies the connection between the Manifest and the digital asset. As explained in section 4.1.1.1, in the case where the Active Manifest is an Update Manifest then the content binding assertion shall be retrieved from the first parent Standard Manifest.

Throughout this operation, the term "digital asset" refers to the image codestream without the Manifest Content type JUMBF box. It is of paramount importance that every time the digest of an asset is calculated, the provenance metadata bytestream is not taken into consideration. Consequently, the application that uses this algorithm shall strip the provenance metadata beforehand. With this approach it is possible to avoid the circular dependency where the digest of the asset assumes that the provenance information is finalized and embedded into the asset. However, for this to happen, the digest must have been already computed.

Supposing that the consumer selects to fully inspect the provenance history of an asset, it is implied that the Active Manifest contains Ingredient assertions referencing Ingredient Manifests. These manifests need to be inspected as well and this procedure is performed recursively until all the provenance chain is validated. The algorithm summarizing the aforementioned steps is depicted in algorithm 1.

Algorithm 1 Consume Provenance Metadata

```
1:  procedure INSPECT-PROVENANCE(manifest_store, asset, is_full_inspection)
2:      active_manifest ← locateActiveManifest(manifest_store)
3:      is_standard ← isStandardManifest(active_manifest)
4:      if is_standard then
5:          verifyManifestIntegrity(active_manifest)
6:          verifyContentBinding(active_manifest, asset)
7:      else
8:          while not is_standard do
9:              manifest ← locateParentManifest(active_manifest, manifest_store)
10:             is_standard ← isStandardManifest(manifest)
11:             verifyManifestIntegrity(manifest)
12:         verifyContentBinding(manifest, asset)
13:     to_be_checked ← []
14:     if is_full_inspection then
15:         to_be_checked ← addComponentIngredients(active_manifest)
16:         while isEmpty(to_be_checked) do
17:             manifest_reference ← to_be_checked.next()
18:             manifest ← locateManifest(manifest_reference, manifest_store)
19:             verifyManifestIntegrity(manifest)
20:             to_be_checked ← addComponentIngredients(manifest)
21:     result ← active_manifest
22:     return result
23: procedure VERIFY-MANIFEST-INTEGRITY(manifest)
24:     validateClaimSignature(manifest)
25:     validateAssertionIntegrity(manifest)
```

4.3.2 Produce provenance metadata

Producing provenance metadata corresponds to any addition of provenance point in the history of a digital asset. This means that there are multiple use cases that need to be taken into consideration. First of all, the algorithm starts by providing the current state of an asset along with its provenance history that has been recorded so far. On top of that, the producer application that calls the algorithm, needs to provide the asset after the new modifications have taken place as well as the assertions that describe them. These assertions could be any assertion out of the ones defined in section 4.1.1. The assertions must be provided as a list of JUMBF Boxes ready to be embedded in the Assertion Store. This allows the producer applications to perform any encryption and apply any access rules policy they fit proper for their use case. Regarding the redaction process, the producer application shall provide a set of URIs referencing the assertions that shall be redacted. These references point to assertions located in ingredient manifests inside the asset's manifest store. Not all assertion types can be redacted. Specifically, the assertions that are able to be redacted are the ones that do not directly affect the content of the image. For instance, an action assertion is not allowed to be redacted while an EXIF assertion is. Finally, in case the producer application wants to provide Verifiable Credentials corresponding the assertions to an entity, the Credential Store Content type JUMBF box must be provided to the algorithm in advance.

Now that all the possible input parameters have been described, the algorithm starts by checking whether there is provenance history registered to the digital asset describing its modifications so far. In this case, before the new manifest is produced it is crucial that at least the current active manifest of the digital asset is verified to ensure that the new manifest is not added on top of a corrupted or malformed provenance chain. In addition, it is also important to verify that the assertions that compose the new claim contain the proper parent ingredient assertion referencing (i.e. URI reference) the current active manifest.

Next, the new manifest shall be produced. Depending on the type of the manifest (i.e. depending on the assertions provided), a content binding assertion might be needed in order to relate the new manifest to the revised digital asset. Since all the assertions are provided in the form of JUMBF Boxes, they need to be wrapped with an Assertion Store Content type JUMBF box. Then the Claim Content type JUMBF box shall be constructed consisting of all the fields described in section 4.1.3. Subsequently, the producer application shall be able to digitally sign the constructed Claim using the private key that is used to generate the digital certificate corresponding to the producer actor who builds this provenance point. All these boxes - along with the Credentials Store box, provided that the producer wants to include one - are wrapped inside a new Manifest Content type JUMBF box which constitutes now the latest provenance point in the digital asset's provenance chain.

Moreover, the algorithm needs to take into consideration the use cases where the producer includes a set of ingredients assets that have their own provenance history. In that case, the active manifest of these ingredient manifest stores are validated and included in the digital asset's manifest store.

Finally, now that the manifest store is updated, the algorithm needs to handle the "Redact assertions" request. This request is related to the "redactable" assertions of the ingredient manifests that reside inside the Manifest Store Content type JUMBF box. The redacted assertions, that are already included in the new active manifest's claim, need to be removed from the ingredient assertions of the entire provenance history of the digital asset. The manifest store is, eventually, complete and the summarized algorithm of producing it is depicted in algorithm 2.

Algorithm 2 Produce Provenance Metadata

procedure PRODUCE-PROVENANCE($CA, NA, MS, AL, RL, IM, signer, CS$)
    // CA: Current Asset, NA: New Asset,
    // MS: Manifest Store, AL: Assertion List,
    // RL: Redaction List, IM: Ingredient Manifest List,
    // CS: Credentials Store
    if MS exists **then**
        $Provenance - Consume(CA, False)$ ▷ See Algorithm 1
        $active\_manifest \leftarrow locateActiveManifest(MS)$
        $checkParentIngredientAssertion(AL, active\_manifest)$
    else
        $MS \leftarrow createEmptyManifestStore()$
    $new\_active\_manifest \leftarrow Produce - Manifest(NA, AL, RL, signer, CS)$
    if IM exists **then**
        $validateUriReferences(AL, IM)$
        $MS \leftarrow MS + IM$
    if RL exists **then**
        $MS \leftarrow performRedaction(MS, RL)$
    $MS \leftarrow MS + active\_manifest$
    $return\ MS$
procedure PRODUCE-MANIFEST($asset, AL, RL, signer, CS$)
    $manifest\_id \leftarrow issueNewUuid()$
    $manifest\_type \leftarrow discoverManifestType(AL)$
    if $manifest\_typeis'StandardManifest'$ **then**
        $ABox \leftarrow buildAssertionStoreWithContentBindingAssertion(asset, AL)$
    else
        $ABox \leftarrow buildAssertionStore(AL)$
    $CBox \leftarrow buildClaim(manifest\_id, ABox, RL)$
    $CSBox \leftarrow buildClaimSignature(signer, CBox)$
    $manifest \leftarrow buildManifest(ABox, CBox, CSBox, CS)$
    $return\ manifest$

4.4 Fulfillment of JPEG Fake Media Requirements

This section explains how the proposed specification covers the set of JPEG Fake Media requirements listed in Table 3.

First and foremost, with the definition of various assertions and the ability to specify additional metadata around them (e.g. date, metadata, software agent fields), the proposed specification meets requirement R1.1 related to providing description about the modifications that are taking place in a digital asset.

Secondly, it offers the ability to keep track of the provenance history of a digital asset either by embedding the Manifest Store Content type JUMBF box in the digital asset itself or by storing it in a Manifest Repository on the cloud (R1.8).

Next, the fact that jumbf-core library supports the extraction of JUMBF metadata in a separate .jumbf file, it allows for Provenance information to be stored separately as a self-contained structure (R2.7).

By allowing Protection Content Type JUMBF Boxes to describe a protected (e.g. encrypted) Assertion, the proposed specification is compliant with ISO/IEC 19566-4: Privacy & Security standard (R2.10). Moreover, it takes into consideration the privacy of both individuals and locations by allowing assertions related to EXIF metadata to be protected (R2.3).

Subsequently, the proposed specification offers the choice to a provenance producer to redact a subset of assertions (e.g. EXIF metadata) (R2.5).

When the need of a cryptographic tool is required - e.g. computing digital signatures or calculating the hash of a JUMBF box - the algorithm that is used is also recorded. This is done in order to allow for the adoption of more than one hashing and signing methods (R.2.8.1 and R.2.8.2).

Finally, related to the proposed provenance operations for producing and consuming provenance information, it is part of the specification to verify the integrity and authenticity of the referenced digital asset through the content binding assertion which is required in every standard manifest (R3.3 and R3.5).

5 Provenance Reference Software

In section 4, a new specification has been introduced focusing on the JPEG Fake Media initiative. In this section, a reference software called mipams-fake-media is presented showcasing how jumbf-core proposed reference software can be extended in order to support the new proposed specification. Both the specification and the library shall be part of our contribution to the JPEG Call for Proposals [7] related to the exploration a new standard addressing the problem of Fake Media. The mipams-fake-media library can be found in the following URL: `https://github.com/nickft/mipams-fake-media`

Based on the definitions proposed in section 4 it is evident that a set of Provenace Content Type JUMBF Boxes need to be defined. The components required to define those JUMBF Boxes (i.e. Description and Content Boxes) are already supported in jumbf-core-2.0 and jumbf-privsec-1.0 libraries. Consequently, what needs to be implemented is the Content Type classes as illustrated in the "Provenance Module" in figure 11.

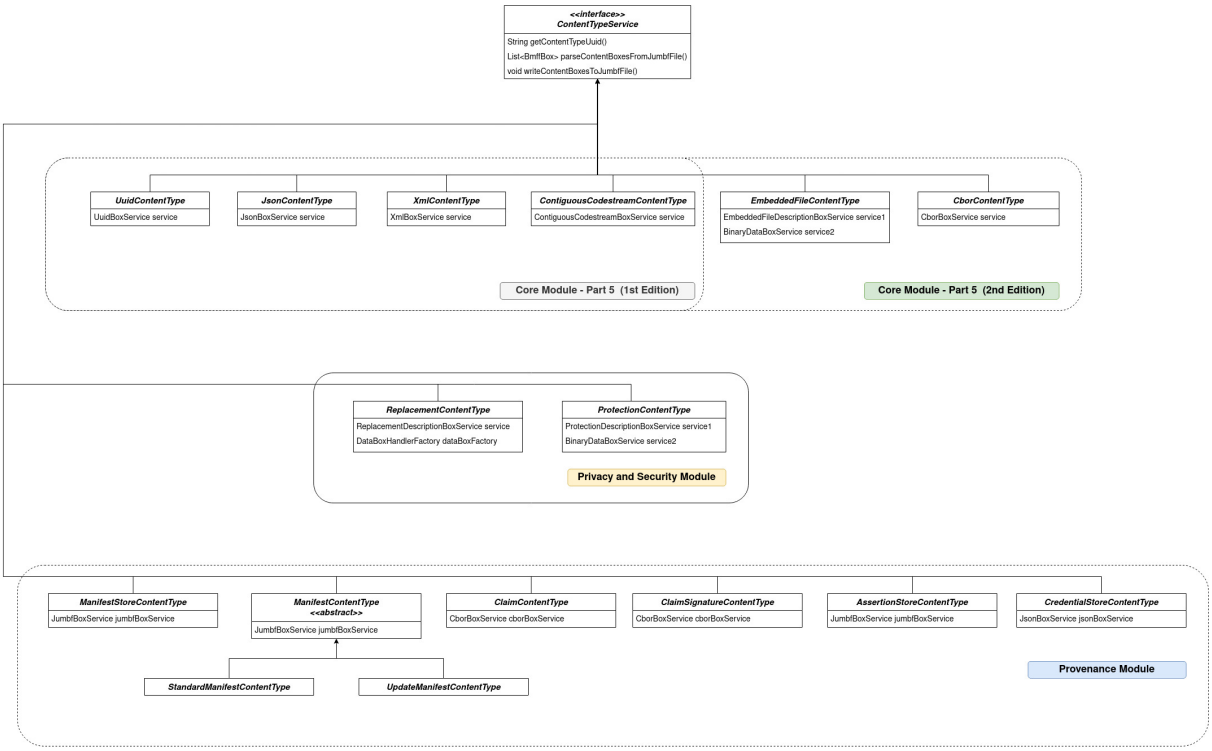


Figure 11: Extended Content Type class hierarchy containing the Provenance module

At the bottom level of figure 11, the Provenance Content Types are defined. It's worth noting that each Content Type uses a Box service that is already defined in the jumbf-core-2.0. Specifically, in the cases of ClaimContentType, ClaimSignatureContentType and CredentialStoreContentType classes only one specific box service is required. However, this is not the case for the remaining Provenance Boxes. As explained in section 4.1, the content of an Assertion Store Content type JUMBF box is a list of JUMBF Boxes that is of type CBOR, JSON, Embedded File or Protection. This is the reason why a

JumbfBoxService field is assigned for the AssertionStoreContentType class. Regarding the Manifest Content type JUMBF box, it contains a set of JUMBF Boxes (i.e. Claim, Claim Signature, Assertion Store and - optionally - one Credential Store). Thus, the only Box Service that is required is the JumbfBoxService. Last but not least, the same applies for a Manifest Store Box which contain a list of Manifest Content type JUMBF boxes.

In addition to the new JUMBF Box definitions, the provenance operations specified in 4.3 are implemented as well. Different service classes are implemented for each part of the "Consume" and "Produce" provenance operations. As in the case of Content Type classes, they are implemented as Java Beans facilitating the dependency injection for each of these classes. The dependency graph of the service classes for each of the two provenance operations is depicted in figures 12 and 13.

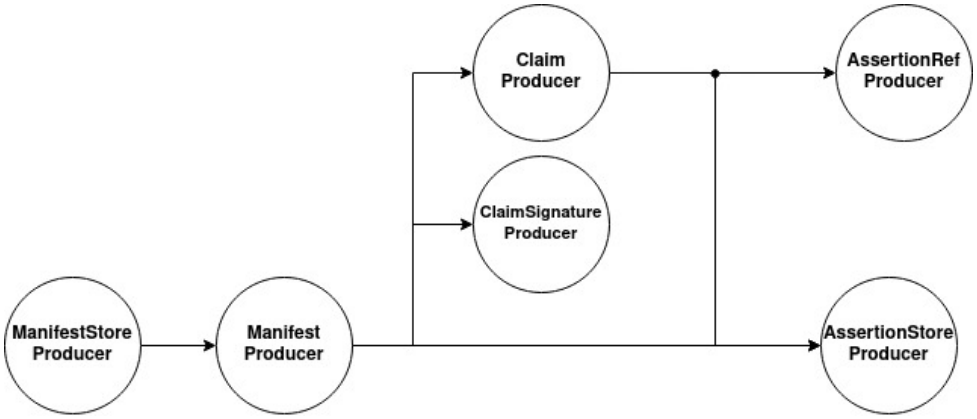


Figure 12: Producer services dependency graph

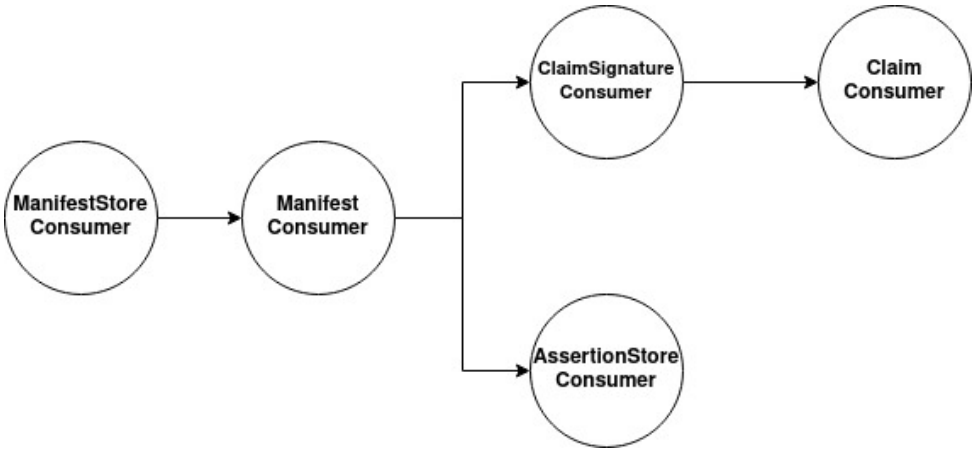


Figure 13: Consumer services dependency graph

Both "Consume" and "Produce" provenance operations require a set of cryptographic tools in order to handle digital signatures, encrypted assertions etc. As a result, a new library called "crypto" is introduced in mipams-jumbf project that performs a set of

cryptographic operations (encrypt, decrypt, sign, validate signature, parse keys from files). It has been decided to implement these operations in a different submodule in order for additional projects to use these functionalities. These cryptographic operations are implemented using the Java Cryptography Architecture (JCA) [24]. Currently in jumbf-crypto-1.0 the following cryptographic operations are supported:

- Generating and Validating digital signatures: SHA1 with RSA
- Encrypting/Decrypting a byte stream: AES-256
- Calculating the Hash of a byte stream: SHA256

Finally, in mipams-fake-media library the set of assertions is specified according to section 4.1.1. Although the provenance operations are implemented handling the assertions as JUMBF Boxes, it is imperative that this library defines the structure of the content of each Assertion JUMBF Box. Moreover, all the serialization and deserialization operations (i.e. extracting information from the JUMBF Box Content Boxes or storing Assertion information into a JUMBF Box) are implemented as well. The class hierarchy of the Assertion classes is depicted in figure 14. Notice that there are two interfaces, namely RedactableAssertion and NonRedactableAssertion that assist with the definition of each assertion class. As of mipams-fake-media-1.0 only ExifMetadata assertion is considered to be redactable.

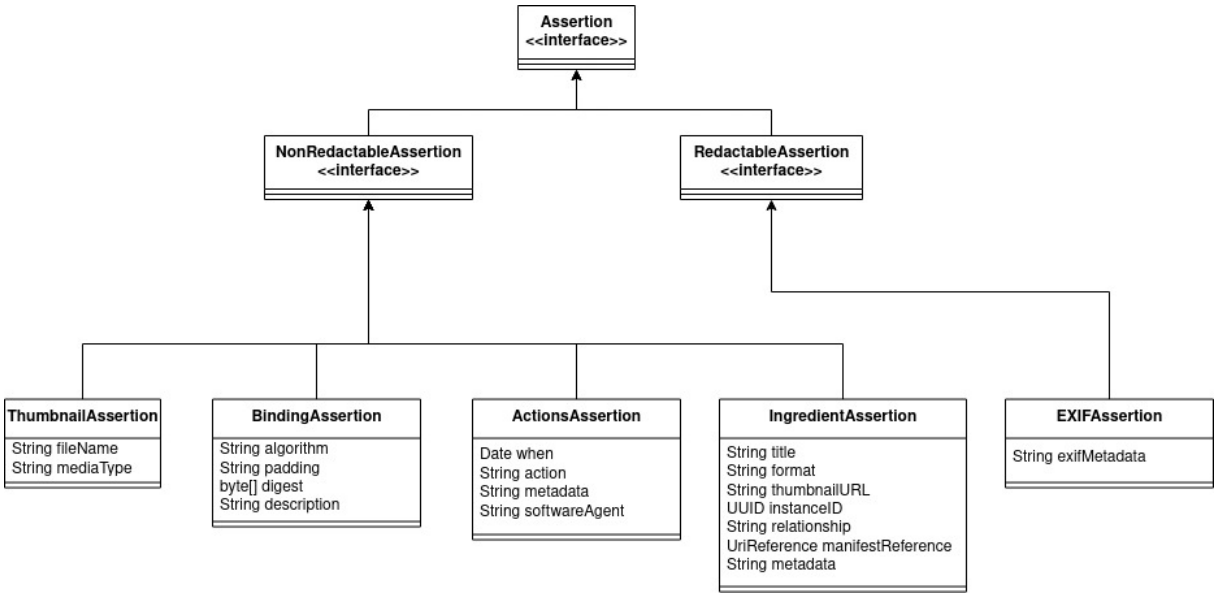


Figure 14: Assertions class hierarchy

6 Application

This section presents two demonstrator applications. The first one - "Practical demo for JUMBF metadata" - focuses on assisting developers getting familiar with the structure and the concepts behind JUMBF metadata. The second application focuses on providing a scenario where users can produce and consume provenance information following the specification presented in section 5.

6.1 Practical demo for JUMBF metadata

This demonstrator application provides a graphical user interface for parsing and generating JUMBF metadata. On the one hand, the application generates JUMBF metadata in separate files with .jumbf extension. On the other hand, it supports parsing JUMBF metadata from both .jumbf files and JPEG images.

This demo application consists of two services: the RESTful API server that uses the jumbf-core-2.0 library and the client application that provides the GUI implemented using React JS [25]. For this application two services are launched using docker [26] containers. As shown in figure 15 the one docker container launches the Practical Demo GUI while the other container is managing the demo server. The demo-server implements the RESTful controller for communicating with the front-end and contains the services that are responsible for integrating the two mipams-jumbf libraries (i.e. jumbf-core-2.0 and jumbf-privsec-1.0) as well as the module that parses the JSON syntax responsible for generating JUMBF Boxes. For more information on how to run the demo application visit the Github repository attached in the Introduction section.

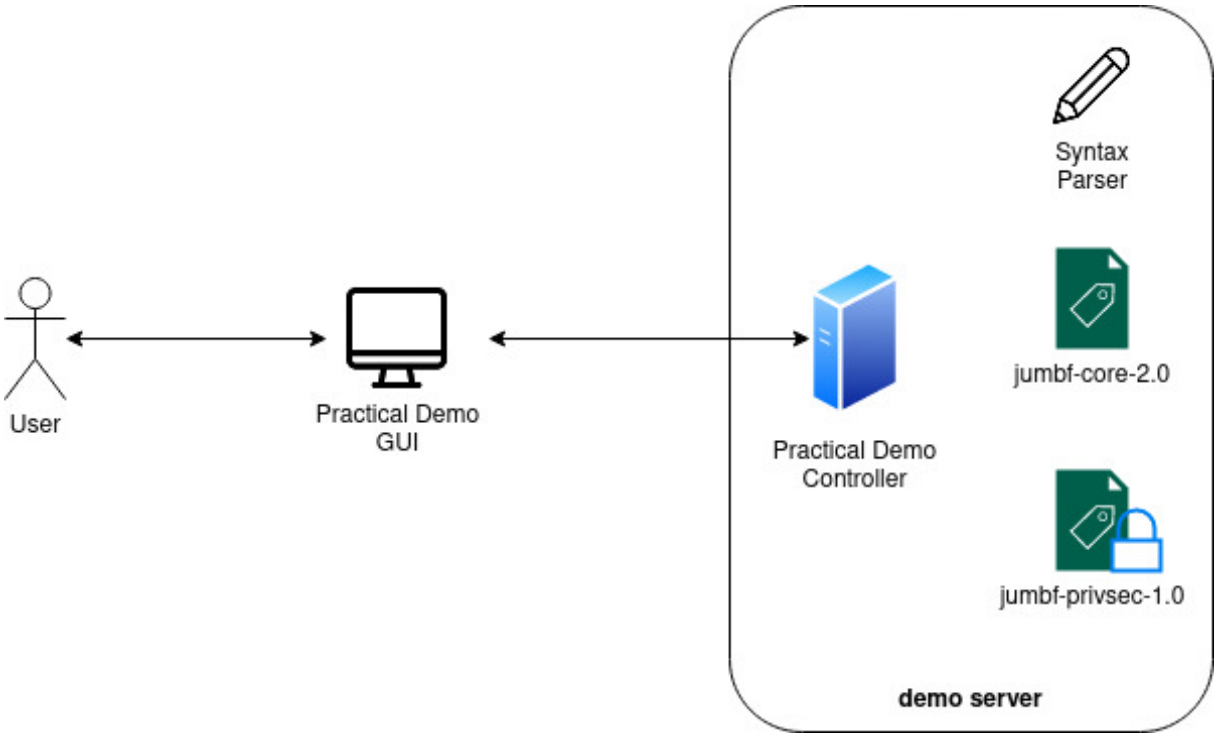


Figure 15: Practical Demo service architecture

Once both services are up and running a user can open a browser and connect to the GUI as shown in figure 16.

Figure 16: Main page of the application

The GUI is divided into two main components: One the left side, the user interacts to Generate a JUMBF file while on the right side, the user uploads a JUMBF File to inspect its contents.

Firstly, the generation of a JUMBF file is explained. For the scope of this demo application, a simple syntax format has been defined in order for the user to express a JUMBF structure in JSON. An example is shown in figure 17.

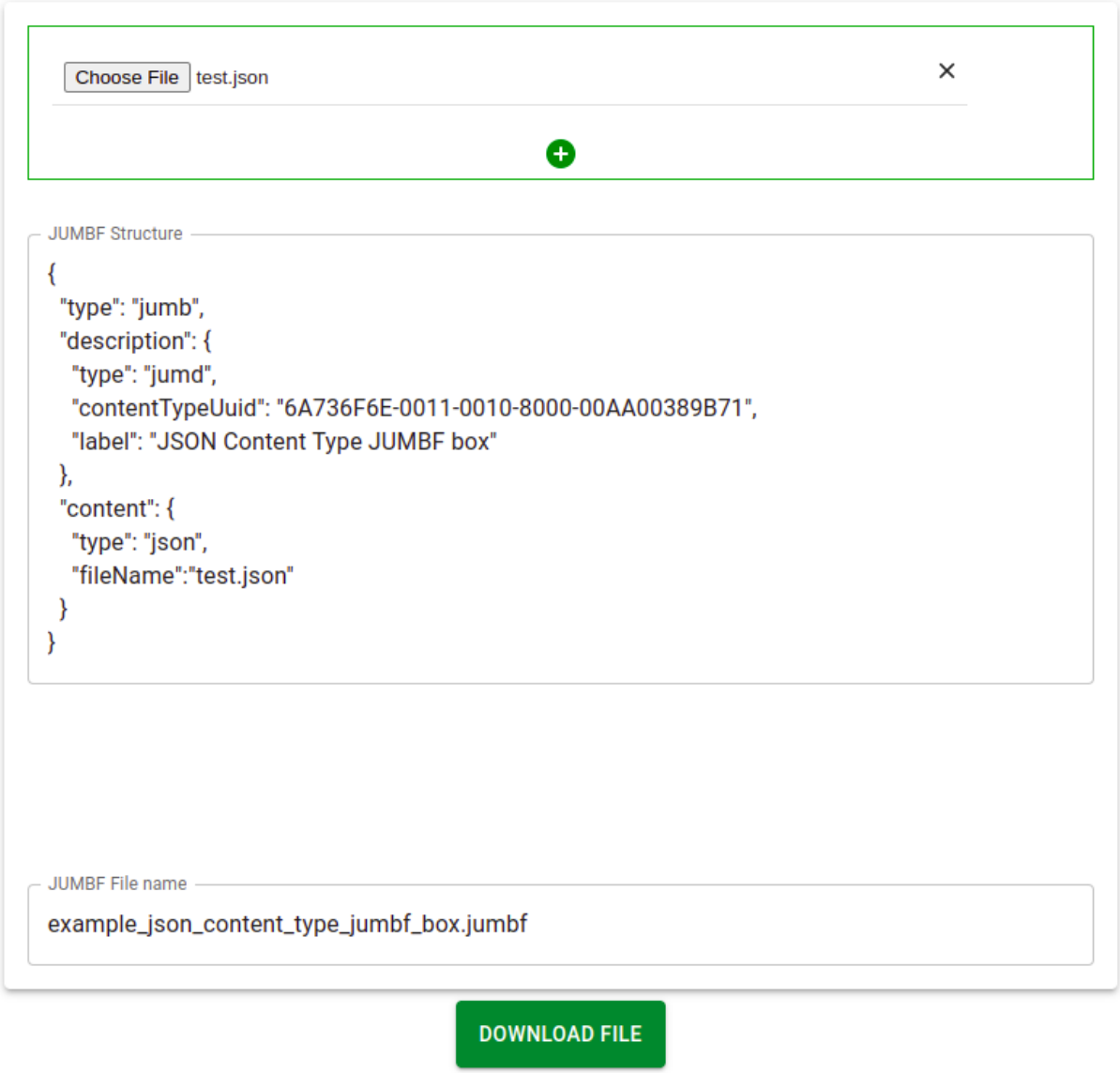


Figure 17: Generating a JSON Content type JUMBF box

Notice that the user has initially specified the file that needs to be uploaded to the application containing the metadata in JSON format (i.e. test.json). Subsequently, the she uses the application's supported syntax to express a JUMBF box with Content Type UUID corresponding to the JSON Content Type and an example label. According to the "content" attribute, the content box of this JUMBF Box is a JSON box that contains the information located in the uploaded file. Finally, the user specifies the name of the target JUMBF file and clicks on the "GENERATE JUMBF FILE". The process has been

performed successfully and a green “DOWNLOAD FILE” button appears, allowing the user to download the generated JUMBF file.

Now this JUMBF file can be provided as an input to the parser functionality. From figure 16, it is observed that there is a button at the right panel that says “UPLOAD JUMBF FILE”. By clicking on that button, the user is asked to select a JUMBF file to upload. Upon selection of the JUMBF file, the application parses its structure and provides a hierarchical structure explaining the ISOBMFF header as well as the fields included in each box. In this example the result of the parsing is depicted in figure 18.

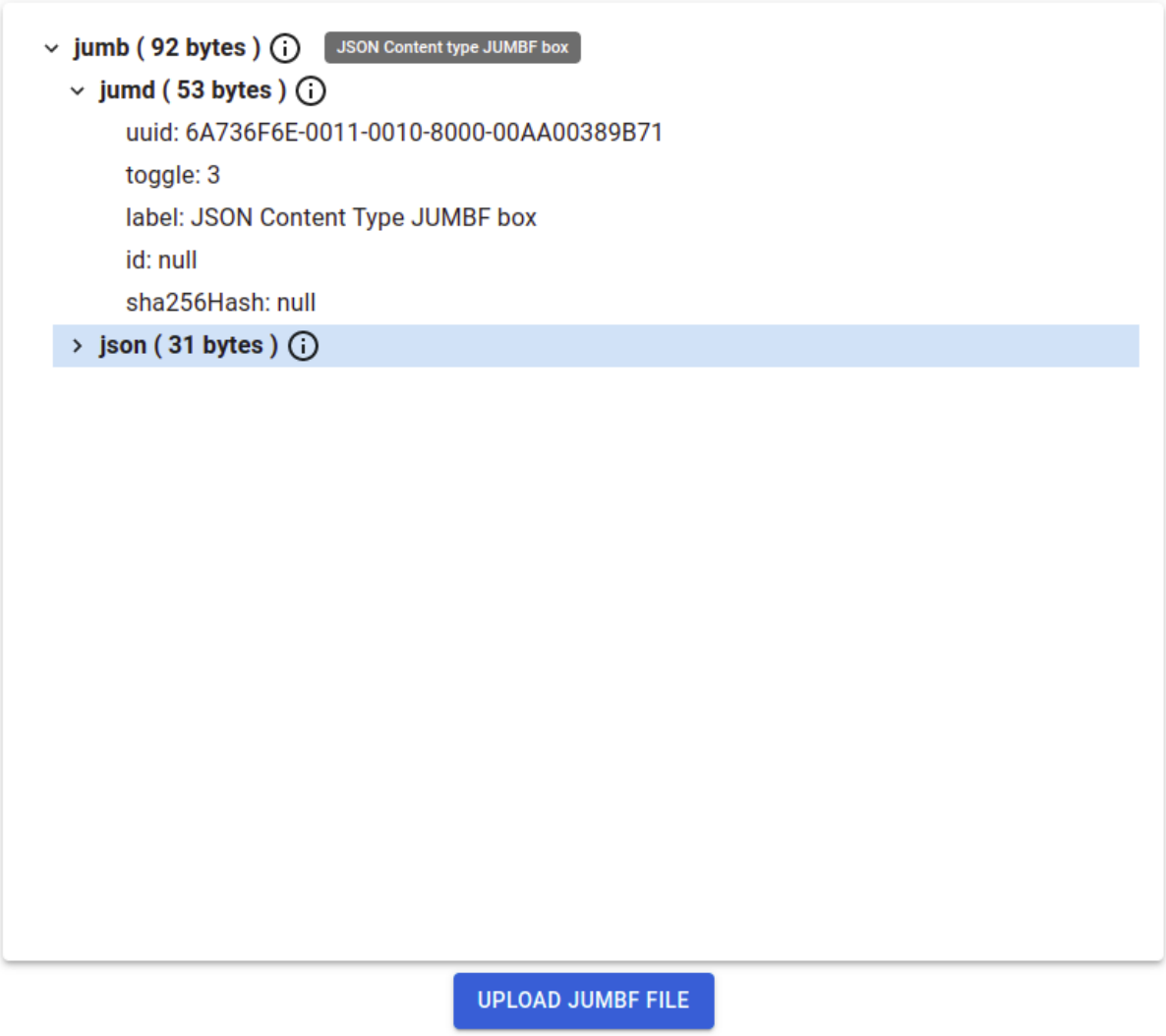


Figure 18: Parsing a JUMBF file

The first level of information that can be extracted is the ISOBMFF type of each box along with its size (expressed in bytes) and an information logo providing a short description for the box. In the case of a JUMBF box this description explains the Content Type of the JUMBF Box while in any other case the description of the ISOBMFF TBox field is

provided. Finally, the supported fields of each Box structure are depicted as well.

6.2 Provenance metadata manager

6.2.1 Introduction

This section illustrates an example application that allows authorized users (i.e. producers) to create provenance information for a specific digital asset. This provenance information shall be embedded as a Manifest Store Content type JUMBF box inside the digital asset which can be then shared across multiple users that shall try to view (i.e. consume) it. The goal of this application is to focus on the use cases where a producer wants to protect/control the access to a set of metadata that could disclose personal information such as GPS Location. For this reason, a set of roles have been defined for the authenticated users of this application: Consumers and Producers. Based on that, three workflows are examined in the application: first, the producer is not interested in protecting EXIF metadata that contain GPS and device specific information. In the second use case, the producer selects to encrypt the existing EXIF metadata and make it accessible only to the authenticated consumers. In this case, access rules are implemented at software level in the sense that the application shall decide whether a user has access to a protected resource. Finally, the producer wants to apply access rules in order to control the access to this piece of metadata such that only fellow Producers can access this information. This application is implemented under the project called mipams-fake-media-demonstrator and it provides a demonstration of the functionalities of the mipams-fake-media library. Through the analysis of these use cases a detailed explanation of the software design will be provided as well. The entire provenance demonstrator application can be inspected by visiting the following URL: `https://github.com/nickft/mipams-fake-media-demonstrator`

This provenance metadata manager application is developed using Java and Spring boot framework [16] for the RESTfull API, while the front-end application is written using React JS [25]. These services are configured and initialized as docker [26] containers. The interconnection between those services is implemented using a tool called docker-compose. In figure 19 the two service, for the front-end application and the back-end application are depicted. The server service consists of a RESTfull Controller which provides the endpoints that the GUI can access. In addition, it implements those service classes that integrate the four core library dependencies, namely the mipams-fake-media-1.0 library to access the provenance data model and operations, the jumbf-crypto-1.0 library to access the cryptographic operations as well as the two JUMBF-related libraries jumbf-core-2.0 and jumbf-privsec-1.0.

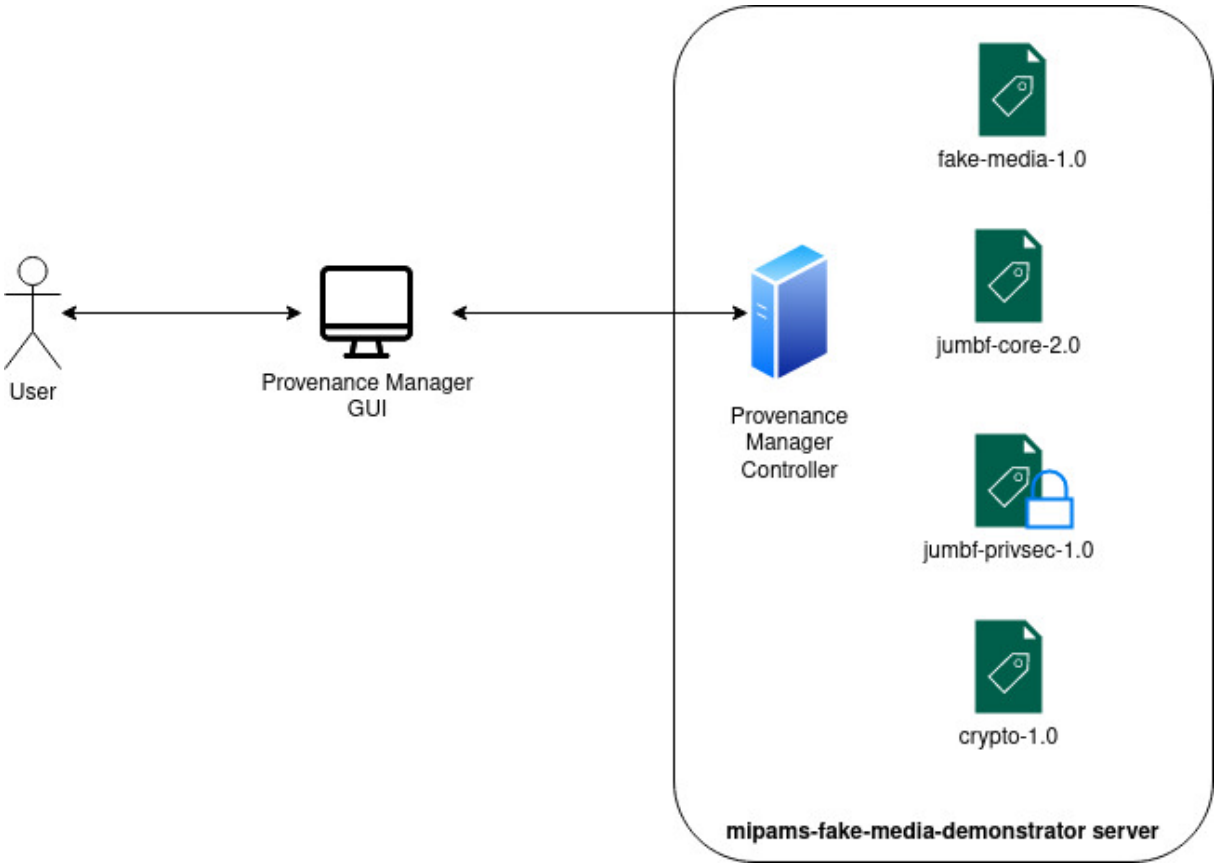


Figure 19: Provenance Manager service architecture

An additional service is also executed that runs along with the aforementioned ones. Specifically, this service is called "authorization-service" and it is responsible for validating whether a user (i.e. a Consumer) is authorized to access the EXIF metadata of the digital asset under inspection. As explained below, this authorization service performs this validation using XACML [21], a standard developed by OASIS, that defines a declarative fine-grained, attribute-based access control policy language expressed in XML. All the cryptographic operations - i.e. encryption/decryption of a bytestream, creation/validation of digital signatures - including the XACML policy creation and enforcement are implemented in a separate library called jumbf-crypto-1.0. This library is part of the mipams-jumbf project.

This demonstration software simulates an application related to the journalism field where producers are members of journalist associations. Upon registration to the application, journalists (i.e. producers) have received a key pair and a digital certificate. As explained in section 4.2, the digital certificate is crucial for the producer in order to generate - i.e. digitally sign - provenance information.

6.2.2 Generating producer key pair and digital certificate

When the application is launched a set of users is initialized. Specifically, in this example two producer users have been created and each one of them belongs to a different journalist organisation: user "nickft" is a journalist from CNN while user "reporter1" is a journalist from BBC. Finally, a user "user" is created having only consumer capabilities in the application. These users are created and stored in-memory using Spring Boot Security framework class "InMemoryUserDetailsManager". This configuration takes place in org.mipams.fake_media.demo.config.WebSecurityConfig.java class.

During the creation of the producers, a script called "generateCredentials.sh" is generated that is responsible to generate the key pair for each producer along with its digital certificate. For the scope of this demonstration the validity of the certificates is not covered; digital certificates are assumed to be valid. For each producer, two files are stored. The first one has the format [username].priv.key and corresponds to the generated RSA [27] private key in Distinguished Encoding Rules (DER) encoded format [29]. The contents of an example RSA private key destined for user "nickft" are depicted in figure 20. Notice that the as depicted in the figure, the length of the key is set to 4096 bits and the file itself contains all the parameters that are required for RSA to work properly (e.g. modulus, private/public exponents). As for the second file, it has the format [username].crt and corresponds to the X.509 Certificate issued for a particular producer. This certificate has been issued using the private key included in the [username].priv.key file. An example of the certificate corresponding to "nickft" producer is depicted in figure 21. Notice that in both "Issuer" and "Subject" field the value "OU=CNN" is set. These attributes show the entity that issued the certificate as well as the entity to whom this certificate is issued for. In a real-case scenario these entities wouldn't be the same. According to the trust model in section 4.2 the Issuer could be a Journalist Association and the Subject would be the organization - in our case CNN - that this certificate is issued for. However, in scope of this example there is no Certification Authority issuing certificates for the news organizations meaning that the generated certificates need to be self-signed.

```
$ openssl rsa -noout -text -inform DER -in nickft.priv.key
RSA Private-Key: (4096 bit, 2 primes)
modulus:
    00:fa:07:a1:fa:d6:36:4b:07:66:d0:2e:7a:c1:4b:
    37:6d:dc:b5:c2:02:45:af:05:57:de:61:74:95:21:
    df:da:36:c1:7a:aa:bf:db:89:d1:98:50:d8:e8:bb:
    a8:6e:b5:6e:29:13:3a:15:5c:26:58:9a:10:e8:b5:
    0e:e9:96:db:26:98:d4:26:0b:41:01:2d:a7:88:e0:
    11:8b:e4:b7:7f:96:cd:b9:c4:eb:61:70:c8:f6:c4:
```

Figure 20: Parse RSA private key context using openssl

```
$ openssl x509 -in nickft.crt -noout -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 182840380 (0xae5ec3c)
        Signature Algorithm: sha384WithRSAEncryption
        Issuer: CN = CNN, OU = CNN, O = MIPAMS
        Validity
            Not Before: Jun 21 13:33:11 2022 GMT
            Not After : Sep 19 13:33:11 2022 GMT
        Subject: CN = CNN, OU = CNN, O = MIPAMS
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (4096 bit)
                Modulus:
                    00:fa:07:a1:fa:d6:36:4b:07:66:d0:2e:7a:c1:4b:
```

Figure 21: Parse X.509 Certificate context using openssl

6.2.3 Producing provenance history for digital assets

In order for a producer to add provenance information to a digital asset, she must first authenticate herself. Once authenticated, the producer shall receive a JSON Web Token (JWT) [30] that she can use in order to access the protected resources accessible only by producers. Specifically, inside the JWT claims a custom claim "role" is added containing the list of the roles that the authenticated user has. Then, during a REST request to a protected resource the JWT must be included in the Authorization header of the HTTP request. The login page of this application is depicted in figure 22. Upon successful authentication the JWT is received and stored in a browser cookie - as shown in figure 23. Now the producer has access to the "Mipams for Producers" functionalities.

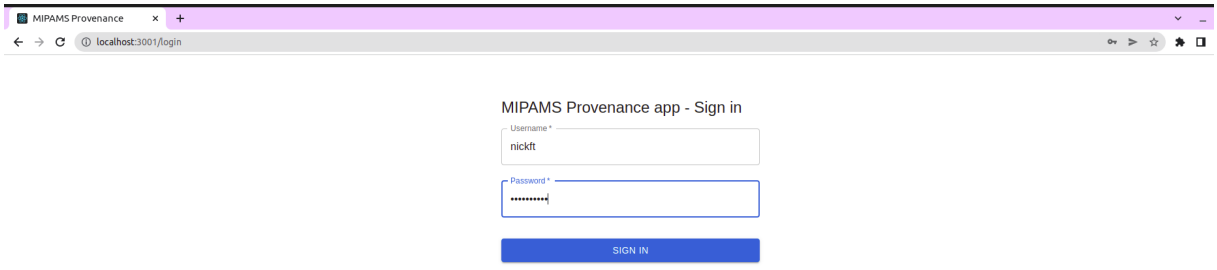


Figure 22: Producer "nickft" logs in to application

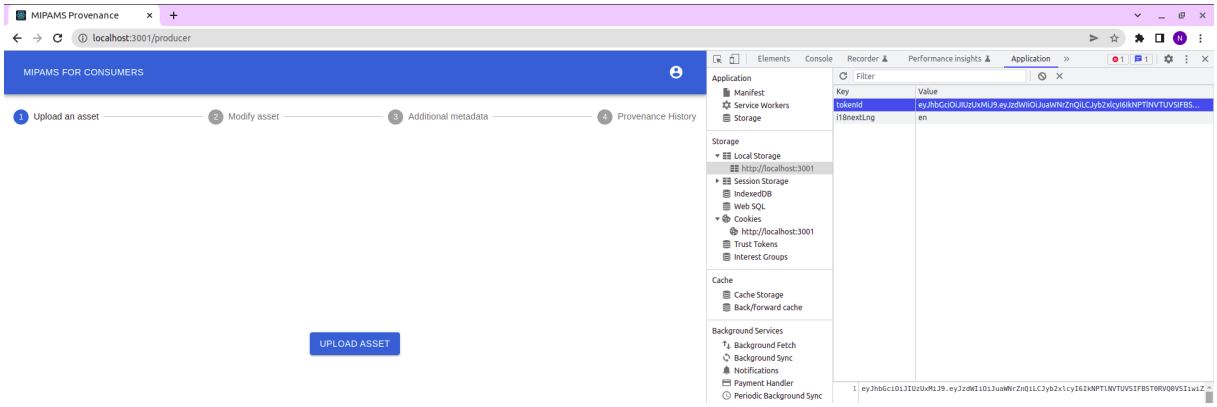


Figure 23: Left screen: Producer is logged in, Right screen: JSON Web Token "tokenId" is stored in a browser cookie

Subsequently, the producer clicks on the "Upload Asset" button to start the provenance generation workflow. For this example, a JPEG Image taken from a camera device is selected. Once uploaded, an Image Editor is presented and the image is loaded as shown in figure 24. The image editor is part of the Toast UI ImageEditor project [31].

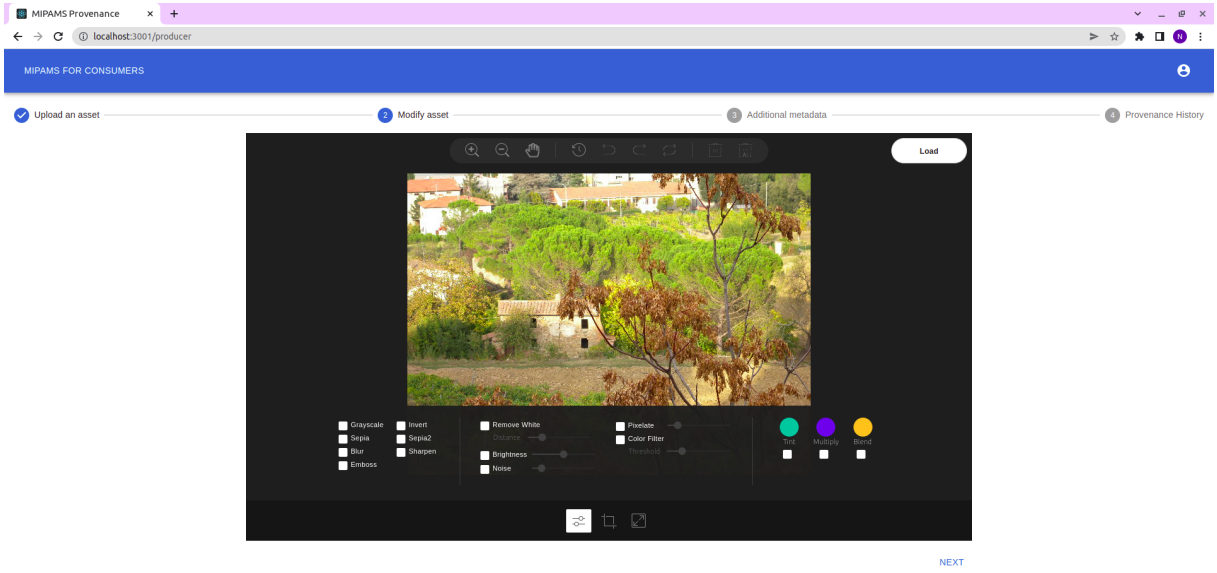


Figure 24: A digital asset loaded in the provenance image editor

The producer "nickft" has selected to perform the following set of modifications: Filter the image (adding Sepia filter and reducing the brightness), crop the image and change its size. The resulted image is depicted in figure 25. The producer advances to the next step by clicking on the "Next" button at the bottom right of the screen.

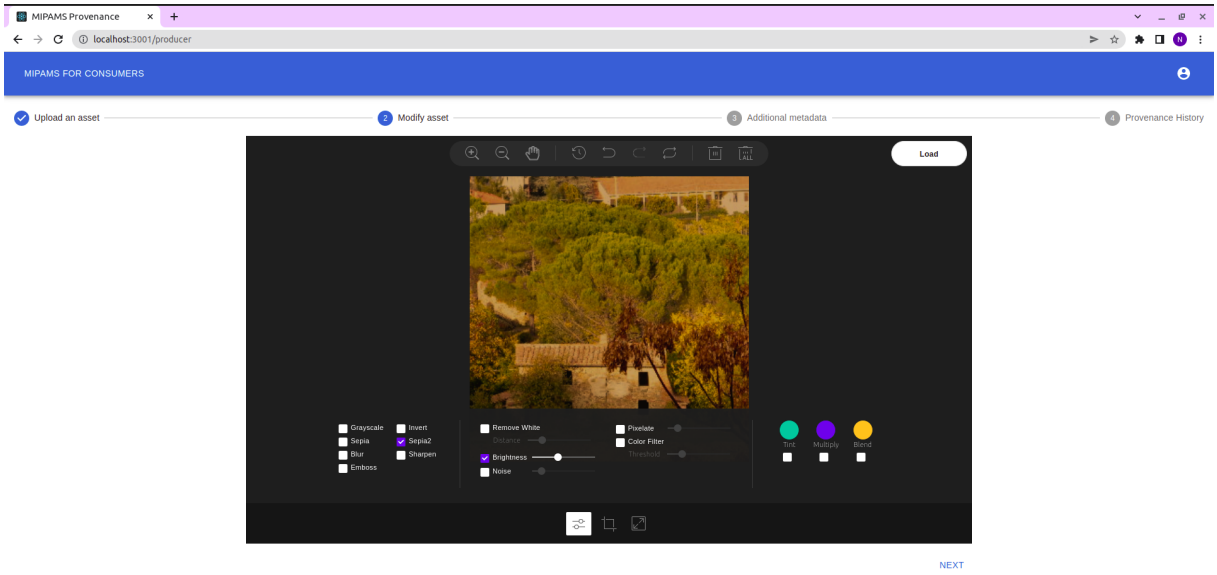


Figure 25: The uploaded digital asset upon modifications

At this stage, the application displays the EXIF metadata that are included in the image. This metadata has been extracted from the original uploaded image using a tool called ExifTool 12.42 [32]. This tool is already installed in the container that launches this application and it is invoked by a Java Bean class called "ExifToolService". As shown in figure 26, this metadata contains information that the producer might select not to disclose to all the users who want to view this asset. As depicted in the figure below, the producer has three options: (i) include EXIF metadata unprotected in the Assertion Store Content type JUMBF box, (ii) encrypt them as a Protection box that shall be accessible only by authenticated users of this application or (iii) apply access rules so that only "Producer" users have access to this metadata. In this first example the plan is to leave EXIF metadata disclosed to anyone who consumes the image. Thus, the producer clicks "Next".

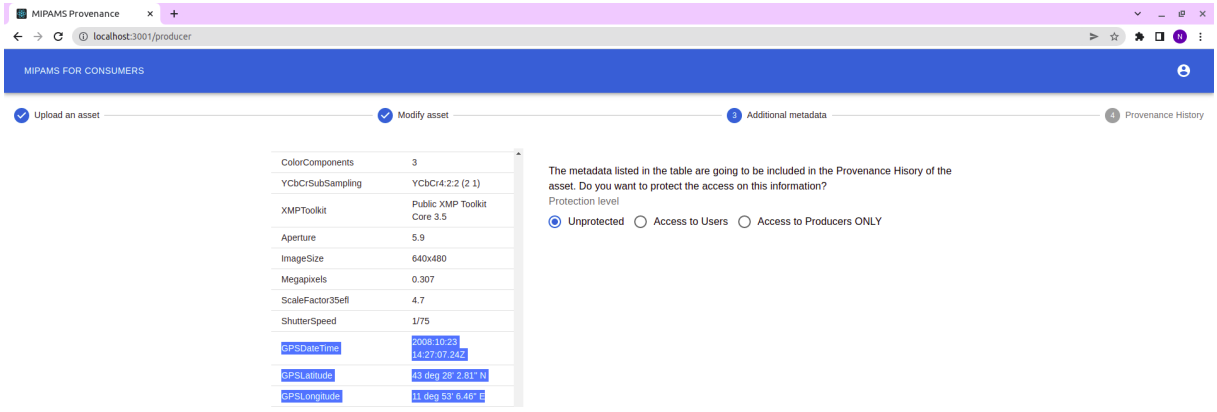


Figure 26: GPS information in EXIF metadata of the uploaded digital asset

Finally, the producer reviews the set of assertions that is going to be included in the new Manifest Content type JUMBF box (figure 27). Once the producer clicks the "Generate Provenance" Button, a REST call will be sent to the controller of this application in order to produce and embed Manifest Store Content type JUMBF box to the newly created digital asset. Basically, all this functionality is implemented in the mipams-fake-media library explained in sections 4 and 5.

Before calling these services it is important to extract all provenance information from the uploaded digital asset. Provided that there is already an Active Manifest, it needs to be validated as well. All these preparation steps along with the invocations to produce the new manifest store are implemented in the service called FakeMediaProducerService inside the mipams-fake-media-demonstrator software.

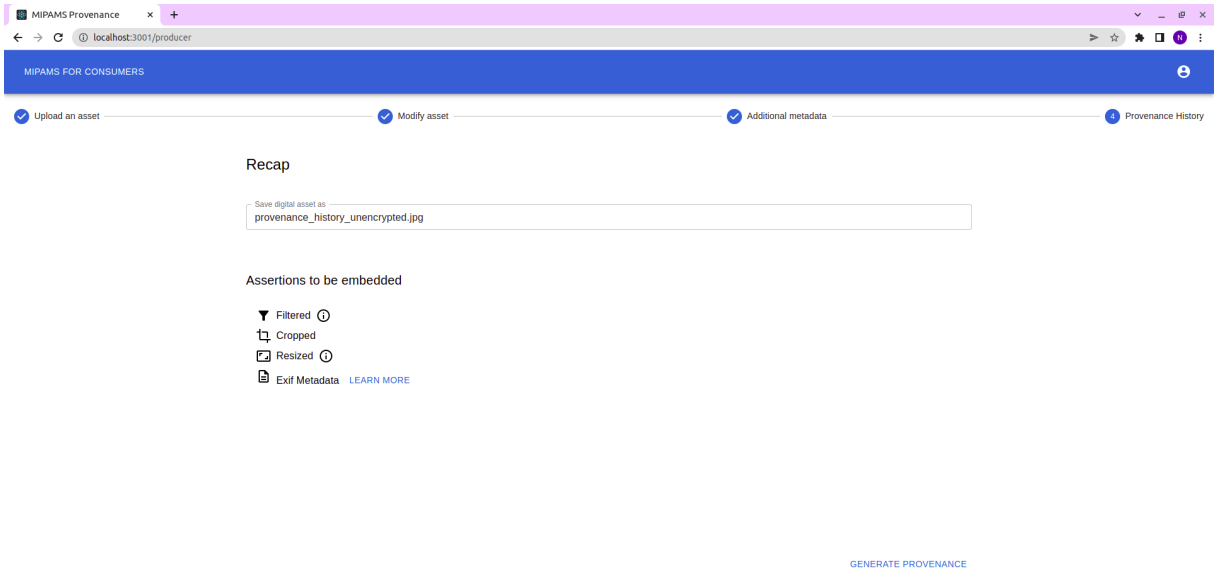


Figure 27: Producer reviews the assertions that shall be included and names the produced asset as "provenance_history_unencrypted.jpg"

The producer creates the provenance history of two additional images. In the first one, called "provenance_history_protected.jpg" the producer selects to allow only authenticated users to view the EXIF metadata of the asset (figure 28). In this case, the application doesn't define any access rule for accessing this EXIF metadata. Instead, this authorization check is implemented inside the software. However, in the final image - i.e. "provenance_history_protected_ar.jpg" - the producer selects to include access rules so that only fellow producers have access to the asset's EXIF metadata (figure 29). With these two additional images, it is possible to simulate two different ways of managing access rules: in the first case the whole access rule policy is implemented at a software level while in the second case the policies are appended to the desired digital asset and the authorization is achieved through an external authorization service.

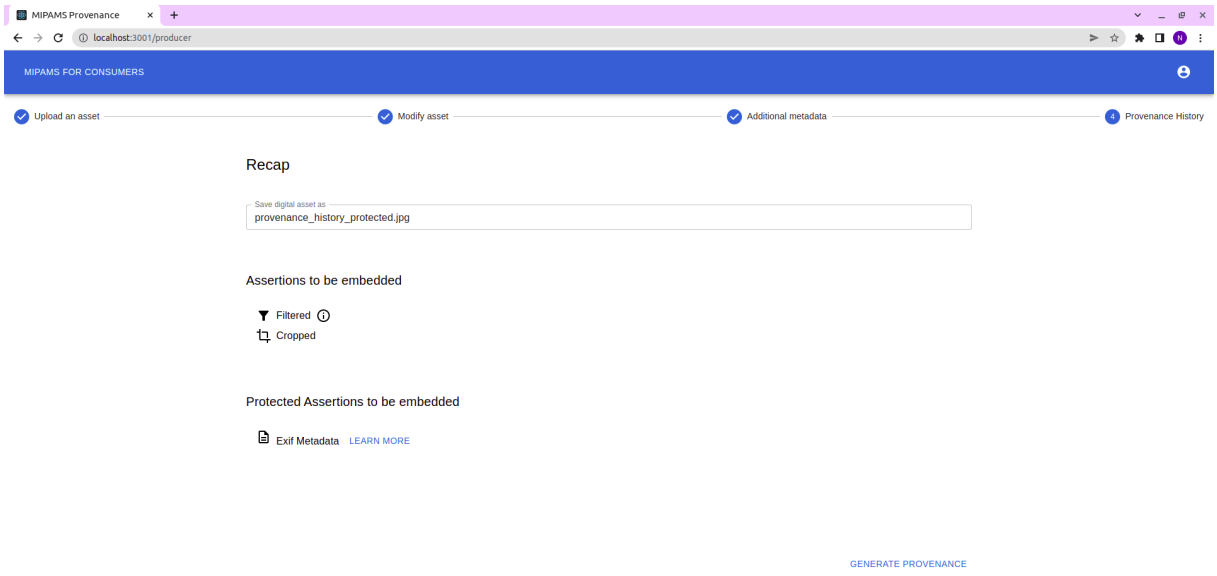


Figure 28: Producer reviews the assertions that shall be included and names the second produced asset as "provenance_history_protected.jpg"

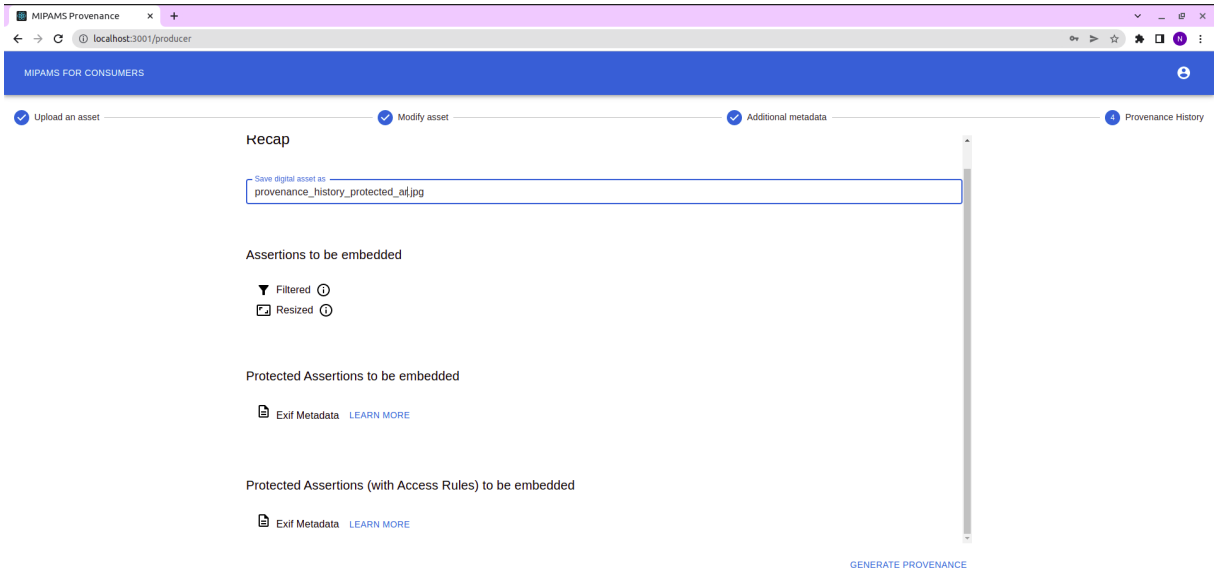


Figure 29: Producer reviews the assertions that shall be included and names the third produced asset as "provenance_history_protected_ar.jpg"

Finally, the producer can review the generated digital assets along with their embedded provenance information as shown in figure 30. Now "nickft" can start distributing her digital assets via social media or in her articles.

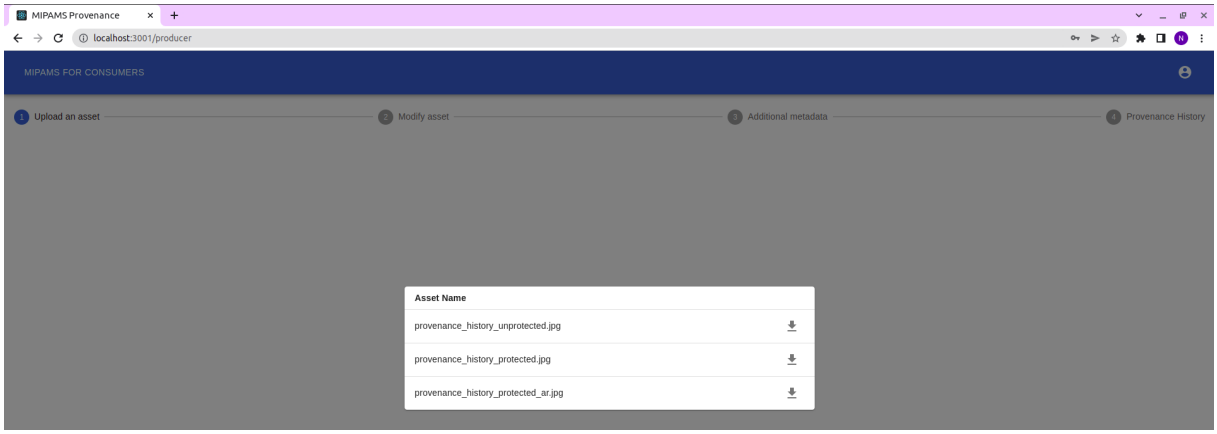


Figure 30: Producer can view and download the produced digital assets

For debugging purposes, a developer could inspect the structure of the generated manifests using the application presented in section 6.1. Specifically, a user could take the first produced digital asset (i.e. the one called "provenance_history_unencrypted.jpg") and upload it to the first application. As depicted in figure 31, the application parses a JUMBF Box structure embedded in the JPEG image. As seen in the descriptive message the first JUMBF Box that is encountered is a Manifest Store Content type JUMBF box. Apart from a description box (i.e. box with type jumd), it contains only one content box that has total size of 7994 bytes. It is another JUMBF box (i.e. type jumb) that corresponds to the single Manifest Content type JUMBF box that was just created by producer "nickft". Inside the Manifest a set of JUMBF Boxes is placed as its Content Boxes, namely a Claim, a Claim Signature and an Assertion Store Content type JUMBF box. Finally, the content of the Assertion Store is a set of five JUMBF Boxes. The first three JUMBF Boxes correspond to the three actions (i.e. Crop, Filter and Resize) that the producer applied via the Image Editor. The forth assertion is a JSON Content type JUMBF box and corresponds to the EXIF metadata that was included unencrypted. Finally, as depicted in the figure, the final assertion is a Content Binding assertion. It has been added automatically by the mipams-fake-media library and it is required in order to bind the manifest with the specific digital asset as explained in section 4.1.1.1.

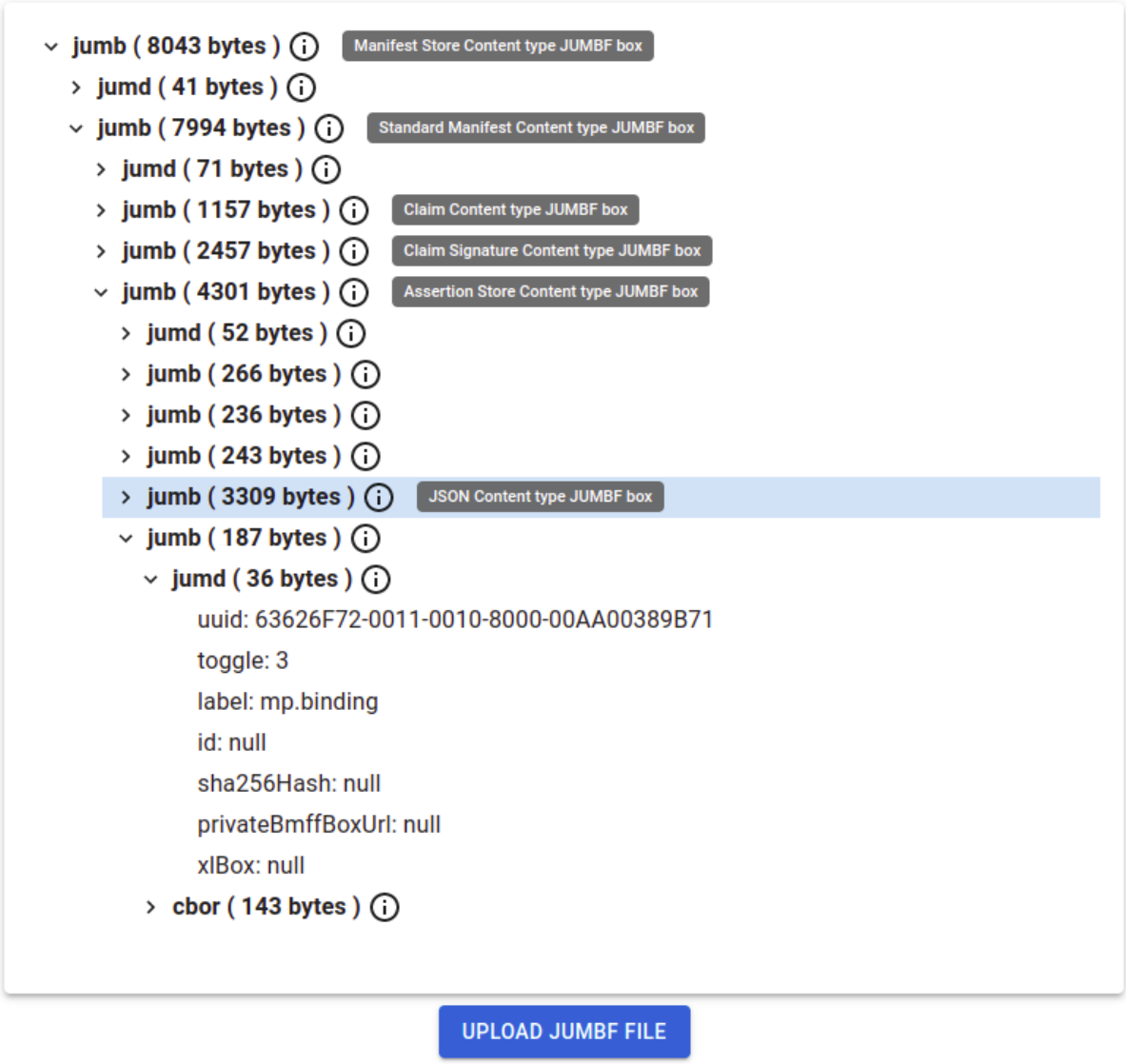


Figure 31: Manifest Store content type JUMBF box structure embedded in provenance_history_unencrypted.jpg

However, if a user would upload the remaining two images she would see a slightly different behaviour. Specifically, by uploading the second asset the user would still see a Manifest Store with only one Manifest Content type JUMBF box. This time the Assertion Store Content type JUMBF box contains the two Assertions corresponding to the two actions performed via the Image Editor, the Content Binding assertion (it is not visible in the figure; it is below the last visible assertion) and the protected EXIF metadata located inside a Protection Content type JUMBF box. As per section 3.2, a Protection JUMBF box contains a Description box, a Protection Description box (pspd) containing information related to the encryption process and a Binary Data box (bidb) containing the ciphertext (i.e. protected exif metadata). Notice that in the Protection Description box the arLabel field is set to null. This means that no JUMBF Box is referenced containing access rules

for this protected content. This encryption has been carried out using AES-256 Cipher Block Chaining (CBC) mode with a nonce. The secret (i.e. the symmetric encryption key) is generated and configured through an application variable defined in application.yml file.

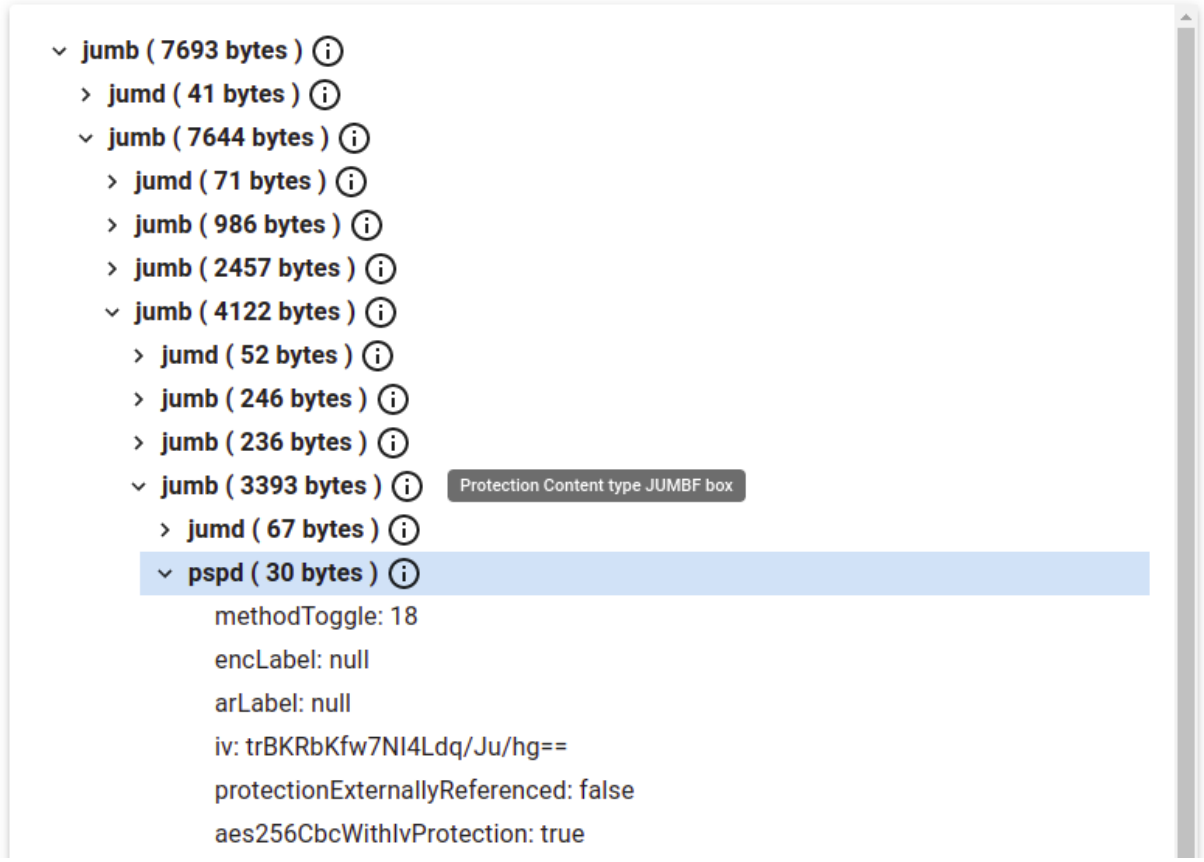


Figure 32: Manifest Store content type JUMBF box structure embedded in provenance_history_protected.jpg

The final inspection is related to the third digital asset produced by "nickft" user. In this asset the EXIF metadata have been protected with access rules as well. As shown in figure 33 inside the Assertion Content type JUMBF box there is the Protection Content type JUMBF box related to the EXIF metadata. In this case, the arLabel field in the Protection Description box is set to a string. This string corresponds to the label of the XML Content type JUMBF box which is the JUMBF Box containing the access rules that should be applied in case a user wants to access the protected content.

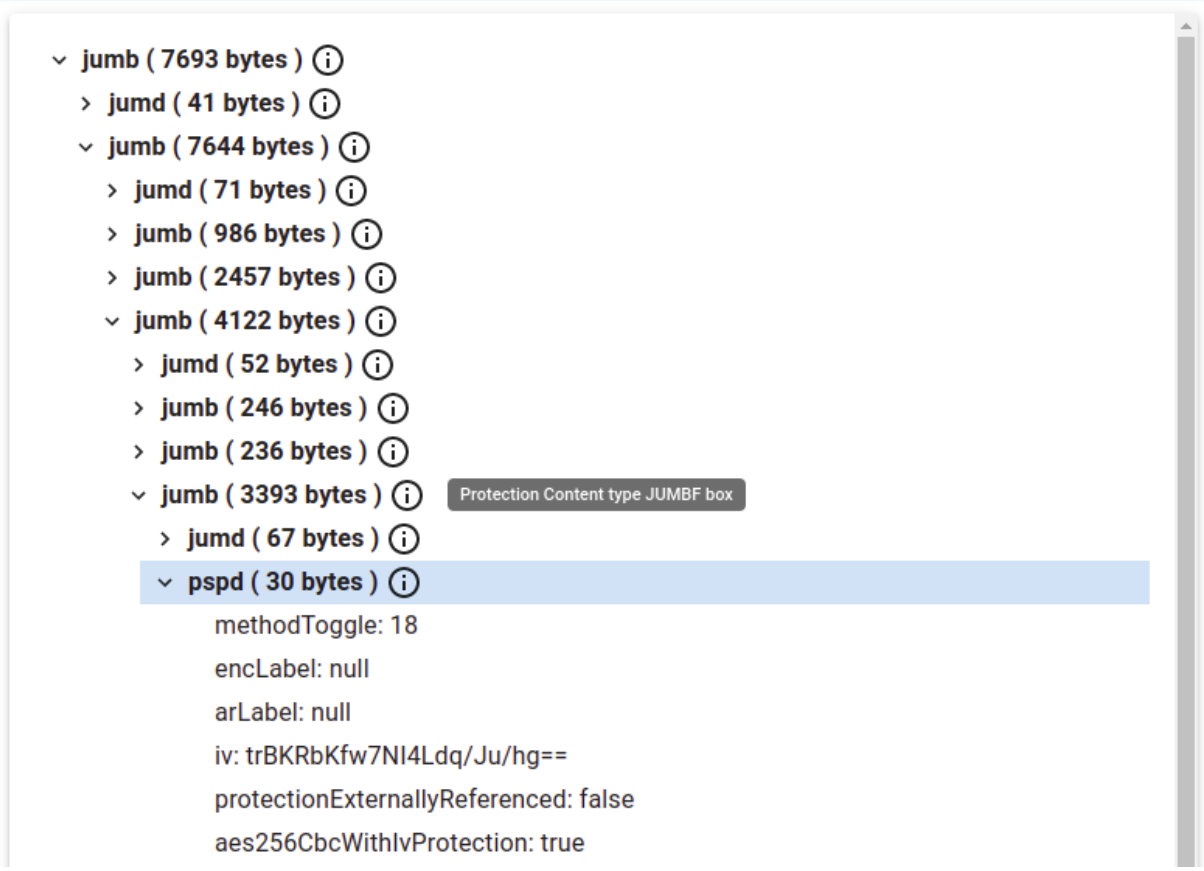


Figure 33: Manifest Store content type JUMBF box structure embedded in provenance_history_protected_ar.jpg

As shown in figure 34, upon parsing the embedded provenance history, the contents of each JUMBF Box are extracted and stored in separate files inside the container that runs the "Practical demo for JUMBF" application. Thus, it is possible to view the contents of the XML box that contains the access rules. In figure 35 a XACML policy is specified describing the rule under which a user can access the referenced exif metadata. Specifically, the user shall have access to the protected resource provided that she has the "PRODUCER" role.

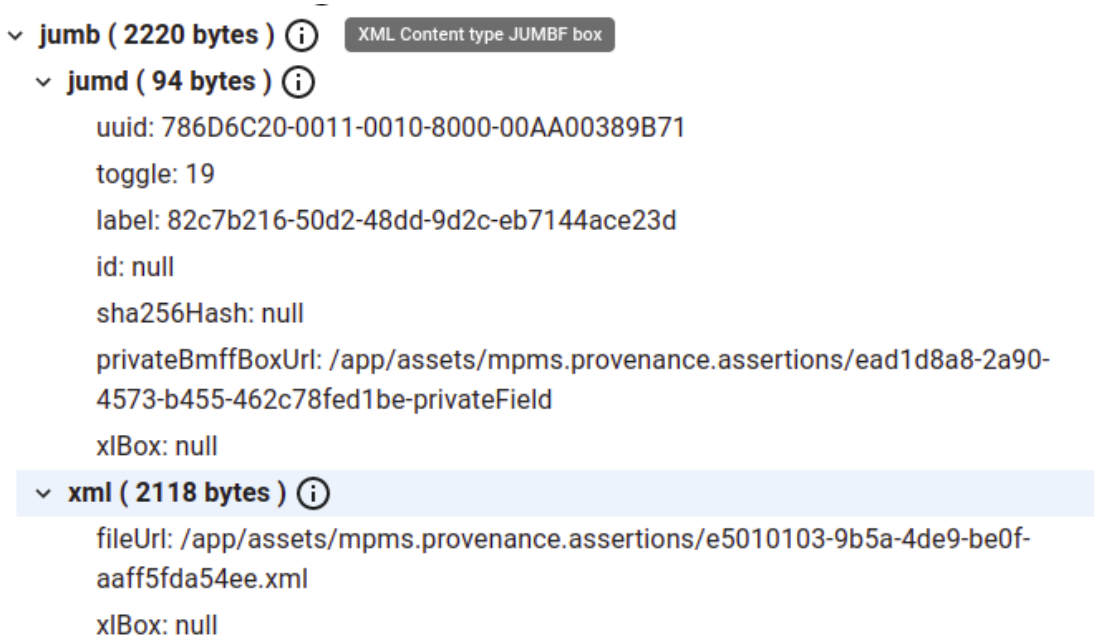


Figure 34: XML box contents are stored in a separate file

```
/app/assets $ cat mpms.provenance.assertions/e5010103-9b5a-4de9-be0f-aaff5fda54ee.xml
<?xml version="1.0" encoding="UTF-8"?>
<Policy xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17
            http://docs.oasis-open.org/xacml/3.0/xacml-core-v3-schema-wd-17.xsd"
        PolicyId="urn:isdcm:policyid:1"
        RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:first-applicable"
        Version="1.0">
    <Description> Policy template </Description>
    <Target/>
    <Rule RuleId="urn:oasis:names:tc:xacml:2.0:ejemplo:RuleMed" Effect="Permit">
        <Description> This is a template to generate a policy in MIPAMS Provenance </Description>
        <Target>
            <AnyOf>
                <AllOf>
                    <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
                        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">PRODUCER</AttributeValue>
                        <AttributeDesignator MustBePresent="false"
                            Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject"
                            AttributeId="urn:oasis:names:tc:xacml:3.0:example:attribute:role"
                            DataType="http://www.w3.org/2001/XMLSchema#string"/>
                    </Match>
                    <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
                        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">GET</AttributeValue>
                        <AttributeDesignator MustBePresent="true"
                            Category="urn:oasis:names:tc:xacml:3.0:attribute-category:action"
                            AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
                            DataType="http://www.w3.org/2001/XMLSchema#string"/>
                    </Match>

                </AllOf>
            </AnyOf>
        </Target>
    </Rule>
    <Rule RuleId="urn:oasis:names:tc:xacml:2.0:ejemplo:RuleDeny" Effect="Deny"/>
</Policy>/app/assets $
```

Figure 35: XACML policy describing that only users with Role "PRODUCER" can access the resource

6.2.4 Consuming provenance history

In this section focus is given on the different cases of consuming the provenance information of a digital asset. The consumption functionality could be an independent application, separate from the producer's functionality. For instance, it could be a browser extension that, upon the display of an image, it could also display an icon providing all its provenance information. Consequently, the first case that shall be examined is the provenance consumption by a user that is not authenticated with our application. In theory, any user can access, consume and validate the integrity of provenance information. Figure 36 shows the use case where an unauthenticated user consumes the first digital asset which discloses the EXIF metadata. At the right-side of the screen a short description of the Active Manifest's contents is displayed. Firstly, the consumer has access to the modifications that producer made. Next to some assertions, an "Info" icon contains the metadata that accompany each action. Regarding the EXIF metadata, by clicking "LEARN MORE" the consumer has access to the entire set of EXIF metadata (figure 37). Moreover, the date and the entity who signed this Claim are available too. Notice that entity's information (i.e. CNN) is extracted from the certificate Subject field. No information about the producer user is available.

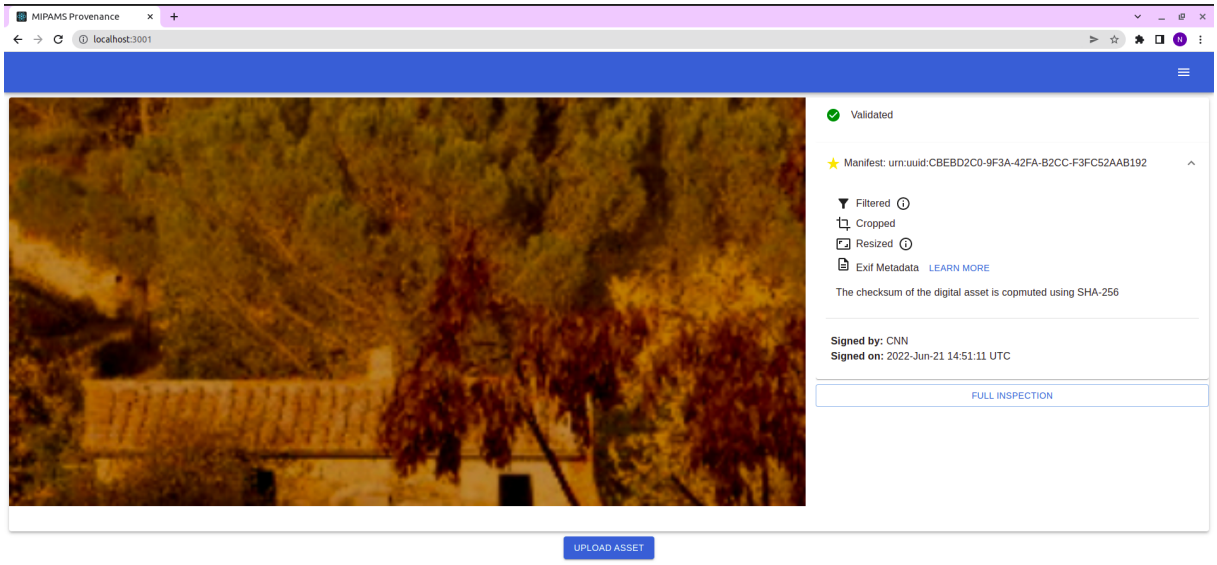


Figure 36: User consumes provenance information without being authenticated

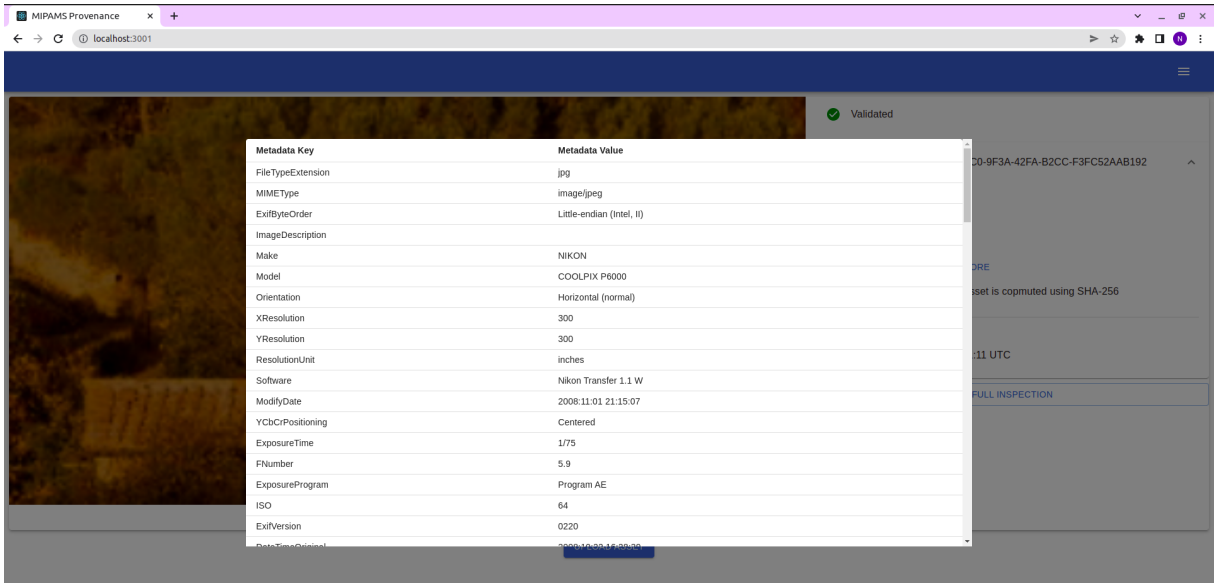


Figure 37: User has access to the EXIF metadata inside the Claim

It is worth mentioning that once a user wants to consume the provenance history of a digital asset, the application needs to extract all the JUMBF boxes that are contained in it. The Manifest Store shall be a unique JUMBF Box in the list of the extracted JUMBF Boxes. Then this can be fed to the mipams-fake-media library for inspection. The aforementioned preparation step is implemented in the mipams-fake-media-demonstrator project in a service called FakeMediaConsumerService.

Furthermore, in figure 36 right below the active manifest's information a button "Full Inspection" is depicted. As explained in section 4.3 there are two types of provenance inspection: the first one locates only the active manifest and validates its integrity, while the second one performs a full inspection on the entire provenance history of the asset. With this button, the user has the ability to check the entire history; however, it has no effect in our examples since each of the digital images contain a single manifest.

Advancing to the second example, the unauthenticated user tries to consume the provenance information of the second digital asset which contains the protected EXIF metadata. In figure 38 it is apparent that the unauthenticated user cannot fully consume the embedded provenance information. Although she can validate the integrity and validate the digital signature of the producer, she cannot access the EXIF metadata. Thus, the inspection status is set to "Partially validated".

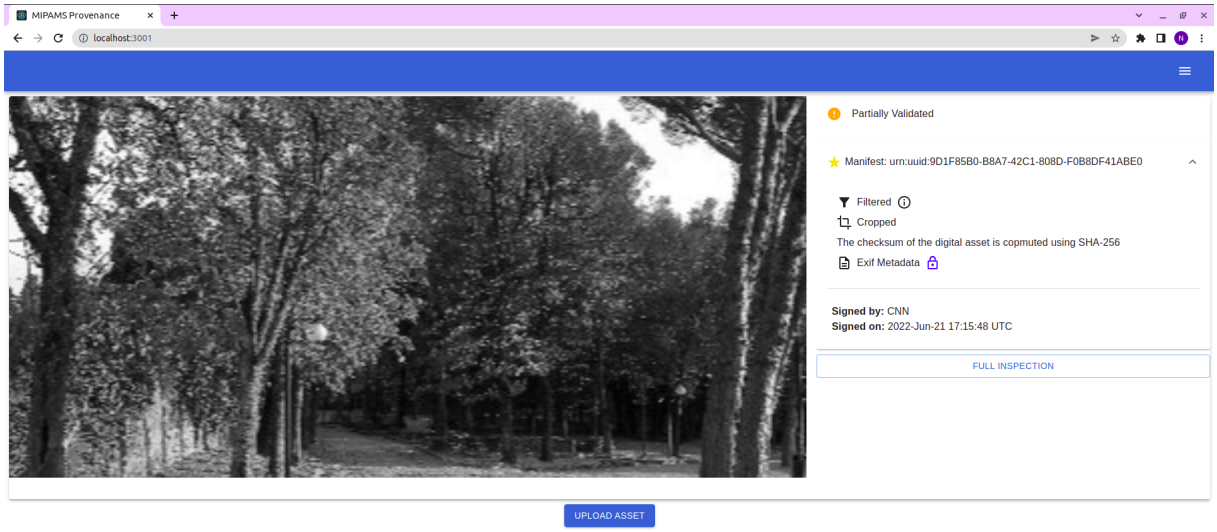


Figure 38: Unauthenticated user cannot access protected EXIF metadata

Now, if the simple consumer user "user" logs in to the application and tries to consume the provenance information of the same digital asset, she will successfully validate and access all the provenance information as shown in figure 39.

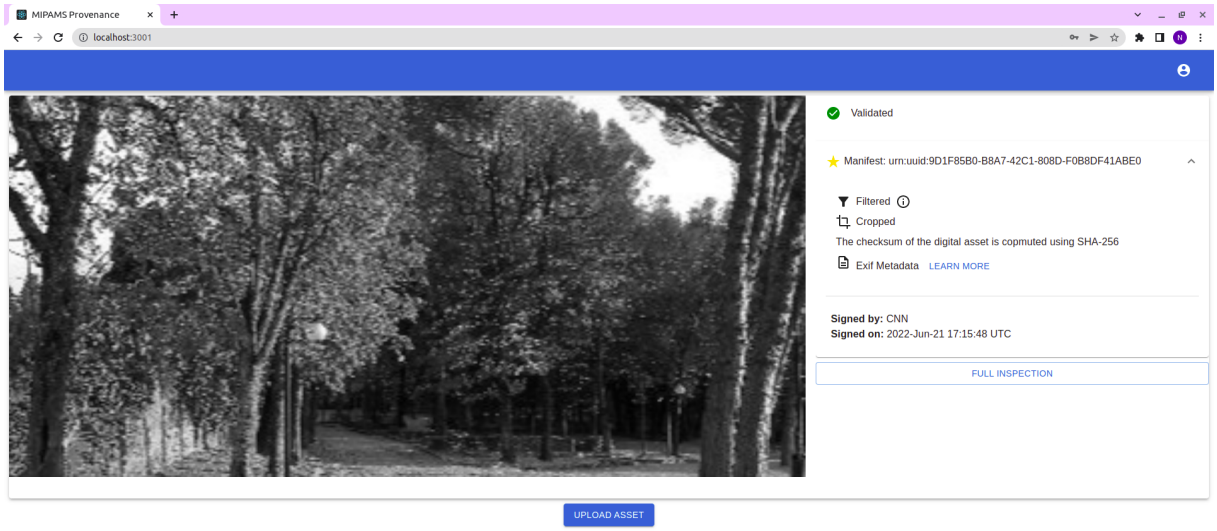


Figure 39: Authenticated consumer user access protected EXIF metadata successfully

Finally, if the simple consumer user tries to access the final provenance information produced by "nickft", she shall experience a similar "Partially validated" message since she does not have the proper access rules (i.e. she is not a Producer user). In the case where an assertion is protected based on a XACML policy, the application needs to decide whether the authenticated consumer user has the correct access rules. For this reason, the authorization service developed in scope of the work related to Genomic Information

Protection And Management System (GIPAMS) [33] is used. This authorization service implements a REST endpoint called "authorize_request" that returns a XACML response "Permit" or "Deny" depending on the XACML request and XACML policy provided. In XACML term, this authorization service plays the role of a Policy Decision Point (PDP). So, in case the FakeMediaConsumerService class encounters a Protection Box with access rules reference enabled, it extracts the XACML policy from the respective JUMBF Box, builds the XACML request based on the JWT claim "role" that contains the roles of the authenticated user and performs a Rest call to the authorization service.

7 MIPAMS Environment

The application presented in section 6.2 mainly illustrates the workflows of users who want to produce and consume provenance history for digital assets. Since this software is a proof of concept application explaining the provenance operations, it does not specify the way to manage the producers' digital certificates used for signing the claims nor does it provide an external service which creates the XACML policies and requests for the inclusion of privacy rules that protect the EXIF metadata of a digital asset. These aforementioned required services are already implemented in the Multimedia Information Protection And Management System (MIPAMS) [3]. This section focuses on defining the interconnections that need to be implemented in order for a Provenance application to operate inside the MIPAMS environment. The schema of the updated MIPAMS environment integrating the provenance application is depicted in figure 40. This schema describes the updated MIPAMS architecture supporting the provenance application.

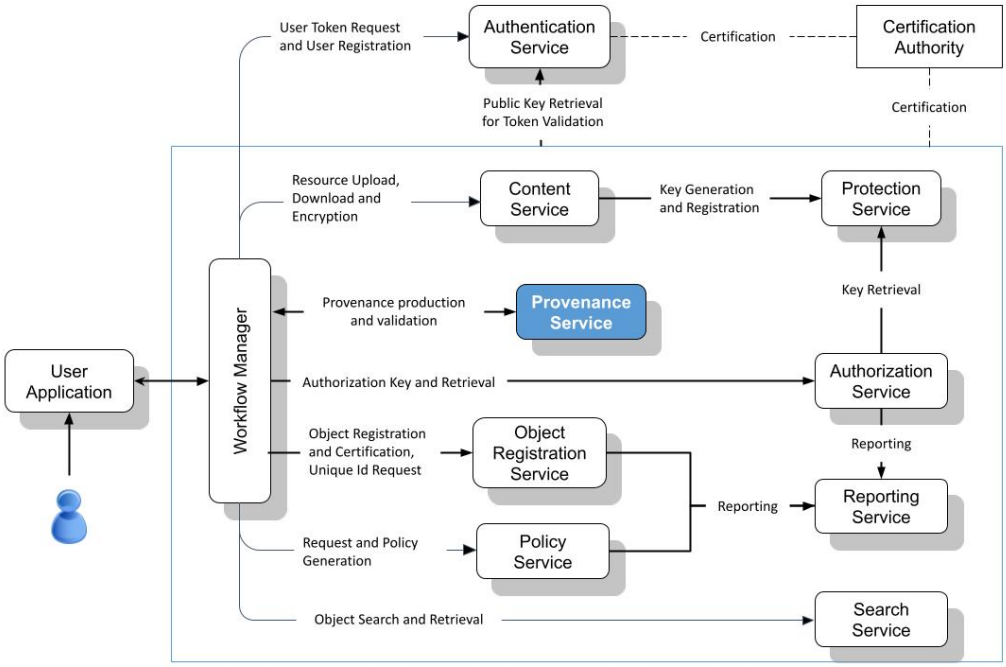


Figure 40: MIPAMS architecture enhanced with Provenance Service

First and foremost, to generate a secure environment it is of paramount importance to ensure the proper management of encryption and signing keys. The creation and storage of users credentials (e.g. keys, certificates) is handled by the MIPAMS Protection Service, a module that only authenticated and authorized services should have access. In addition, all the cryptographic operations described so far need to be implemented in a specific service that is authorized to access the specific producer private key and certificate. Moreover, this service should properly handle as well as destroy these credentials to avoid any leakage of cryptographic material that would compromise the entire security of the solution. In MIPAMS Environment this module is called Content service. In the mipams-fake-media

library all these cryptographic operations are performed by the jumbf-crypto-1.0 library. This "outsourcing" of such critical operations was decided specifically to demonstrate the ability of the mipams-fake-media library to support this communication with a protected service such as the Content Service.

Subsequently, regarding the generation and validation of privacy rules, MIPAMS Environment already provides a set of services that can be used by the provenance manager application. Specifically, there is a Policy Service (PoS) that is responsible for creating a XACML policy according to a set of provided parameters. In parallel, this service is responsible for creating a XACML request according to the authenticated user information. With these two pieces of information the Provenance application manager could validate whether a consumer user has the correct access rules by communicating with the already implemented MIPAMS Authorization Service. Similar to the cryptographic operations, the creation of XACML policies as well as the preparation for communicating with the Authorization Service for a XACML request is outsourced and implemented inside the jumbf-crypto-1.0 library, outside of the provenance manager application. The purpose behind this decision is, again, based on the preparation of provenance manager to be integrated with the MIPAMS Environment in a future release.

Figure 41 shows the sequence diagram inside the MIPAMS Environment for creating provenance information about an asset, applying access rules and protecting the content of sensitive metadata. The steps are briefly described as follows:

1. Producer uploads an image to the Provenance Manager application (e.g. Provenance Service) which is registered in the application.
2. A new identifier shall be issued through the Object Registration Service (ORS) for the newly registered image.
3. The producer performs a set of modification using the text editor. Once she has finished, the modified image is uploaded to the application.
4. During step 3, all actions are being recorded and converted into provenance assertions.
5. She selects a piece of metadata to be protected (e.g. EXIF metadata)
6. The Assertion Store is constructed including a content binding assertion and without assertion that needs to be protected. The Claim Content type JUMBF box is also partially constructed.
7. The assertion that needs to be protected is sent to the Content Service for encryption.
8. Content Service queries the Protection Service in order to get the producer's symmetric key used for protecting assertions.
9. The protection box is created and sent to the Provenance Service
10. The privacy policy is constructed via the Policy Service (PoS) and a new JUMBF Box is generated containing the proper XACML policy. The label of this JUMBF

Box is stored in the Protection Content type JUMBF box and specifically in its Protection Description box. The Assertion Store and the Claim are updated properly to contain the newly created JUMBF Boxes.

11. The Claim is sent to the Content Service in order to be digitally signed by the producer's credentials.
12. Content Service queries Protection Service in order to retrieve the producer's private key used for signing claims.
13. The digital signature is retrieved and the Claim Signature is constructed. Now the new Manifest Content type JUMBF box can be assembled.
14. The new Manifest is appended to the Manifest Store which can be now embedded in the image (i.e. digital asset).
15. The newly created image (along with its provenance history) is registered using the Object Registration Service.
16. Eventually the updated digital asset is returned to the producer who can now distribute it.

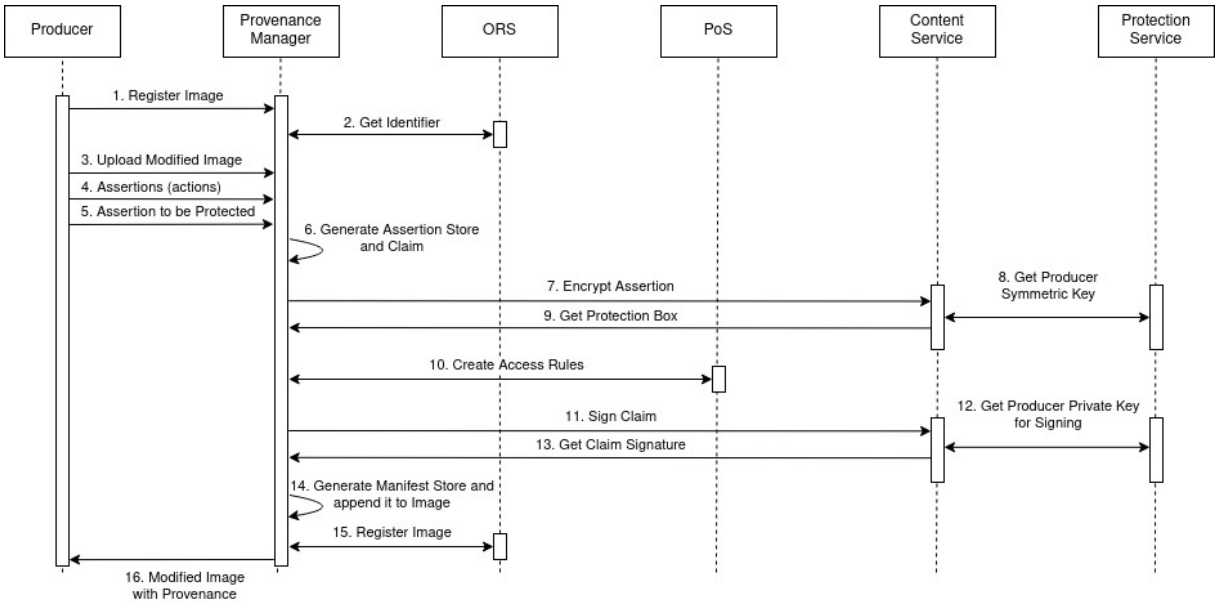


Figure 41: Including protected EXIF metadata in MIPAMS Environment

Once this digital asset is distributed through the Internet, it is ensured that the sensitive metadata is protected against unauthorized access while allowing the users to consume the rest of the provenance information of that particular digital asset. Figure 42 describes the steps that are followed inside the MIPAMS Environment, in order for a consumer to view the provenance information.

1. Consumer uploads a digital asset to the Provenance Manager application. She is interested in inspecting the provenance information of the uploaded digital asset.

2. Provenance information - and specifically the Manifest Store Content type JUMBF box - is extracted from the digital asset. The active Manifest Content type JUMBF box is located too.
3. The Claim and the Claim Signature are sent to the Content Service in order to validate the digital signature. Specifically, inside the Content Service the digest of the Claim Content JUMBF box is calculated.
4. In parallel, Content Service extracts the public key from the certificate inside the Claim Signature Content type JUMBF box. The module applied the public key to the digital signature and compare the output with the calculated digest. The result of this verification is then returned to the application.
5. Subsequently, the application verifies the integrity of the assertions inside the Assertion Store Content type JUMBF box with the URI references located inside the Claim.
6. Next, the application locates the Protection Content type JUMBF box corresponding to the protected metadata. Inside the Protection Description box the access rules label is referencing a JUMBF Box that is also located in the Assertion Store. This access rules JUMBF Box is located as well.
7. Based on the consumer information, the application communicates with the Policy Service module in order to build the XACML request.
8. The XACML policy is extracted from the referenced JUMBF Box. A XACML request is created from the Policy Service.
9. The Authorization Service answers whether the user is authorized to access this protected metadata.
10. Provided that the user is authorized, it sends the Protection Content type JUMBF box to the Content Service in order to be decrypted.
11. Content Service requests from Protection Service the symmetric key that the producer used to encrypt the protected metadata.
12. Content Service module sends the decrypted content to the application.
13. Eventually, the application displays the digital asset along with the provenance information to the consumer.

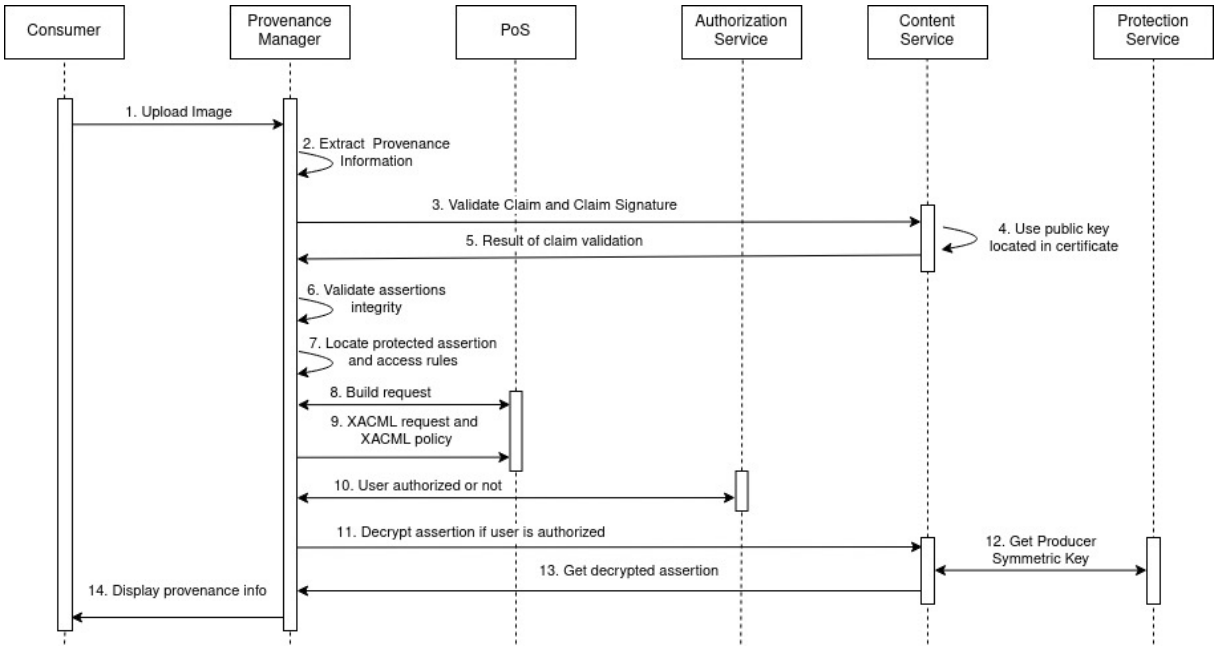


Figure 42: Consuming protected EXIF metadata in MIPAMS Environment

8 Future Work

This section focuses on defining some key future steps that could be followed based on the work that has been produced in scope of this project. These future steps are related to how the product manager application could be extended to fully showcase the functionalities of the mipams-fake-media library. In addition, this chapter discusses the possibility of integrating on-going work of the Distributed Multimedia Applications Group (DMAG) [34], part of the Information Modeling and Processing Research Group (IMP) [35] of the UPC. On top of that, it is also worth investigating how the specification could be more resilient to disassociation of digital assets with their provenance information. Finally, a possible extension of the application is examined where provenance information is stored and processed outside of the digital asset.

The following subsections provide some technical directions related to a set of future work projects related to this thesis work. Specifically, in section 8.1 a set of improvements regarding the existing demonstrator application is provided. Secondly, section 8.2 introduces the challenge of maintaining the association of a digital asset with its provenance information and provides some possible solutions. Subsequently, in 8.3 focus is given on the support of additional techniques for providing access control to the privacy-related assertions of provenance claims. Finally, 8.4 explores - discovers the challenges of - a different architecture than the one presented in the demo application where the provenance information is not embedded inside the digital asset.

8.1 Enriching provenance manager application

In section 6.2 the provenance manager application illustrated most of the main functionalities introduced in the mipams-fake-media library that implemented the provenance specification proposed in section 4. However, as recent future work, it is imperative that the provenance manager application demonstrates the entirety of the mipams-fake-media functionalities. In other words, there are still functionalities supported in mipams-fake-media library that are not possible to be performed via the demonstrator application.

First and foremost, the demonstrator application should provide a graphical interface to the producer in order to redact existing metadata. For instance, consider the scenario where a producer creates some provenance metadata for a particular digital asset. This provenance metadata contains GPS information unprotected. At a later point in time, the producer decides that she prefers to hide (i.e. redact) this information. A Redaction service has already been implemented in mipams-fake-media software, though, a user interface needs to be implemented in the mipams-fake-media-demonstrator application in order for a producer to have access to this functionality.

Secondly, with this demonstrator application only a subset of the assertions defined in table 5 is available to be stated in a claim. Specifically, a producer has the ability to perform only a subset of action assertions (i.e. filter, crop, resize) even though there are many more on table 6. Moreover, thumbnail assertions need to be supported as well. Thumbnail assertions could be included either manually by the producer or automatically by the demonstrator application.

Next, according to the specification, a user might produce a digital asset as the combination of multiple component pictures. These ingredient pictures might have their own provenance history so the producer might select to include their Manifest Content type JUMBF boxes inside the final digital asset's manifest store. This scenario is addressed during the design of the provenance operations in section 4.3. What needs to be implemented is the integration of this functionality with the demonstrator application. The producer workflow needs to be enriched by allowing the producer to add ingredient pictures when modifying a digital asset.

Regarding the specification itself, it is imperative that subsequent versions cover more metadata structures. These additional families of metadata are proposed in scope of the technical specification developed by C2PA. As of the first version of MIPAMS Fake Media specification, only EXIF metadata assertion is supported. In future versions EXIF metadata assertion shall be stored according to the JSON-LD schema defined by XMP [20] in order for wider adoption of the specification. In addition, other standardized metadata formats need to be included in order to cover more use cases. For instance, The International Press Telecommunications Council (IPTC) defines a standard set of descriptive, administrative and rights metadata typically used by photographers, distributors, news organizations, archivists, and developers. In fact, they have also defined their own standardized Information Interchange Model (IPTC-IIM) [36], a binary format which allows to describe this kind of metadata. The IPTC Photo Metadata assertion can be used to ensure that IPTC Photo Metadata - for example describing ownership, rights and descriptive metadata about an image - is added to the asset in a way that can be validated cryptographically. Another assertion type that could be integrated in the specification is focusing on "fact checking" entities that review the assertions made about a digital asset and provide a statement evaluating their truthfulness. This new assertion type could be named after the name of the respective schema "Claim Review" [37]. ClaimReview is a source label attached to website content, or images that provides search engines with some information about the referenced content. This can include a tag stating that asserted statements are fact-checked by verified sources and organizations with the aim of providing accurate information to users.

8.2 The Content Binding problem

When distributing a digital asset through the Internet it is likely that Internet Service Providers and/or end-user applications perform compression techniques to the digital asset or change its quality in order to increase both their performance and the quality of experience (QoE). The industry as well as research and standardization organisations have shifted their interest towards dynamic content delivery. Specifically, adaptive bitrate streaming techniques - like the Dynamic Adaptive Streaming over HTTP (DASH) standardized by MPEG [38]- consist of expressing a digital asset in various qualities, split in multiple chunks. The most suitable quality chunk shall be sent to the end-user according to the network and end-user's device conditions.

Unfortunately, this situation poses plenty of challenges in the field of associating provenance information to a specific digital asset. Both in our proposed specification as well as

in the case of C2PA technical specification, the single supported content binding method consists of computing the hash (i.e. SHA-256) of the digital asset and including the digest as a separate assertion inside the assertion store. However, when an entity stores a digital asset in various qualities, it creates versions of the digital asset with completely different hashes. Thus, the proposed specification is vulnerable to disassociation of provenance metadata. This is the main reason that this kind of content binding is considered as "hard binding". Consequently, it is of paramount importance to investigate different techniques for binding a digital asset with its provenance history.

There is a number of directions that can be followed. First of all, there is some research activity related to the design and implementation of perceptual hash functions that generate a watermark and embed it inside the host digital asset during compression or rendition procedures [39]. According to C2PA these techniques are considered as soft binding algorithms and are more resilient regarding the rendition of digital assets. In fact, C2PA investigates and evaluates the inclusion of global unique identifiers like the International Standard Content Code (ISCC) [40]. The ISCC Code is a unique, hierarchically structured, composite identifier. It is built from a generic and balanced mix of content-derived, locality-sensitive and similarity-preserving hashes generated from metadata and content.

8.3 Extension of Privacy policy implementation

The implementation of the MIPAMS Fake Media specification is agnostic to the privacy policy management. The application that uses the mipams-fake-media library is responsible for handling the privacy policies related to one ore more protected assertions. To that extent, the provenance manager application presented in section 6.2 extracts the JUMBF Box referenced by the "arLabel" field which is located inside the Protection Content type JUMBF box. This JUMBF Box is then sent to the crypto library for further processing. Up to this point, the Content Type of the JUMBF Box is not taken into consideration meaning that the access rules could be stored in any of the supported Content Type JUMBF Boxes. This creates the context to support multiple access control solutions that handle privacy policies differently. In MIPAMS Environment the implemented Policy Service (PoS) expresses policies and authorizes requests using XACML where all the rules are written in XML. Thus, privacy policies are stored in XML Content type JUMBF boxes.

However, in parallel to the progress of this project, there is a work at the DMAG group that focuses on designing a simplified access control language based on XACML using JSON. Thus, the XACML policy rule presented in figure 35 could be also expressed using this new JSON-based access control language. An example of expressing the MIPAMS Provenance policy in JSON is presented in figure 43.

Consequently, a future step concerning the provenance manager application would be to integrate this additional way of expressing and enforcing privacy rules. To achieve that, a developer would need to update the crypto library and specifically the methods that generate the privacy policy and execute the authorization query. Both methods are located in the CryptoService class.

```
{   "Policy":{
        "policyId": "1",
        "ruleCombiningAlg": "first-applicable",
        "version": "1.0",
        "desc": "Policy template",
        "rule":[
        {
            "ruleId":"urn:oasis:names:tc:xacml:2.0:ejemplo:RuleMed",
            "effect":"Permit",
            "desc": "This is a template to generate a policy in MIPAMS Provenance",
            "target":[{
                "AnyOf":[{
                    "AllOf":[{
                        "match": {
                            "matchId" : "urn:oasis:names:tc:xacml:1.0:function:string-equal",
                            "attributeValue":{
                                "value":"PRODUCER",
                                "dataType":"string"
                            },
                            "attributeDesignator":{
                                "mustBePresent": false,
                                "category": "subject",
                                "attributeId": "subject-id",
                                "dataType": "string"
                            }
                        }
                    },
                    {
                        "match": {
                            "matchId" : "urn:oasis:names:tc:xacml:1.0:function:string-equal",
                            "attributeValue":{
                                "value":"GET",
                                "dataType":"string"
                            },
                            "attributeDesignator":{
                                "mustBePresent": true,
                                "category": "action",
                                "attributeId": "action-id",
                                "dataType": "string"
                            }
                        }
                    }]
                }]
            }]
        },
        {
            "ruleId":"urn:oasis:names:tc:xacml:2.0:ejemplo:RuleDeny",
            "effect":"Deny"
        }]
    }
}
```

Figure 43: Policy expressed using JSON-based access control language

8.4 Storing provenance information on the cloud

Among the JPEG Fake Media requirements that the proposed specification is addressing, is the ability of the provenance structure to be self-contained in the sense that it can also live outside of its digital asset. Thus, in some cases where the size of the Manifest Store is comparable or even larger than the digital asset itself, it could be advisable to store the provenance information outside of the asset. The database that stores the provenance information of digital assets is called Manifest Repository. This section presents a complete

provenance solution that could be built using the services presented in MIPAMS Environment while supporting a Manifest Repository for storing and requesting provenance information.

First and foremost, a new database service will be required that shall be responsible for storing the Manifest Repository table. This Manifest Repository table shall contain entries that correspond to the generated Manifest structures. Specifically, each entry shall have as primary key the label of the Manifest Content type JUMBF box which is a unique string issued for each particular Manifest (e.g. "urn:uuid:7f4d23..."). An additional value should contain the entire structure of the stored manifest in JSON format. The proposed reference software jumbf-core-2.0 supports the expression of a JUMBF structure in JSON. However, it is worth mentioning that, jumbf-core-2.0, the JSON representation of a JUMBF box contains links to a set of files pointing to its contents. For instance, to avoid loading all the contents of a XML file to the memory, they are stored in a file with ".xml" extension and the XML Content type JUMBF Box contains the location of that particular file. That being said, it is important that when an application requests a specific Manifest Content type JUMBF box from the Manifest Repository via an HTTP request, the complete JUMBF bytestream should be sent to the application (i.e. and not the JSON representation with the file references).

Secondly, since the Manifests are stored outside of the digital assets that they describe, it is imperative that the proposed solution keeps track of at least the active manifest of each digital asset. Thus, a new database table should be created containing the correlation between a digital asset and its active manifest. Only by knowing the active manifest, it is possible to navigate through the entire provenance chain of the digital asset by following the ingredient manifest references. As presented in section 7, upon upload of a new modified digital asset a new identification number is issued via the Object Registration Service (ORS). This number called instanceID shall be the primary key that unambiguously references the digital asset. Thus, this new database table, called Asset Repository, shall consist of entries whose primary key is the instanceID field. The values of each entry shall be the hash of the referenced digital asset as well as the id that corresponds to asset's active manifest. This active manifest id shall point to an existing Manifest Repository entry. The relationship between the two database tables is depicted in figure 44.

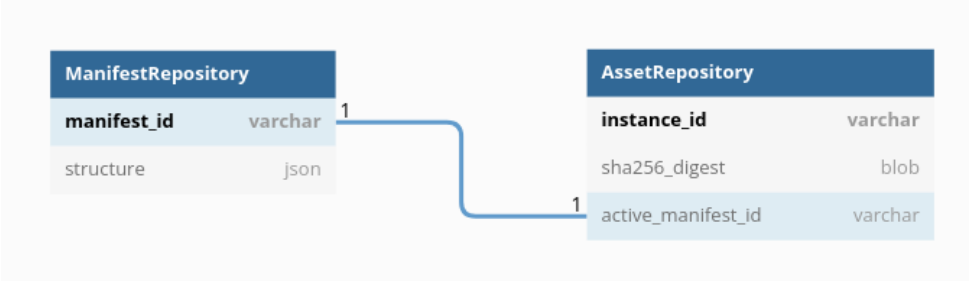


Figure 44: Database schema presenting Manifest and Asset Repository tables

Since the active manifest is not embedded inside the digital asset, there should be a reference string embedded in the image that an application could use in order to request

the manifest from the Manifest Repository. One solution would be to embed an identifier using the Extensible Metadata Platform (XMP) serialization format. XMP [20] - initially designed by Adobe but then adopted as an ISO standard - standardizes a data model, a serialization format and core properties for the definition and processing of extensible metadata. Among other formats, it also provides guidelines for embedding XMP information into JPEG images.

As proposed by C2PA's technical specification, our provenance solution could use XMP to store the active manifest id using a custom XMP metadata field (e.g. XMP field "dcterms:provenance"). That way, it would be straightforward to access the active manifest of a digital asset by simply requesting it from the Manifest Repository using its manifest id. Another approach would be to store the generated instance id number that uniquely characterises the digital asset in the Asset Repository. Specifically, the XMP:instanceID metadata field could be used in order to embed this unique id to the digital asset. Consequently, an application, that wants to access the active manifest of a digital asset, has to request it from the Asset Repository. Finally, provided that no XMP metadata is embedded in the digital asset, our provenance solution could try to recover the instance id of the digital asset through its hash. As described in figure 44 apart from the instance id, the SHA-256 digest of a digital asset is stored in the Asset Repository. Thus, the instance id of the digital asset could be retrieved even if there is no information embedded in the digital asset. This is crucial as it makes our solution resilient to disassociation attacks where an adversary wants to strip any metadata and break the reference between the digital asset and its provenance history.

With the Asset Repository our proposed provenance application would keep tracks of the association between digital assets and their active manifests. This could provide the means to address the content binding problem described in section 8.2 without the need of new binding techniques (i.e. soft binding algorithms). Specifically, when an application uploads and registers a new digital asset through the MIPAMS Provenance Service, a set of renditions could be automatically computed. This means that each of the digital asset renditions would have its own instanceID in the Asset Repository. In order to track this new modification (i.e. the rendition of a digital asset in a specific quality) a new Manifest should be included by the MIPAMS Provenance ecosystem. That way, our ecosystem wouldn't rely on ISPs and end-user applications to update the provenance history of the distributed digital asset. However, this approach wouldn't be transparent in the sense that a new Manifest should be added by MIPAMS Provenance Service on top of the producer's Manifest. As an alternative approach would be to issue a new ID for a digital asset called documentID following the approach of XMP. This documentID - as opposed to the instanceID - would be a unique identifier that describes all the renditions and different expressions of a specific digital asset. That way, the association between the digital asset and its active manifest in the Asset Repository would be through the documentID field.

Up until now, it is assumed that digital certificates included in the Claim Signatures were valid. However, MIPAMS architecture should provide the corresponding service that, given a configurable set of trusted certification authorities, it responds to requests related to the validity of producer's digital certificates. In Public Key Infrastructure (PKI) the status of a X.509 certificate is checked using the Online Certificate Status Protocol (OCSP) [41].

The service that is responsible for replying in such requests is called OCSP responder and there are multiple open source implementations (for instance an OCSP responder could be launched using openssl tool) that would allow the inclusion of such a service in our ecosystem.

To conclude, a widened MIPAMS architecture could be developed in order to provide complete provenance functionalities. On top of that, by storing the provenance information in a database, it allows for additional analytics and query capabilities on the provenance history of digital assets. Furthermore, this design would support various producer and consumer applications to be developed independently in different projects depending on the use case. For instance, a provenance consumer application could be developed to run as a standalone application, a browser extension or an additional service for a social media application. As long as the developed application is aware of the end points in order to query the OCSP Responder and the Manifest and Asset Repositories, it can be able to perform all the provenance operations successfully. Cryptographic material (e.g. private keys) is not accessible to the developed applications as all the operations are taking place internally, inside the MIPAMS Environment. Finally, as a way to ensure proper use of the MIPAMS environment from the developed applications, the access to the MIPAMS services should be accessible by authorized applications only.

9 Conclusions

The surge in the number of fake news flooding the Internet along with the development of sophisticated techniques that modify the content of a digital media have created the urgent need for countermeasures. At the same time, social media users lack of proper education regarding the security and privacy risks lurked when distributing an image of their personal moments. Both these problems could be addressed by annotating the provenance history of the modifications that a specific digital asset has been subjected to while allowing the redaction or protection of privacy-related information. This annotations is required to be securely stored in order to ensure that the expressed statements have not been tampered with after the time of generation. Consequently, this project focuses on defining a specification that expresses the data model of provenance information as well as the operations that need to be followed in order to ensure that this structure cannot be modified in the future. This specification is based on the technical specification issued by the Coalition for Content Provenance and Authenticity (C2PA) and extends its functionality in order to support the protection (i.e. encryption) of privacy-related metadata. Since it should be possible to embed provenance metadata inside the corresponding digital asset, the defined structure shall be based on the JPEG Universal Metadata Box Format (JUMBF).

In scope of the proposed specification, a set of software libraries have been developed. First of all, since the data model is expressed in JUMB format, a project called mipams-jumbf has been implemented which focuses on developing the data model as well as the means to parse and generate such metadata to a JPEG image. In addition, a library called mipams-fake-media is implemented using the functionalities defined in mipams-jumbf project. The goal of the new library is to define the provenance metadata structure and provide all the tools in order to generate and verify the validity of such metadata. All the libraries are written in Java.

Finally, in order to demonstrate the applicability of securely annotating provenance history to digital assets, an application has been developed simulating the scenario where journalists annotate the modifications that they perform on a digital asset before distributing it. This application showcases the ability of a journalist to protect a subset of metadata that could disclose privacy-related information like the location of the captured image.

The output of this work is related to the JPEG Systems multi-part specification [8]. One of the core parts of JPEG Systems specification is the JUMBF standard (ISO/IEC 19566-5 [11]). The rest of the JPEG Systems parts defining their own JUMBF Boxes addressing their specific application. Although there are plenty of JPEG System standards using JUMBF notation, there is not yet a reference software that showcases the applicability of the JUMBF data model.

To that extent, part of mipams-jumbf project has been submitted as proposed reference software to the JPEG Systems committee. Specifically, an input document [17] has been submitted describing the jumbf-core-2.0 library that implements the JUMBF data model standardized in [11]. In addition, the submission of a second document [18] focuses on presenting a reference software for the JUMBF Boxes proposed in scope of ISO/IEC

19566-4 standard [12]. The submission of the two aforementioned input documents has led the JPEG Systems Committee to issue a new Part - Part 10 - focusing on the development of a reference software for the various parts of JPEG Systems.

During the development of the contribution related to the JUMBF standard, the JPEG Systems committee had been working on finalizing the second edition of the ISO/IEC 19566-5. During this activity, a set of modifications, both editorial and technical, where proposed from our side to the committee. Specifically, these modifications were focusing on the definitions of the Content Boxes that were supposed to be included in the second edition of the standard. After discussion with the group, the proposed modifications were accepted and included in the new edition of the standard which is currently in Draft International Standard (DIS) under ballot status.

Apart from the contributions concerning the ISO/IEC 19566-5 standard, it has been decided to participate in the JPEG "Call for Proposals" procedure which searches for contributors to a JPEG standard tackling the Fake Media problem. Our proposed specification addresses a subset of the requirements proposed by JPEG Fake Media call for proposals. Thus, the plan is to contribute the standardization process by presenting a specification - based on top of the C2PA specification - that focuses on JPEG digital media and aims to support the encryption of privacy-related metadata to include those use-cases where disclosure of private information is not tolerated. The mipams-fake-media library shall accompany our submission in order to showcase the applicability of our proposal.

References

[1] Luisa Verdoliva. Media forensics and deepfakes: An overview. *IEEE Journal of Selected Topics in Signal Processing*, 14(5):910–932, 2020.

[2] Temmermans et al. JPEG Privacy and Security framework for social networking and GLAM services. *EURASIP Journal on Image and Video Processing*, 2017, 2017.

[3] Delgado J, Llorente S. Improving Privacy in JPEG images. *in Proceedings of the IEEE International Conference on Multimedia & Expo Workshops (ICME), Seattle, CA*, 2016.

[4] JPEG. `https://www.jpeg.org/`.

[5] C2PA. Coalition for Content Provenance and Authenticity. `https://c2pa.org`.

[6] C2PA. C2PA Technical Specification v1.0. `https://c2pa.org/specifications/specifications/1.0/specs/C2PA_Specification.html`.

[7] ISO/IEC. ISO/IEC JTC1SC29/WG1 N100157, Call for Proposals for JPEG Fake Media. `https://ds.jpeg.org/documents/jpegfakemedia/wg1n100157-095-REQ-Final_Call_for_Proposals_for_JPEG_Fake_Media.zip`, 2022.

[8] JPEG. JPEG Systems. `https://jpeg.org/jpegsystems/index.html`.

[9] JPEG. JPEG Fake Media. `https://jpeg.org/jpegfakemedia/index.html`.

[10] ISO/IEC. ISO/IEC IS 19566-5:2019, Information Technologies - JPEG Systems - Part 5: JPEG Universal Metadata Box Format (JUMBF), 2019.

[11] ISO/IEC. ISO/IEC DIS 19566-5:2022, Information Technologies - JPEG Systems - Part 5: JPEG Universal Metadata Box Format (JUMBF), 2022.

[12] ISO/IEC. ISO/IEC IS 19566-4:2020, Information Technologies - JPEG Systems - Part 4: Privacy and security, 2020.

[13] CAI. Content Authenticity Initiative. `https://contentauthenticity.org`.

[14] Microsoft. Project Origin. `https://innovation.microsoft.com/en-us/project-origin`.

[15] ISO/IEC. ISO/IEC JTC1SC29/WG1 N100156, Use Cases and Requirements for JPEG Fake Media. `https://ds.jpeg.org/documents/jpegfakemedia/wg1n100156-095-REQ-Use_Cases_and_Requirements_for_JPEG_Fake_Media.pdf`, 2022.

[16] Spring Boot framework. `https://spring.io/projects/spring-boot`.

[17] Fotos N., Delgado J. ISO/IEC JTC1SC29/WG1 M96014, UPC Proposal for JUMBF (ISO/IEC 19566-5) Ed2 Reference Software. `https://github.com/nickft/mipams-jumbf/blob/main/docs/UPC%20proposal%20for%20JPEG%20Systems%20JUMBF%20Ed2%20RefSW.pdf`, 2022.

[18] Fotos N., Delgado J. ISO/IEC JTC1SC29/WG1 M96015, UPC Proposal for JPEG Systems' Privacy and Security (ISO/IEC 19566-4) Reference Software v2. https://github.com/nickft/mipams-jumbf/blob/main/docs/UPC%20proposal%20for%20JPEG%20Systems%20Privacy%20and%20Security%20RefSW%20v2.pdf, 2022.

[19] IETF. RFC 8949: Concise Binary Object Representation (CBOR). https://www.rfc-editor.org/rfc/rfc8949.html, 2020.

[20] ISO 16684-1. Graphic technology — Extensible metadata platform (XMP) specification — Part 1: Data model, serialization and core properties, 2012.

[21] OASIS. eXtensible Access Control Markup Language (XACML) Version 3.0. https://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html, 2017.

[22] IETF. RFC 8152: CBOR Object Signing and Encryption (COSE). https://datatracker.ietf.org/doc/html/rfc8152, 2017.

[23] W3C. Verifiable Credentials Data Model v1.1. https://www.w3.org/TR/vc-data-model/, 2022.

[24] Oracle. Java Cryptography Architecture (JCA) Reference Guide.

[25] React. A JavaScript library for building user interfaces. https://reactjs.org/.

[26] Docker. https://www.docker.com/.

[27] RSA (cryptosystem). https://en.wikipedia.org/wiki/RSA_(cryptosystem).

[28] X.509 Certificate. https://en.wikipedia.org/wiki/X.509".

[29] OpenSSL. DER format. https://wiki.openssl.org/index.php/DER.

[30] JSON Web Tokens (JWT). https://jwt.io/.

[31] Toast UI ImageEditor. https://github.com/nhn/tui.image-editor.

[32] ExifTool by Phil Harvey. https://www.exiftool.org/.

[33] Delgado J, Llorente S, Reig G. Implementation of privacy and security for a genomic information system. *pHealth*, 2021.

[34] Distributed Multimedia Applications Group (DMAG). https://dmag.ac.upc.edu/.

[35] Information Modeling and Processing Research Group (IMP). https://imp.upc.edu/en.

[36] IPTC - NAA. Information Interchange Model Version No. 4.2. http://www.iptc.org/std/IIM/4.2/specification/IIMV4.2.pdf, 2014.

[37] Schema.org. ClaimReview: A Schema.org Type. https://schema.org/ClaimReview, 2015.

[38] ISO/IEC 23009-1:2012. Information technology – Dynamic adaptive streaming over HTTP (DASH) - Part 1: Media presentation description and segment formats, 2012.

[39] Rhayma, H., Makhloufi, A., Hamam, H. et al. Semi-fragile watermarking scheme based on perceptual hash function (PHF) for image tampering detection. *Multimed Tools Appl*, 80(17):26813–26832, 2021.

[40] ISO/AWI 24138. ISCC - Specification v1.x. `https://iscc.codes/specification/`, 2022.

[41] IETF. RFC 6960: X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. `https://www.rfc-editor.org/rfc/rfc6960`, 2013.