Technische Universität Berlin

Open Distributed Systems
Fakultät IV
Einsteinufer 25
10587 Berlin
https://www.tu.berlin/ods

Master's Thesis

Design and Implementation of Content Provenance using C2PA Signatures in Live Streaming

Philip Nys

Matriculation Number: 397914
09.07.2025

Supervised by
Prof. Dr. Manfred Hauswirth
Prof. Dr. Volker Markl

Assistant Supervisor
Stefan Pham

Hereby I declare that I wrote this thesis myself with the help of no more than the mentioned literature and auxiliary means.

Berlin, July 2, 2025

. . . .

(Philip Nys)

Abstract

It is getting more and more difficult to trust the media you see on the internet. The tools for editing, manipulating, creating and generating media are getting easier to use and more accessible every day.

There are tools, like image filters on social media, which essentially require no expertise to use at all, but also highly specialized tools for manipulating all kinds of media, like images, videos, audio, documents and more, with results that make it harder than ever to distinguish real from forged.

Additionally, there are also now a plethora of tools that are able to generated these kinds of media from just a single text prompt. The results of which are getting more and more convincing as these tools are being iterated upon.

All of this is creating new problems in society, like false perceptions of beauty, unbelievable documentations, faking personalities and content to scam people and many more.

This has lead to the founding of the Coalition of Content Provenance and Authenticity (C2PA) by large companies in the industry. Their goal is to create a standard of embedding metadata into all kinds of data types to document the road media took from creation to the consumption and spread its use to as many places as possible. This metadata is to detail everything involved in the creation, editing, distribution and anything else relevant of the corresponding media.

This data should provide users the option to inform themselves whether the media they are viewing is from a trustworthy source or perhaps may have been tampered with or is from a nefarious source.

Zusammenfassung

Es wird immer schwieriger Media im Internet zu vertrauen. Die Programme zum Bearbeiten, Manipulieren, Erstellen und Generierung von Media werden jeden Tag leichter zu bedienen.

Es gibt Programme, wie zum Bespiel Bildfilter in Soziel Medien, welche praktisch kein Vorwissen benötigen, aber auch hochspezialisierte Programme zur Bearbeitung von jeglichen Arten von Media, wie Bilder, Videos, Audio, Dokumente und mehr, mit Ergebnissen, welche schwerer denn je sind, echten Inhalt von Gefälschten zu unterscheiden.

Dazu kommen zahlreiche Programme, welche in der Lage sind diese Arten von Medien aus nur einer einfachen Textaufforderung zu erzeugen. Deren Ergebnisse mit der Zeit immer überzeugender werden, da sich diese Programma stetig verbessern.

All das führt zu neuen Problem in der Gesellschaft, wie zum Bespiel eine falsche Vorstellung von Schönheit, unglaubwürdigen Dokumenten, Identitätsraub und gefälschten Inhalten, welche verwendet werden, um Leute zu täuschen und auszurauben und Vieles mehr.

Dies hat zu der Gründung der Coalition of Content Provenance and Authenticity (C2PA), das Bündnis für Inhaltsherkunft und -authentizität, durch großen Unternehmen aus der Industrie geführt. Dessen Ziel ist es einen Standart zu entwicklen, welcher Eckdaten in all möglichen Datentypen einzubetten. Diese Daten sollen alle Schritten dokumentieren, die dieser Datentyp von Erstellung bis hin zum Endnutzer durchlaufen ist. Dies schließt jedes Detail ein, welche bei der Erstellung, Bearbeitung, Verteilung und jegliche weitere Information beigetragen haben.

Diesen Daten ermöglichen den Endnutzern die Option sich selbst für den Entstehungsweg des entsprechenden Mediums zu informieren und sicher stellen, dass Dies aus einer vertrauenwürdigen Quelle kommt oder manipuliert wurde oder möglicherweise eine schädlichen Ursprung hat.

Contents

List of Figures xi

List of Tables xiii

List of Algorithms xv

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Objective 2
- 1.3 Scope 2
- 1.4 Outline 3

2 State of the Art 5
- 2.1 HTTP Adaptive Streaming 5
- 2.2 C2PA 6
  - 2.2.1 Assertion 6
  - 2.2.2 Claim 7
  - 2.2.3 Claim Signature 7
  - 2.2.4 Manifest 7
  - 2.2.5 Manifest Store 7
  - 2.2.6 BMFF Hash Assertion 7
  - 2.2.7 Signing Fragmented BMFF Files 13

3 Requirements 15
- 3.1 Overview 15
- 3.2 Technical Requirements 15
  - 3.2.1 Producer 15
  - 3.2.2 Signer 15
  - 3.2.3 Distributor 16
  - 3.2.4 Consumer 16

4 Design 17
- 4.1 Producer 17
  - 4.1.1 Command Line Interface 17
  - 4.1.2 Settings Files 20
- 4.2 Signer 21
  - 4.2.1 Command Line Interface 21

4.2.2 Signing Approaches . . . 22
4.3 Distributor . . . 25
4.3.1 Command Line Interface . . . 25
4.4 Consumer . . . 26

5 Implementation **29**
5.1 Environment . . . 29
5.2 Project Structure . . . 29
5.3 Important Implementation Aspects . . . 30
5.3.1 Signing Optimizations . . . 30
5.3.2 Extending the Client C2PA Package . . . 34
5.3.3 Styling the Video Element . . . 36
5.3.4 Reading the `c2pa.hash.bmff.v2` Assertion . . . 37
5.3.5 CDN Caching . . . 39
5.3.6 Producer Inputs . . . 40
5.4 Documentation . . . 42
5.4.1 Producer . . . 42
5.4.2 Signer . . . 44
5.4.3 CDN . . . 44
5.4.4 Consumer . . . 44

6 Evaluation **47**
6.1 Test Environment . . . 47
6.1.1 Signing Duration . . . 47
6.1.2 Data Upload Size Required . . . 48
6.2 Performance Measurements . . . 49
6.2.1 Signing Duration . . . 49
6.2.2 Data Upload Size Required . . . 50

7 Conclusion **57**
7.1 Summary . . . 57
7.2 Dissemination . . . 58
7.3 Outlook . . . 58

List of Acronyms **61**

Bibliography **63**

Annex **65**

List of Figures

1.1 Proof of Concept Testbed Setup 2

2.1 Example Manifest Store [1] 8
2.2 Simple Merkle Tree Example 11

4.1 Default Audio Settings File 20
4.2 Default Video Settings File 21
4.3 Example Merkle Tree Visualization 26
4.4 Example Rolling Hash Visualization 27

5.1 Seekbar Video Styling 38
5.2 Video Content Credentials Overlay 38
5.3 Video Content Credentials Overlay With Errors 39

6.1 Input File Sizes 48
6.2 Signing Duration Results 51
6.3 Signing Duration Results 52
6.4 Upload Size Results 54
6.5 Upload Size Results 55

.1 Example Incomplete Merkle Tree Visualization 67

List of Tables

4.1 Producer CLI Options 18
4.2 Local Command Options 19
4.3 Standard Input Command Options 19
4.4 Device Command Options 19
4.5 Test Command Options 20
4.6 Signer CLI Options 21
4.7 Live Command Options 22
4.8 CDN CLI Options 26

5.1 Hardware and Software Environment 29
5.2 Signer Environment Variables 44
5.3 CDN Environment Variables 44

List of Algorithms

2.1 Validating a Fragment 13

6.1 Duration Measuring Method 48

1 Introduction

1.1 Motivation

For millennia human beings have been forging documents, paintings, photographs, videos, and any other type of media, ranging from analog manipulations, like altering shipping manifests on clay tablets, duplicating paintings and altering the development of photos and films, to digital tools with varying levels of difficulties, from complex tools like PhotoShop, GIMP and Photopea for images, AfterEffects, Blender, and Nuke for videos, to easy tools like image filters on social media.

In this digital age, these tools are not only becoming more accessible, but also easier to use, while delivering better results than ever. This has the consequence that it is becoming more difficult to discern real media from manipulated media, especially for people who are less familiar with these tools.

To make matters worse, in the past few years many new tools have been released that are capable of generating new media from just a simple text prompt. Large Language Models (LLMs), like ChatGPT, Gemini, and DeepSeek, are capable of generating texts, like essays, articles, and more. There are also image generators, like Midjourney, Stable Diffusion and Firefly, as well as Sora, Synthesia, and Capsule for videos. The list of these tools grows steadily every day, with numerous additional application types: music generation, voice cloning, face replacements, and many more.

All this has led to the founding of the Coalition for Content Provenance and Authenticity (C2PA) by Adobe, Arm, BBC, Intel, Microsoft, and Truepic on February 22nd, 2021 [1]. The ultimate goal of C2PA is the development and integration of a taper-proof manifest into digital media, which protocol all steps it has gone through from its creation to the present, as well as the tools, people, devices and locations involved and any additional relevant metadata. This allows everyone interacting with C2PA-signed media to verify that that media is trustworthy and look at the metadata and potentially see that it has been automatically generated, edited, applied with a filter, or anything else.

At the writing of this thesis, there is no specification in regards to applying C2PA to live streaming media and there is also not much material about other people researching into this topic. This thesis will fill in this gap by implementing and evaluating C2PA in a live streaming context.

[1] C2PA Founding Press Release: `https://c2pa.org/post/c2pa_initial_pr/`

1.2 Objective

The current version 2.2 of the C2PA technical specification [2] has no explicit specifications for live streaming content and there are currently no public discussions or proof-of-concepts on how the C2PA signing should work on live streaming media. However, there is a specification with a working implementation for fragmented BMFF (Base Media File Format) media.

This thesis will describe and evaluate a proof-of-concept of adapting the existing fragmented BMFF implementation to a live streaming testbed compliant with the DASH and HLS streaming protocols. In addition to this, it will also propose an alternative approach that is specifically catered to live streaming.

1.3 Scope

The aforementioned proof-of-concept is a live streaming testbed, which will roughly emulate a real-world scenario. It consists of four components.

The first component is the Producer with the task of creating a DASH and HLS live stream. That live stream is then forwarded to the Signer, which will sign the live stream with a C2PA manifest. The signed live stream is then published to the Distributor, an emulated CDN (content delivery network) hosting the live stream to make it available for consumption. Finally, the Consumer will playback the live stream, while simultaneously validating the trustworthiness of the embedded C2PA manifest. This testbed is visualized in Figure 1.1.

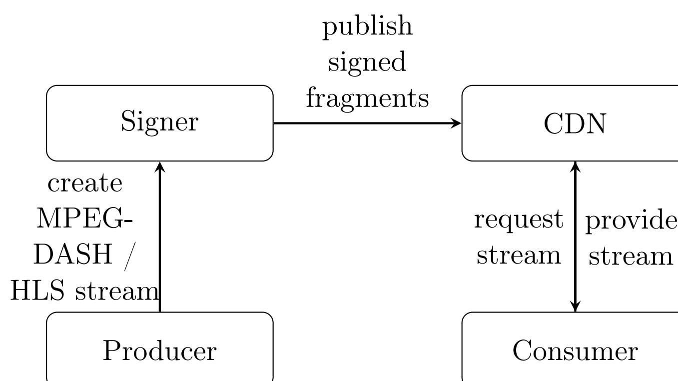


Figure 1.1: Proof of Concept Testbed Setup

[2] C2PA Technical Specification: `https://c2pa.org/specifications/specifications/2.2/specs/C2PA_Specification.html`

1.4 Outline

The remaining thesis is structured into the following chapters:

Chapter 2 will describe the State of the Art with an introduction to HTTP Adaptive Streaming (HAS), an in-depth overview of C2PA, as well as related works.

Chapter 3 will list the requirements of this thesis and its implementation.

Chapter 4 will detail the design of the testbed, including specifics on the four components.

Chapter 5 will outline specific technical aspects of the implementation.

Chapter 6 will evaluate C2PA in a live streaming scenario as part of this proof-of-concept.

Chapter 7 will conclude this thesis with a summary, dissemination, and an outlook to possible further research.

2 State of the Art

2.1 HTTP Adaptive Streaming

For more than 15 years, HTTP adaptive streaming (HAS) has been the defacto standard media streaming protocol with the release of HTTP live streaming (HLS) [1] in 2009 [2] and subsequently in 2012 [2] with the release Dynamic adaptive streaming over HTTP (DASH).

In both of these protocols, a media stream is split into a number of different representations, each being an alternate version of the original stream, for example different quality levels for video or in case of audio and subtitles, they can also represent different languages. Each of these representations in-turn has its media stream split into small fragments. These fragments enable the media players to dynamically switch the currently playing representations to adapt to the current situation, they also enable separation of the acquisition of the stream into many individual HTTP requests instead of downloading the entire stream in one go.

Both of these protocols make use of a manifest to give the media player all relevant information to acquire the media stream. In case of DASH, this manifest is the Media Presentation Description (MPD). This MPD is an XML [3] file, which contains every detail about one specific media stream. In contrast to this approach, HLS splits this information into one Master Playlist and separate Media Playlists. These files use the file ending `.m3u8`. They are encoded as plain text UTF-8 [4] files with a proprietary method of encoding the encapsulated information. The Master Playlist contains information about the individual representations, as well as references to the Media Playlists, each of which corresponds to one specific representation.

The MPD and Master Playlist act as the entry points of their respective media stream and are used to initialize a media player to play that stream. The players download these files and use the encoded information to play the media stream. While the media is playing, the players analyze the current streaming situation and based on a number of parameters, like network conditions, buffer level, playing quality, available qualities, etc. employ an adaptive bitrate (ABR) algorithm to determine whether a representation switch is required to ensure smooth playback.

[1] HLS for Developer: `https://developer.apple.com/streaming/`
[2] DASH First ISO Standard: `https://www.iso.org/standard/57623.html`

2.2 C2PA

As previously mentioned, the goal of C2PA is the creation of a tamper-proof method of embedding metadata into media. This metadata is supposed to contain all relevant information about the creation of said media as well as all the editing steps applied to it. Ultimately, this data encodes the provenance and authenticity of media to provide transparency, knowledge, and trust in it.

The technical specifications of C2PA detail the motivations, goals, and methodologies. These specifications are currently on version 2.2 [3], however, when I started work on this thesis the C2PA implementation, this work is based on, was still on version 2.0, thus I will be referencing version 2.0 for everything related to the technical specifications [4]. Everything in this section is based on version 2.0, unless otherwise denoted.

Now, I will give an example of what an ideally integrated life-cycle of media content with C2PA would look like, using a photograph:

A photographer takes an image using a camera. This camera has integrated C2PA capabilities and embeds C2PA metadata into the image. This metadata contain information , like the location where the image was taken, model name of the camera, camera settings, like ISO values, zoom level, used to take the image, EXIF data, credentials of the photographer who took the image, automatic processing steps applied by the camera and possibly any additional data the camera was configured to include. The photographer then takes this image to an editor who further edits this image with software. This program also has integrated C2PA capabilities and extends the already embedded C2PA metadata with information about the editing process. This additional data includes information about color corrections, lighting changes, cropping, removal and addition of objects, application of filters, the editor's credentials and any other image manipulations applied to the image. The image is now ready for publication. It is uploaded to various social media websites and each of these services apply modifications to the image upon uploading it to their servers, like compression to make them more portable, conversion to a different data type or other steps. These are also C2PA integrated and they further extend the C2PA metadata by including the processing formation, service that applied the changes and so on. The image is now viewable by everyone on the respective platforms. The websites and programs displaying this image are integrated with C2PA validation tools and verify that the image hasn't been tampered with by validating the C2PA metadata. They then indicate that the image is trustworthy and also provide the option for users to look at the metadata.

2.2.1 Assertion

An `Assertion` encapsulates a piece of information about the media asset. It is created during the signing process. Assertions are an integral part of the C2PA Manifest. Each

[3] C2PA Technical Specification v2.2: `https://c2pa.org/specifications/specifications/2.2/specs/C2PA_Specification.html`

[4] C2PA Technical Specification v2.0: `https://c2pa.org/specifications/specifications/2.0/specs/C2PA_Specification.html`

Assertion is identified by an unique label. There are predefined labels and Assertion data but it is also possible to include any type of proprietary data as Assertion.

An example of a predefined Assertion follows in Section 2.2.6.

2.2.2 Claim

The `Claim` has references to all Assertions of the C2PA Manifest. It is digitally signed and tamper-proof. The Claim is also an integral part of the C2PA Manifest.

2.2.3 Claim Signature

The `Claim Signature` is the final part of the C2PA Manifest and it is the digital signature of the Claim.

2.2.4 Manifest

The `C2PA Manifest` contains the three aforementioned parts: the Claim Signature, the Claim and the Assertions. Each Manifest contains one Claim Signature, one Claim and a set of Assertions. They are contained in the Manifest Store of a media asset.

2.2.5 Manifest Store

The C2PA metadata is wrapped in the `Manifest Store`. A Manifest Store contains all C2PA Manifests. The last of these Manifests is the so called Active Manifests, which is the most recently added Manifest and contains the credentials that are verifiable. The Manifest Store can be either directly embedded into the media asset or it can be referenced in the asset as external.

A sample Manifest Store can be seen in Figure 2.1.

2.2.6 BMFF Hash Assertion

The BMFF Hash Assertion (label: `c2pa.hash.bmff`) is the C2PA Data Hash [5] Assertion embedded in fragmented BMFF media files and is required in these files for validation. This Assertion is encapsulated by the Rust data structure `BmffHash`, see Listing 2.1. It contains the exclusion ranges used to create the data hashes of the fragmented files (struct field `exclusions`), the name of hashing algorithm used (struct field `alg`), the data hash, as byte buffer, of the corresponding file (struct field `hash` / this field is not used for BMFF media which is split into multiple files, this use case of the thesis), the Merkle Trees (struct field `merkle`) and the BMFF hashing version used (struct field `bmff_version`).

[5] Data Hash: `https://c2pa.org/specifications/specifications/2.0/specs/C2PA_Specification.html#_data_hash`

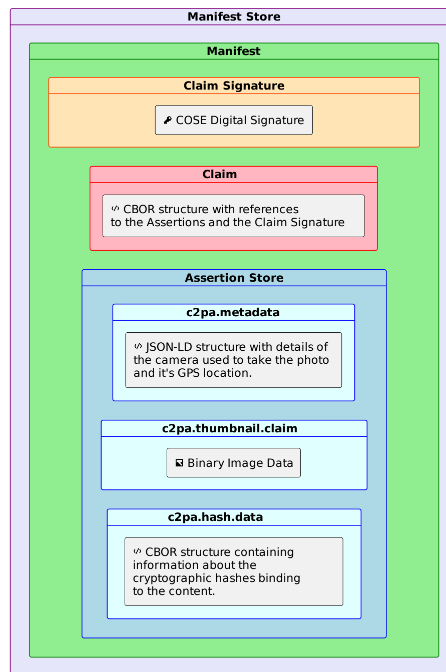


Figure 2.1: Example Manifest Store [1]

```
pub struct BmmfHash {
    // the exclusion ranges
    exclusions: Vec<ExclusionMap>,
    // name of the hashing algorithm
    alg: Option<String>,
    // the data hash
    hash: Option<ByteBuf>,
    // the Merkle Tree
    merkle: Option<Vec<MerkleMap>>,
    // name of the assertion
    name: Option<String>,
    // deprecated beginning with BMFF Version 2
    url: Option<UriT>,
    // BMFF Version, here 2
    bmff_version: usize,
}
```

Listing 2.1: BmffHash Rust Definition

The Merkle Trees are described by an array of the Rust data structure `MerkleMap`, see Listing 2.2. Each Merkle Tree contains its unique ID (struct field `unique_id`), its local ID (struct field `local_id`), the number of leaves in this tree (struct field `count`), the

name of the hashing algorithm used (struct field `alg`), the data hash of the initialization fragment, as byte buffer, and the layer of the Merkle Tree stored for reference, more on that later, as array of byte buffer (struct field `hashes`).

```
pub struct MerkleMap {
    // unique ID
    pub unique:id: u32,
    // local ID
    pub local_id: u32,
    // number of leave
    pub count: u32,
    // name of hashing algorithm
    pub alg: Option<String>,
    // hash of initialization fragment
    pub init_hash: Option<ByteBuf>,
    // Merkle Tree reference layer
    pub hashes: VecByteBuf,
}
```

Listing 2.2: MerkleMap Rust Definition

BMFF-Based Hashing

The hashing of BMFF-based data has certain requirements set by the specifications that need to be followed. All the data of a BMFF file must be included in the hashing data, except specific parts as defined by the `ExclusionMap` list. These exclusions are included in the corresponding Assertion, `BmffHash` in this case, as mention in the preceding section.

The Rust data structure of an exclusion is detailed in Listing 2.3. It contains one mandatory field `xpath`, which describes the path to the corresponding BMFF Box, similar to a file system path, using the BMFF Box names according to `4cc` notation. When a specific BMFF Box occurs multiple times they can be addressed by their index using array notation and positive non-zero indices. All remaining data fields are optional. When all fields are left out, the BMFF Box specified by `xpath` is to be excluded in its entirety. The next field is `length`, which denotes the length a BMFF Box must have to be excluded from the hash, including the box header. The `data` field is a list of `DataMap` type, which holds an byte offset and a data value, if the BMFF Box contains the exact data value at the specified offset it is to be excluded from the hash. The `subset` data field allows exclusion of specific data ranges. It is a list of `SubsetMap`, which encapsulates a byte offset and a length. Data is excluded from the hash starting at the offset for the specified length. The offsets in the list must be monotonically increasing and no subset must overlap with another. The final entry can have a length of `0` to indicate the exclusion of the remainder of the BMFF Box. The length can also exceed the BMFF Box boundary to achieve the same as length `0`. The field `version` indicates to exclude the BMFF Box if it has this version, this can only be set for BMFF Boxes which are FullBoxes. The

`flags` is a byte string of exactly three bytes and if the BMFF Box has these flags set it must be excluded. This can also only be applied to FullBoxes. The final field `exact` is a boolean and can only be set if `flags` is also set and indicates, whether the flags must match exactly to exclude the BMFF Box from the hash. Otherwise the flags must have at least the same bits set and additional flags can also be set. If `exact` if not set it defaults to `true`.

```rust
pub struct ExclusionMap {
    // path to the BMFF Box
    pub xpath: String,
    // exclude if box has this exact length
    pub length: Option<u32>,
    // exclude when exact data is contained
    pub data: Option<Vec<DataMap>>,
    // exclude specific data ranges
    pub subset: Option<Vec<SubsetMap>>,
    // exclude if the BMFF Box has this version
    pub version: Option<u8>,
    // exclude if these flags are set
    pub flags: Option<ByteBuf>,
    // exclude if flags match exactly
    pub exact: Option<bool>,
}
```

Listing 2.3: ExclusionMap Rust Definition

In case the C2PA Manifest is embedded directly into an BMFF file, the BMFF Box it is contained in must be excluded from the hash.

When a BMFF Box is removed after the C2PA Manifest has been created, it should be replaced with a `"free"` BMFF Box with the same length to ensure the hash is not invalidated by the removal of the Box. If the removal of a Box is expected to happen, the effected Box shall be replaced by a `"free"` BMFF Box of the same size for the duration of the signing. The placeholder must then be excluded from the hash. Once the signing is completed, the Box needs to be put back in its original place.

Since adding C2PA Manifest into BMFF-based file via MP4 boxes changes the positions of these Box, the position of root BMFF Box needs to be included in the hash to ensure the correct placement of Boxes.

Merkle Tree

A Merkle Tree can be used to create a hash signature of a large dataset by fragmenting it into smaller pieces. Then it is possible to validate one of these pieces as part of the whole dataset without needing to have access to the entire dataset. The C2PA specifications use a Merkle Tree for fragmented BMFF media.

A Merkle Tree is a binary tree that is built from the bottom up. The first row are the leaves of the Merkle Tree. The leaves are the fragmented pieces of the dataset, in this

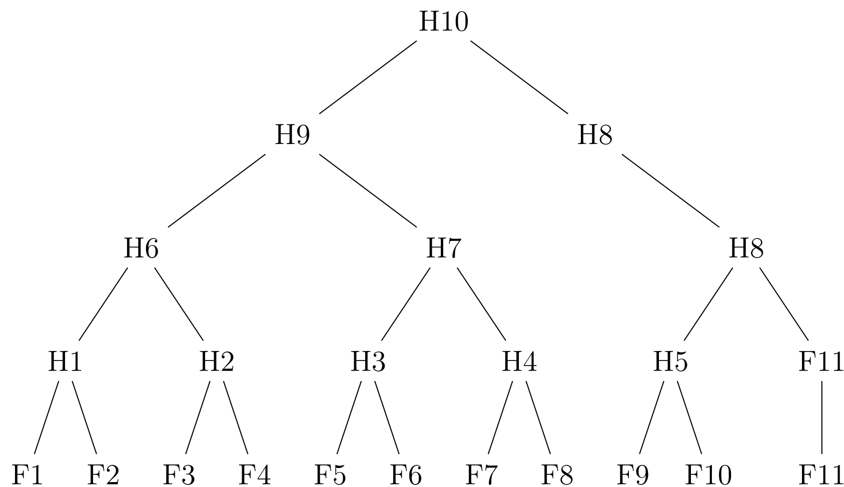


Figure 2.2: Simple Merkle Tree Example

case the media fragments, more specifically they are the hashes of the fragments accoring to Section 2.2.6. The next row contains the parents of the previous row's nodes. If a parent node has both left and right children, then the data hash of the parent node is the hash of the two children hashes concatenated, otherwise the children node hash is copied. This is repeated until the row with only a single node has been reached, this node is the root of the Merkle Tree [5].

Once the Merkle Tree is fully built, only one of the layers has to be kept in memory or stored in a data structure, I am calling this the `reference tree layer` from here on. It is up to the application, which layer is used as the `reference tree layer`. The weight, between number of hashes stored and number of steps required for validation, has to be considered, when choosing which layer to keep. The higher the layer used, the smaller this layer is, but the more nodes, the proof hashes, are needed to validate individual fragments. These proof hashes are assigned to each leaf. The proofs are hashes that are part of the full Merkle Tree and they are the hashes, which are required to validate a leaf. They are the sibling nodes along the path from the leaf up to the reference tree layer.

A simple example of this can be seen in Figure 2.2. For example the hash `H3` is the resulting hash of the concatenation of fragment hashes `F5` and `F6`, while the hash `H9` is the result using the hashes `H6` and `H7`. The fragment hash `F11` and hash `H8` are examples of the case where there is one child missing and that node is simply copied. In this example, hash `H10` is the root of the Merkle Tree.

Validation

The initialization fragment can be validated standalone. However, fragments can only be validated in conjunction with the corresponding initialization fragment.

The initialization fragment contains the C2PA Manifest packaged in a `uuid` BMFF

box. In a fragmented BMFF context this manifest must contain the aforementioned `c2pa.hash.bmff` Assertion. The initialization fragment is verified by decoding its embedded C2PA Manifest. From the BMFF Hash Assertion in this manifest, the exclusion ranges and the initialization hash are used to replicate the data hash of the initialization fragment and then compare, whether the hashes match. Matching hash indicate a valid C2PA Manifest, otherwise the file has tampered with.

Each fragment also contains a `uuid` box. However, the fragments only contain a CBOR-encoded [6] data structure, which holds information required for validation: the proof `hashes`, the `local_id` and `unique_id` and the `location` on that particular Merkle Tree, shown by the Rust structure BmffMerkleMap in Listing 2.4. The two IDs are required to identify the corresponding Merkle Tree from the C2PA manifest in the initialization fragment and the location is the leaf index of the fragment which is needed to determine whether this fragment is the left or right child of their parent node.

```
pub struct BmffMerkleMap {
    // unique ID
    pub unique_id: u32,
    // local ID
    pub local_id: u32,
    // leave index on the Merkle Tree
    pub location: u32,
    // proof hashes
    pub hashes: Option<VecByteBuf>,
}
```

Listing 2.4: BmffMerkleMap Rust Definition

The validation of a fragment requires the successful validation of the accompanying initialization fragment. The first step of the validation is the calculation of the data hash using the exclusion ranges according to the C2PA manifest. Next, the proofs are used in sequence to partially reconstruct the Merkle Tree. This is done by first determining whether the fragment is a left or right node. The `location` values begin at zero, indicating that an even `location` represents a left node and an odd `location` a right node. The first proof is the direct sibling of the fragment. The first parent is calculated by concatenating the first proof hash with the data hash of the fragment (left + right) and hashing the result using the hashing algorithm specified in the C2PA Manifest. This is repeated with the newly created hash and the next proof until all proofs have been used. The final hash resulting from these steps should equal one of the hashes located in the `reference tree layer` from the C2PA manifest. If that is the case, the fragment has been validated as trustworthy [6]. A simplified version of the can be seen as pseudo code in Algorithm 2.1.

[6] BMFF-Based Hash Validation: `https://c2pa.org/specifications/specifications/2.0/specs/C2PA_Specification.html#_validation`

Algorithm 2.1 Validating a Fragment

Require: $refTreeLayer$ ▷ read from C2PA Manifest
Require: $location$ ▷ read from Fragment

1: $fragHash \leftarrow hash(fragment)$
2: **for all** $proof \leftarrow fragment.proofs$ **do**
3:     **if** location is right **then**
4:         $fragHash \leftarrow hash(proof + fragHash)$
5:     **else**
6:         $fragHash \leftarrow hash(fragHash + proof)$
7:     **end if**
8: **end for**
9: **if** fragHash in refTreeLayer **then**
10:     fragment is valid
11: **else**
12:     fragment is invalid
13: **end if**

2.2.7 Signing Fragmented BMFF Files

As previously mentioned, in the context of a fragment MP4 video, the initialization fragment file contains the `"uuid"` BMFF Box with the Manifest Store and each corresponding media fragment contains a `"uuid"` BMFF Box with the CBOR-encoded `BmffMerkleMap`.

Currently, the signing process is only intended to be used for VoD content. It requires a full set of media fragments and the initialization fragment. The BMFF-specific parts of the signing process starts with the creation of a placeholder Merkle Tree based on the number of media fragments. In the next step the fragments all receive a preliminary `BmffMerkleMap` embedded into their `"uuid"` BMFF Box, it contains the final values for the two ID fields and the location but the hashes are only placeholder data. Now that the media fragments have the embedded `"uuid"` BMFF Box, they are ready to be hashed. These hashes can now be used to generate the final Merkle Tree and this in-turn can be used to insert the correct proof hashes in the media fragments. Now, any of the rows of the Merkle Tree can be added into the `MerkleMap` alongside a placeholder hash of the initialization fragment. This preliminary data is now inserted into the initialization fragment, which is now ready to be hashed. This hash replaces the previously set placeholder.

3 Requirements

This chapter will outline the components required to fulfill the task of implementing a testbed capable of producing, signing, publishing, playing, and validating a live stream as trustworthy using C2PA Manifests.

3.1 Overview

To reiterate the task, the testbed is supposed to be able to create a live stream. The created live stream is to be signed with a C2PA Manifest and then made available to be watched and also verified that the contained C2PA Manifest is valid.

3.2 Technical Requirements

These requirements can be separated into four components:

1. **Producer**: creating a Live Stream
2. **Signer**: embedding the C2PA Manifest into the Live Stream
3. **Distributor**: hosting the Live Stream for consumption
4. **Consumer**: playing and validating the Live Stream

3.2.1 Producer

The Producer must be able to create a live stream that is compliant with the DASH and HLS protocols, ideally simultaneously. In addition, it would be beneficial to be able to create live streams from various inputs, i.e. a camera, screen recording, video file, etc.

Furthermore, the Producer needs to be able to interface with the Signer to be able to sign the Live Stream as it is being created with as little latency impact as possible.

3.2.2 Signer

The Signer has to be able to receive the live stream from the Producer. Once the live stream has been received, it needs to be periodically signed with a C2PA Manifest. Additionally, the signing of the live stream must be optimized to the point where it affects the viewing experience as little as possible.

There already is an open-source implementation of the C2PA specifications: `c2pa-rs` [1] written in Rust. This implementation includes the ability to sign fragmented BMFF content and a CLI (command line interface) program. These two offer an ideal baseline for this task.

Once the live stream has been signed, it needs to be published to a Distributor.

3.2.3 Distributor

The Distributor should emulate a CDN. It needs to be able to have content ingested and then digested to viewers in fashion, which does not introduce noticeable delays into the process.

3.2.4 Consumer

Finally, the Consumer must be able to playback the live stream from the CDN using DASH and HLS conform media players. A website is the ideal candidate for this. It offers a robust environment to create a graphical user interface, it is very portable, and is a very common platform to consume media on.

In addition, there are existing implementations for the media players, like `dash.js` [2] and `hls.js` [3] for the DASH and HLS protocols, respectively. Likewise there is an existing implementation for validating C2PA Manifests for a wide range of content types: `c2pa` [4].

[1] `c2pa-rs` GitHub Repository: `https://github.com/contentauth/c2pa-rs`
[2] `dash.js` GitHub Repository: `https://github.com/Dash-Industry-Forum/dash.js`
[3] `hls.js` GitHub Repository: `https://github.com/video-dev/hls.js`
[4] `c2pa` NPM package: `https://www.npmjs.com/package/c2pa`

4 Design

In this chapter, I will detail the design of the implemented testbed shown in Figure 1.1.

4.1 Producer

The Producer is a highly configurable Rust program capable of reading special settings-files with which a FFmpeg [1] command is generated. FFmpeg is a powerful command line tool for audio and video processing.

The generation process starts with the parsing of the CLI, specifics follow in Section 4.1.1, followed by the reading of the settings files, according to Section 4.1.2. Using the parameters gathered from the CLI and settings-files, the Producer generates a shell scripts. The Producer supports a plethora of input types, see Section 5.3.6 for full details.

When a local video file is configured as the input, the FFmpeg command will create a live stream by looping through the video endlessly. Video devices usually represent the built-in laptop webcam, but it is also possible to plug in an external camera via USB and use that as an input, the setup process is described in the following Section 5.3.6. Remote video sources can be anything supported by FFmpeg that can be addressed using an URL, for example remotely hosted DASH/HLS live streams or VoDs (videos on demand) or IP-cameras.

Once the Producer has finished generating the FFmpeg shell script, the script can be executed in a terminal to run the live stream creation. The FFmpeg command will create the live stream using the output format "dash" and point the output to an HTTP server. The "dash" output format also has a convenient option to create HLS master and media playlist files alongside the DASH stream.

An example shell scripts is shown in Listing 1.

4.1.1 Command Line Interface

As previously mentioned, the Command Line Interface (CLI) of the Producer has many options to configure it for many different use-cases. The Producer is primarily intended to create DASH live streams, which is why the output flag must point to a DASH Manifest MPD file with either a local path or an URL. The audio and video settings flags are optional and when not provided, the Producer will automatically generate the settings files and re-use them on subsequent uses.

To simplify the usage of these complex options, I have created a helper script to easily use the Producer with a few pre-configured methods during development, details will

[1] FFmpeg Homepage: `https://www.ffmpeg.org/`

follow in Section 5.4.

Short Flag	**Long Flag**	**Value**	**Description**
-o	–output	Path/ URL	The path to the output DASH Manifest file (*.mpd)
-a	–audio	Path	The path to the audio settings file
-v	–video	Path	The path to the video settings file
-s	–script	Path	The path to the script file to create
	–ll	boolean	Enable Low Latency
-t	–title	String	Set the title of the stream
	–hls	boolean	Enable concurrent HLS Manifest creation
	–an	boolean	Disable audio
	–vn	boolean	Disable video
-k	–keyframes	u8	Number of keyframes per Fragment
	–fps	f32	Frame per second
	–segdur	f32	Target Fragment Duration
	–embed-settings	boolean	Embed selected encoding settings into the video

Table 4.1: Producer CLI Options

The input of the live stream is configured using a subcommand of the Producer. I will give each type of input its own section.

Remote

The **remote** command uses an external resource as input. Anything that can be addressed using an URL and that is natively supported by FFmpeg can be used. For example publicly hosted DASH / HLS streams, IP-cameras or more.

The only option this command has is a positional argument pointing to the remote input in form of an URL.

Local

The **local** command takes a local file as input. This is essentially the same as the remote command, the difference being that it uses an local path instead of a remote URL. The same restrictions apply.

The path is supplied with a positional argument. However, it has a second option. This being whether or not to create a live stream. For example, when using a video file as input, this video has a finite runtime. Setting this flag will make FFmpeg loop through the file endlessly to create a live stream.

Short Flag	Long Flag	Value	Description
		Path	The path to the input file.
-l	–live	boolean	Loop through the video file

Table 4.2: Local Command Options

Standard Input Pipe

The **std-in** command uses the standard input pipe to receive raw video data as input for FFmpeg.

The only required option is the resolution of the video data, given with the height and width of the video in number of pixels.

Short Flag	Long Flag	Value	Description
	–w	u16	The pixel width
	–h	u16	The pixel height

Table 4.3: Standard Input Command Options

Device

The **device** command uses a capture device as input. The most common use-case for this would be the webcam of a laptop. On Linux, device are addressed using a path to the device directory (`/dev/*`), for example the built-in webcam of a laptop is typically the `/dev/video0` device, which this command defaults to.

As before, the path is given as argument with the additional option to set which input format to use.

Short Flag	Long Flag	Value	Description
		Path	The path to the capture device
-f	–format	String	Input format

Table 4.4: Device Command Options

```
    name,sampling,bitrate
default,    48000, 128000
```

Figure 4.1: Default Audio Settings File

Screen

The **screen** command uses the device's screen as input. I have not fully developed this command to be completely usable, since this turned out to be more complex than I initially expected it to be and I needed to focus on more important aspects. The original intention was to have it work similar to other screen capture tools where the user can select a specific window, screen or sub-region of a screen with a resolution and capture frame rate to create a live screen.

Test

The **test** command is the final input method. It uses the built-in FFmpeg virtual input format `lavfi` to generate a test image. This test image has a few vertical colorful bars, a horizontal scrolling gradient with a timestamp counting up the seconds the has been stream running for. Examples for this test image can be seen in Figure .1, Figure 4.3 and Figure 4.4.

This input can be configured with a starting resolution, frame rate and an optional duration of the generated video. If the duration is omitted, it will create an endless stream. Otherwise, the duration is provided in milliseconds.

Short Flag	Long Flag	Value	Description
	–w	u16	The pixel width
	–h	u16	Loop through the video file
-r	–fps	f32	Input frame rate
-d	–dur	u64	Optional duration in milliseconds

Table 4.5: Test Command Options

4.1.2 Settings Files

The settings files are two CSV-files, one for audio and video each. These files are automatically generated the first time the Producer is run. The initial default configurations are shown in Figure 4.1 for audio and video in Figure 4.2 (both are formatted for better readability). These files can be configured by adding new lines with desired settings or by removing lines with unwanted settings, alternatively they can also be made to be ignored by Producer by adding a '`#`' in front of a line.

```
name,resolution , bitrate ,max_rate,buffer_size
#144p,     256x144,    95000,   100000,      150000
#240p,     426x240,   150000,   160000,      240000
#360p,     640x360,   276000,   290000,      430000
#480p,     854x480,   750000,   775000,     1200000
 720p,    1280x720,  2048000,  2200000,     3300000
#1080p,  1920x1080,  4096000,  4300000,     6500000
#1440p,  2560x1440,  6144000,  6500000,    10000000
#2160p,  3840x2160,17408000,18000000,    27000000
```

Figure 4.2: Default Video Settings File

4.2 Signer

The Signer is an extension of the existing `c2patool` command line program and the underlying `c2pa-rs` Rust crate. It has been extended by an additional subcommand to run the tool in live signing mode, alongside the implementation of signing processes optimized for live streaming purposes. The live subcommand turns the `c2patool` into an HTTP server. The Signer will then receive the live stream from FFmpeg via HTTP requests. With every new fragment received the stream will be signed, details of the signing and the optimizations are detailed in Section 5.3.1 and the alternative approach is presented in Section 4.2.2. Finally, the finished files are then sent to the Distributor.

4.2.1 Command Line Interface

The CLI of the Signer, which is actually the `c2patool` is quite extensive. In an effort to keep it concise and not too complex, I will only list the options that I am using in this testbed and those, I have added to implement live signing.

As before, I will begin with the general options. The path to the asset file is an argument, making it not require any designating flag. In normal operation, this file is the target of either signing or validation. However, since my live signing approach uses an HTTP server to receive files, I have no use for this argument. But since this argument is mandatory, I simply pass it arbitrary data and ignore it.

Short Flag	Long Flag	Value	Description
		Path	The path to the asset file.
-o	–output	Path	The path to the output file or directory
-m	–manifest	Path	The path to the C2PA Manifest JSON file

Table 4.6: Signer CLI Options

In addition to the general options the `c2patool` uses a few commands for additional configuration. As previously mentioned, I have added the "live" command to the tool. This command turns the `c2patool` into an HTTP server to receive the live stream from the Producer. The target URL is the ingestion URL of the CDN.

Short Flag	Long Flag	Value	Description
-b	–bind	Socket Address	The Socket Address on which the Signer will listen on.
-t	–target	URL	The CDN URL where to ingest the output to
-w	–window	usize	The number of Fragment in one Merkle Tree group

Table 4.7: Live Command Options

4.2.2 Signing Approaches

As explained in Section 2.2.7, the existing fragment BMFF signing implementation is designed to be used to sign a completed set of files, i.e. a VoD. As a preliminary experiment, I used this implementation to sign the created live stream in this testbed to see how it works. This approach does actually work and the fragments are all correctly validated. However, since this implementation always signs all files given to it, with every newly generated fragment, the entire live stream is signed anew. This caused the signing to take longer and longer with every fragment, continuously slowing down the machine it is running on and eventually it could no longer keep up and the live stream would begin to stall. Additionally, after every signing operation all fragments needed to published again to the CDN, which created a lot of overhead on the network and CDN, while also not letting the CDN effectively cache the fragments. I included this approach in my evaluation of this thesis to better show the effectiveness of my optimized implementations.

Next, I started the implementation of the optimized signing processes. I began with an optimized Merkle Tree approach as per the original proposal and the other alternatives were created in response to some of the drawbacks of the Merkle Tree approach.

Optimized Merkle Tree

As mentioned, the big problem with the existing Merkle Tree approach was the complete re-signing of files that had already been signed during a previous signing. To fix this problem, I made use of the fact that the C2PA Manifest is able to hold multiple Merkle Trees, as shown in Section 2.2.6. Each Merkle Tree has a unique and local ID and each fragment's embedded data has these IDs as well, which are used to assign a fragment to the matching Merkle Tree.

This approach required the addition of a new parameter to the signing function: the window size. This window size corresponds to the number of fragments each Merkle Tree

will hold at most. The window size can be configured using the CLI of the Signer as depicted in Section 4.2.1, for the duration of this thesis, I have used a window size of `8` as the default value. I chose this value for a number of reasons. Firstly, since a Merkle Tree is a binary tree it is a good idea to use a power of two value to completely fill all layers of a tree. Secondly, I figured that `8` strikes a decent balance between having enough fragments grouped together for security, while not having so many fragments that signing would take too much time.

This way only requires to sign the newest up to `8` (or window size) fragments of the live stream. Once a full group of `8` fragments has been signed, they are done and no longer need to be updated on the CDN. This approach greatly improved the performance of the live stream signing, however, it still came with a number of drawbacks:

- each group of fragments adds another Merkle Tree to the BMFF Hash Assertion in the C2PA Manifest, this could result in the C2PA Manifest becoming too big if the Live Stream runs for a very long time
- each fragment still has to be updated on the CDN up to `8` times
- the initialization fragment still needs to be updated every time a new fragment is created
- validation of every fragment requires the fetching of the newest initialization fragment to have the corresponding C2PA Manifest
- a live stream could be tampered with by replacing a full group of a Merkle Tree

The technical details of this approach are described in Section 5.3.1.

The next two approaches are extensions of this method and were created in response to the frequent uploading and downloading of essentially the same fragments.

Optimized Merkle Tree with C2PA Data on separate Server

In this approach, the C2PA data added to the fragments by the signing process are stored on a separate server. First, the unsigned fragments that come in from the Producer are embedded with a "uuid" BMFF Box, which contains only the URL to the C2PA data on the separate server. The fragments are then published to the CDN. These fragments never change and this allows the CDN to work as they would in a non-C2PA scenario.

Now only the C2PA data needs to be updated on the server, which are significantly smaller than the media fragments.

The validation has the added steps of reading the URL in the "uuid" box in the fragments, downloading the data from that URL, replacing the "uuid" box with the downloaded data. Then the fragment can be validated as before.

The big drawback of the approach is the introduction of an additional attack vector, being the extra server.

The technical details and issues with this approach follow in Section 5.3.1.

Optimized Merkle Tree with C2PA Data in DASH/HLS Manifests

This approach is very similar to the previous one with the C2PA data being kept separate from the actual fragments. However, in this approach, the C2PA data is written in to the Manifests of the DASH and HLS protocols, specifically the MPD for DASH and the MediaPlaylist of HLS.

An advantage of this is the fact that in a typical live streaming scenario it is very common that the Manifests are updated and downloaded on a per fragment-basis anyways. This way the C2PA data is essentially received for free, barring some bigger Manifests.

One problem with this approach is adding the data to the Manifests. Initially, I started by forking the respective parsing crates to simply add new data fields to the Manifests, which would have no longer been conform to their respective specifications. I would have needed to do the same for the client as well to read out the new data again and that started to become too complex. An alternative could be to misuse an existing field for this purpose. For example event-signaling could be abused for this purpose, similar to ad-insertion.

The technical details and issues with this approach follow in Section 5.3.1.

Rolling Hash Approach

With the knowledge I gathered from the three previous approaches, I came up with a completely new approach, which attempts to deal with as many of the drawbacks of those approaches.

The notable change is the replacement of the Merkle Tree with a Rolling Hash. This is done through an extension of the BMFF Hash Assertion. This extension holds most notably the Rolling Hash, as well as the Anchor Point. The fragments themselves only contain the Anchor Point.

This new signing function only needs the initialization fragment and the new fragment to be sign. If the new fragment is the first fragment of the live stream, then the C2PA Manifest is embedded into the initialization fragment and the Rolling Hash is the hash of the hash of the fragment. The Anchor Point is empty. All the following fragments will use the Rolling Hash of the previous fragment as their Anchor Point (saved in the fragment and the C2PA Manifest) and create the new Rolling Hash by hashing the concatenation of the Anchor Point + hash of the fragment.

The initialization fragment still needs to be updated with each new fragment, however, each fragment only needs to be published once.

The validation begins as before by validating the initialization fragment and the validation of fragments depends on whether the starting fragment is the first fragment of the live stream or any other fragment.

If it is the very first fragment, validation is very simple and only needs to hash the hash of the fragment and compare it with the Rolling Hash found in the C2PA Manifest of the initialization fragment. The resulting Rolling Hash can then be stored and used as Anchor Point for the next fragment: hash the next fragment and create the Rolling Hash as above through concatenation.

When the starting fragment was not the first fragment in the live stream, validation adds one more step. Extracting the Anchor Point from the fragment itself and comparing it with the Anchor Point found in the C2PA Manifest. From here validation continues as above.

This approach also makes use of the EventStream in the DASH MPD to embed the Anchor Points and Rolling Hash of the C2PA Manifest from the initialization fragment. This way, the client only needs to download the initialization fragment with the contained C2PA Manifest once and then it receives the new Rolling Hash and Anchor Point from the MPD.

This approach fixes most of the problems of the other approaches:

- Rolling Hash doesn't make the C2PA Manifest grow over time
- fragments only need to be published once
- validation doesn't need to download any additional data, uses data from the MPD
- parts of the live stream can no longer be replaced, because that would break the Rolling Hash
- no additional attack vector is added
- it remains specification conform (to DASH and HLS, not C2PA)

However, there still remain some drawbacks:

- the initialization fragment still needs to be updated with every new fragment
- fragments can only be validated if the full Rolling Hash up to the newest fragment is created

4.3 Distributor

The signed live stream is hosted on a CDN. A CDN is an HTTP server, which can receive data and then distribute it using HTTP requests. The CDN receives the live stream from the Signer and then distributes it to the Consumer.

The CDN caches all new files in memory to ensure that they can be provided as fast as possible. The files are cleared from the cache when FFmpeg signals their removal. The technical details follow in Section 5.3.5.

4.3.1 Command Line Interface

The CLI of the CDN is kept rudimentary, since it doesn't require much configuration.

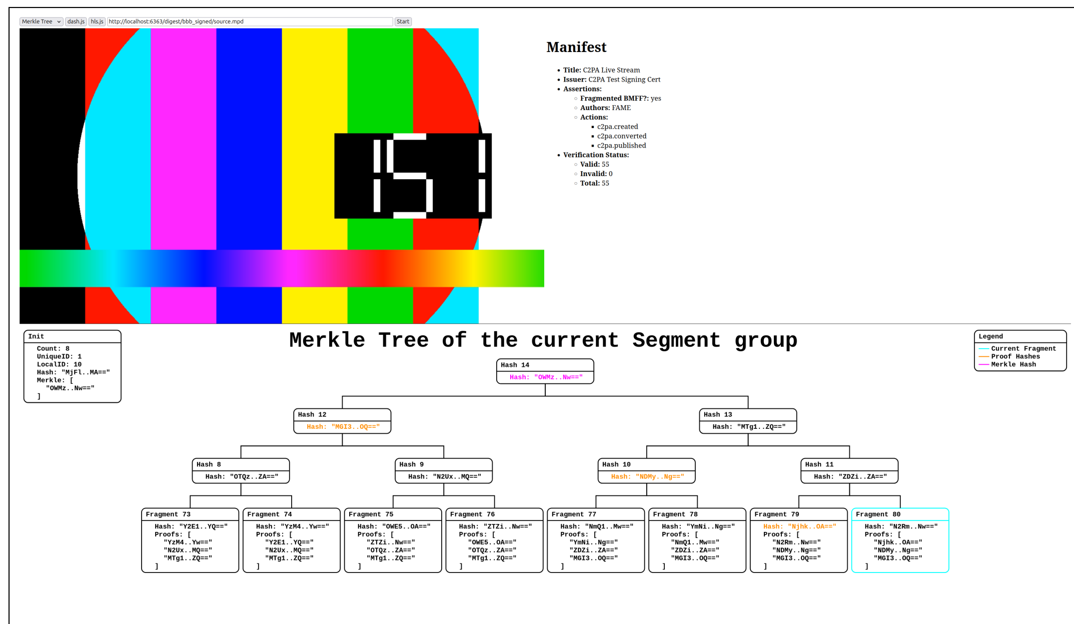


Figure 4.3: Example Merkle Tree Visualization

Short Flag	Long Flag	Value	Description	Default
-b	–bind	Socket Address	The Socket Address on which the CDN will listen on	[::]:6363
-m	–media	Path	The Path to the Directory where to write Data to	./media

Table 4.8: CDN CLI Options

4.4 Consumer

The Consumer is a website built using the SvelteKit [2] framework. The website consists of four major parts: Controls, Player, Manifest and the Visualization of either the Merkle Tree or the Rolling Hash. Figure 4.3 shows an example of the visualization of a full Merkle Tree and a similar example of a Rolling Hash is shown in Figure 4.4. In the appendix there is also an example image of an incomplete Merkle Tree, see Figure .1.

The **Controls** can be separated into two parts.

The first being two buttons and a drop down selector for convenience. The buttons are pre-configured to play the signed live stream generated in this testbed using either `dash.js` or `hls.js`. The drop down selector allows the choosing of which of the live streams to watch: the unsigned original, the Merkle Tree approach or the Rolling Hash method.

[2]Svelte Homepage: `https://svelte.dev/`

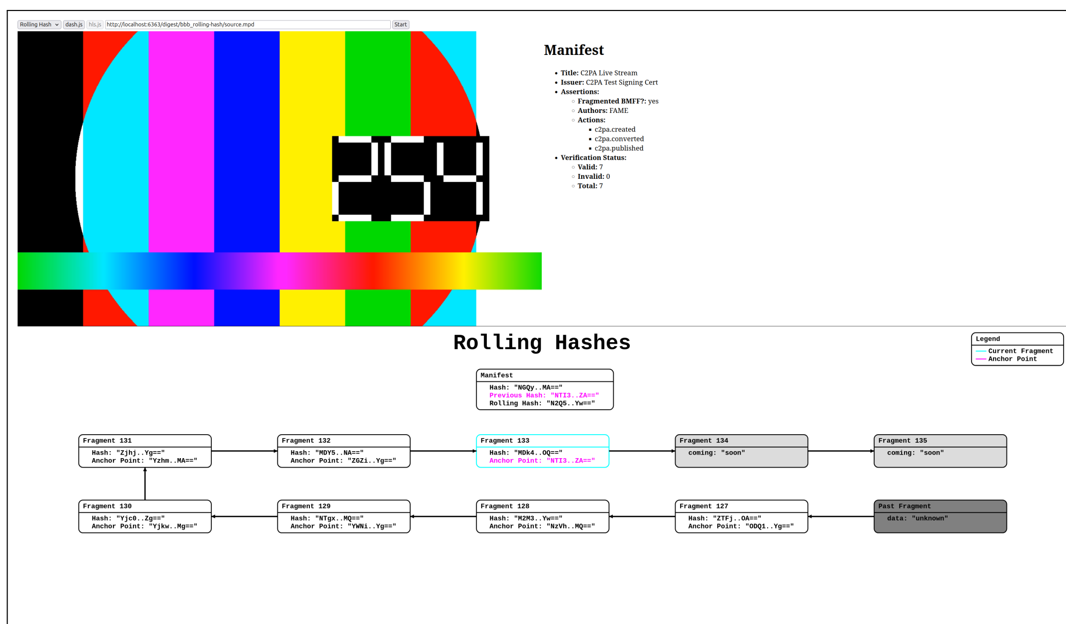


Figure 4.4: Example Rolling Hash Visualization

The second part is an URL text input to use any other live stream and a button to start the live stream using that URL. In this case the website will determine which player to use based on the given URL, specifically using the file extension:

- "mpd" for `dash.js`
- "m3u8" for `hls.js`

The next component is the **Player** and this is a standard HTML5 video element. This video element is used by the players to play the live stream. Both players have also been extended using their public event callback APIs to validate the requested media fragments using the aforementioned `c2pa` JavaScript package. For the Rolling Hash validation, the `c2pa` has been extended with an API to verify those fragments. Details on this implementation follow in Section 5.3.2.

The **Manifest** visualization is third component and is used to display a few selected parts of the C2PA manifest found in the live stream being played. Shown information are the title of the manifest, the issuer of the certificate used to sign the manifest and the assertions: whether it is a fragmented BMFF format, the author of the manifest and all actions applied to the live stream. Finally, I have also used this space to display the current validation status of all fragments by showing the number of valid, invalid and total fragments. In case a fragment is determined to be invalid it will also display the reason why the fragment was invalid.

The final component is the **Visualization** of the Merkle Tree and Rolling Hash and it dynamically creates SVG images of the current state of the Merkle Tree in form of a binary

tree or Rolling Hash as snaking chain of individual hashes. It shows the initialization fragment, the Merkle Tree or Rolling Hash chain and a legend for the colors used. The initialization fragment display contains information of the Merkle Tree referenced by the most recent fragment: the current number of leaves, the unique and local ID, its hash and the reference Merkle Tree hash. The Merkle Tree display shows the reconstructed Merkle Tree based on the current fragment In case of the Rolling Hash the initialization fragment displays its hash and the Rolling Hash and Anchor Point. The leaves (the fragments) list their own data hash and the proof hashes needed for validation. The current fragment is marked by a cyan border. Its proof hashes are colored in orange, in the fragment itself and also in the corresponding node in Merkle Tree. The reference Merkle Tree hashes from the initialization fragment, in this testbed the root node, are colored in magenta. Finally, these colors and their meaning are documented in the legend.

5 Implementation

This chapter details the testbed implementation. It provides insight into the code of the Producer, Signer, Distributor and Consumer components, as well as the challenges I came across.

5.1 Environment

The following hardware and software has been used during the development process:

	Work Laptop 1	**Work Laptop 2**	**Private Laptop**
Vendor	Lenovo ThinkPad L14 Gen 4	Lenovo ThinkPad P1 Gen 7	Framework Laptop 13
Operating System	Ubuntu 24.04.1 LTS	Ubuntu 24.04 LTS	Ubuntu 24.04.2 LTS
CPU	Intel i7 1355U	Intel Core Ultra 9 185H	Intel i5 1340p
GPU	integrated	NVIDIA RTX 4070 Mobile	integrated
RAM	32 GB	64 GB	32 GB
Browser	Google Chrome 119 / Mozilla Firefox 138		
Code Editor	Visual Studio Code		
Rust	Version 1.86.0		
Node.js	Version 22.15.0		
Svelte	Version 4.2.7		
FFmpeg	Version 6.1.1-3ubuntu5		

Table 5.1: Hardware and Software Environment

5.2 Project Structure

This project is distributed across seven subdirectories:

cdn

The **cdn** directory is a standard Rust cargo application project and contains the distributor CDN server, which hosts the live streams created in this testbed.

consumer

The **consumer** directory is a default SvelteKit application project. It holds the client website that plays the created live streams, validates their C2PA data and visualizes it.

media

The **media** directory is the default local disk output local of the live streams created in this testbed.

scripts

The **scripts** directory contains helper shell scripts to effortlessly run the testbed.

c2pa-js

The `c2pa-js` directory is a git submodule and contains the fork of the original `c2pa-js` library containing the client-side validation code.

c2pa-rs

The **c2pa-rs** directory is a git submodule of the fork of the original `c2pa-rs` library. This is the Rust implementation of the C2PA specifications. It also contains the code of the `c2patool`.

producer

The **producer** directory is a git submodule of my FFmpeg script generator.

5.3 Important Implementation Aspects

Over the course of the implementation of this testbed I came across a number of challenges and noteworthy aspects. This section will go over these.

5.3.1 Signing Optimizations

The primary task of this thesis was the optimization of the signing process to be performant enough to be integrated into a live streaming application. The first approach was the optimization of the existing Merkle Tree method. During that implementation I came up with two mutations of that approach which aimed to deal with some of the shortcomings I noticed during development. Both of these kept the actual C2PA data in the "uuid" BMFF boxes separate from the fragment files. The first put this data on a separate server, while the second wrote the C2PA data into the DASH and HLS manifests.

A secondary consideration regarding the optimization of the signing process is the reduction of uploading redundant data to the CDN. This does not only reduces the

overhead and strain on network bandwidth but also allows the CDN to efficiently cache the media fragments. For example the original signing method would have required to publish the entire live stream anew for every created fragment, even the optimized Merkle Tree method still has to publish every fragment up to eight times (or which ever window size is used).

Merkle Tree Signing

The first and most important task of this thesis was the modification of the C2PA signing process to tailor it to the live streaming procedure.

The big problem of the existing signing implementation was that it was designed to be used for signing VoD content, meaning the content that is to be signed is a finalized media stream. In contrast, a live stream is periodically adding new fragments to the stream. As previously outlined, using this implementation as is would require signing the complete live stream with every newly generated fragment.

To provide the option to sign a live stream I added a public method to the C2PA `Builder` structure, which has the same signature as the function to sign fragmented BMFF content, see Listing 5.1, with one addition: a parameters to configure the number of leaves of the individual Merkle Tree. The new function signature is shown in Listing 5.2.

The `signer` is part of all signing functions and is responsible for various tasks related to the digital signature. The `asset_path` is the path to the initialization fragment file. The `fragment_paths` is a list of paths pointing to all fragments that are part of the live stream set, which need to be included in the C2PA Manifest. This list grows by one element with every new fragment created. The `output_path` is the path to the initialization fragment output file. This is where the signed initialization file and all fragments will be saved to during the signing process. The `window_size` is the aforementioned number of leaves of each Merkle Tree.

```
pub fn sign_fragmented_files<P: AsRef<std::path::Path>>(
    &mut self,
     signer: &dyn crate::Signer,
     assert_path: P,
     fragment_paths: &Vec<std::path::PathBuf>,
     output_path: P,
) -> crate::error::Result<()> { ... }
```

Listing 5.1: Original Signing Function

```
pub fn sign_live_bmff<P: AsRef<std::path::Path>>(
    &mut self,
    signer: &dyn crate::Signer,
    assert_path: P,
    fragment_paths: &Vec<std::path::PathBuf>,
    output_path: P,
    window_size: usize, // number of leaves
) -> crate::error::Result<()> { ... }
```

Listing 5.2: New Live Signing Function

The first change I had to make in this function was to ensure it doesn't return an error when the given `output_path` already exists by simply removing the part of the code that resulted in this error. The output will always be populated after the first time this function is called, because the signing builds on top of the previous results rather than starting all over again with every new fragment. Another minor addition was the inclusion of the file extension "m4s" to be part of the BMFF C2PA signing. This file type is typically used in the MPEG-DASH protocol.

The remaining signing process is integrated into the existing process using `window_size` to differentiate between new and old.

In the `c2pa::Store::start_save_bmff_fragmented` method are the next changes. Originally in this function a new empty Merkle Tree was initialized. However, it is imperative to re-use the Merkle Trees from the already signed live stream. This is done by attempting to use the C2PA `Reader` to parse the output file (the signed initialization fragment) to extract the C2PA Manifest. From the manifest I try to find the assertion with the label `"c2pa.hash.bmff.v2"`, which contains the Merkle Tree and is implemented as the `BmffHash` Rust structure. When any of these steps fail, which should only be the case the first time a live stream is signed (when there is no output yet), then it will default to the original behavior of initializing a new Merkle Tree.

With the Merkle Tree now accessible the next step is to update it with the newly added fragment using the `BmffHash::add_merkle_for_fragmented` method. This method expects the asset path, the output path, the fragment paths to add and the unique and local ID. When the configured `window_size` was `0` then these parameters are as they worked previously. Otherwise the paths to all fragments of this live stream are split into groups with each group being `window_size` big, the final group can be smaller. The index of each group will be their local ID of this Merkle Tree (unique ID remains `1` as before) and the final group is part of this Merkle Tree which needs to be updated. All previous groups are assumed to be already properly signed and are left untouched.

The Merkle Tree updating is the next big change. In the original code the number of proofs was hard-coded to be always `4`. I have updated this to be always the number which corresponds to the reference Merkle Tree layer to be the root of the Merkle Tree:

$$\#proofs = \lceil \log_2(\#leaves) \rceil \tag{5.1}$$

Then I updated the part where the fragment paths are converted to `PathBuf` types

and copied to the destination. If the output destination already exists then I use that path from here on instead and skip the copying, otherwise I keep the original path and copy the files to the output (this happens only for the newest fragment). Similarly, I will only copy over the asset file when it doesn't already exist at the destination.

During the placeholder Merkle Tree creation I have removed the restriction that this process can only work for asset files which don't already have an embedded C2PA manifest and instead of always inserting the placeholder, I added the branch to replace the C2PA `uuid` box in files with an existing C2PA manifest. At the end of this function the Merkle Tree originally would have been set on the assertion as a single element of the Merkle Tree list. However, this is now only done when the assertion didn't already have a Merkle Tree list and when it did have one I check if the new Merkle Tree exists on that list. I do this by iterating over all list elements and check if the local and unique IDs match. When I find a match I replace that entry with the new Merkle Tree. If no match is found, this Merkle Tree is new and simply needs to be appended to the list.

The final modification is the process of updating the data hash of the initialization file. This happens in the function `BmffHash::update_fragmented_inithash` and the change is that now it updates this hash on all the Merkle Trees, instead of just the first one.

C2PA Data in DASH / HLS Manifests

This approach leaves the media fragments themselves untouched and forwards them directly to the CDN. However, the DASH and HLS manifests receive modifications before being sent to the CDN. The modifications happened after the signing process. The C2PA data is extracted from the fragments and then written into the manifests.

My initial idea to realize this was to add a new proprietary data field into both manifests, which would hold the C2PA data. To do this, I forked the repositories of both Rust implementations: `dash-mpd-rs` [1] and `m3u8-rs` [2]. Additionally, I have changed the FFmpeg script generator to make it create a script that is creating a live streaming with a fragment timeline instead of a template to have explicit references to the specific fragments, which made the following steps considerably easier.

In the case of the DASH MPD I have then added a "c2pa" data field of type `String` to the data structures `Initialization` and `S` (Segment). Since this implementation uses the `serde` crate for serialization of the data, this was all I had to do, because `serde` automatically handles the serializing and deserializing of Rust data structures. Additionally, on the client `dash.js` seemed to use a generic XML parsing, since the newly added fields showed up without requiring any changes.

For HLS I started by adding, similarly to DASH, a "c2pa" data field of type `String` to the `MediaSegment` and the `Map` (Initialization fragment) data structures. However, this implementation does not use `serde` for de-/serialization, instead it uses `nom` [3], which is crate that can be used to build arbitrary data parsers. This required manually adding the parsing of these new fields into the process. Furthermore, the `hls.js` player has

[1] GitHub Repository: `https://github.com/emarsden/dash-mpd-rs`
[2] GitHub Repository: `https://github.com/rutgersc/m3u8-rs`
[3] GitHub Repository: `https://github.com/rust-bakery/nom`

similarly implemented their own m3u8 parsing, which would have required adding the same parsing to this code base as well.

However, at this point I had notice that the switch to fragment timeline and the whole manifest manipulations lead to both players refusing the playback of the live stream anyways without any error messages whatsoever. Since debugging these players went beyond the scope of this thesis I abandoned this approach at this point.

This approach would also have not been conform with the respective technical specifications. A better alternative would have been to use event signaling, instead of modifying the manifest structures.

C2PA Data on a separate Server

In contrast to the previous approach, here the manifests stay untouched, which allowed switching back to fragment template and fixing the playback issue, and the fragments themselves are modified. Also in contrast to the original method, the fragments are only modified once, meaning once they are published to the CDN they don't change anymore.

After the fragments are signed the C2PA data is extracted from them and then send to a server, which acts essentially the same as the media CDN, just for the C2PA data instead of media. Then the "uuid" BMFF box of the fragments are replaced with another "uuid" BMFF box. This new box simply contains the URL to its C2PA data. Finally, the fragments are published to the CDN.

On the client side before validation, the URL is extracted from the fragment. This URL is then queried with a HTTP GET request to download the actual C2PA data. This data is then inserted in place of the URL "uuid" BMFF box and then the fragment is ready to be put into the validator as usual.

For simplicity I use the CDN server for this as well.

5.3.2 Extending the Client C2PA Package

The client `c2pa` library is part of of the `c2pa-js` [4] suite of JavaScript packages. This library leverages WebAssembly [5] (WASM) to use the Rust library in a web browser. WASM is browser API that allows running other languages (than JavaScript), like C/C++, C# and Rust, in the browser with near to native performance of the original code. With WASM it is possible to port the `c2pa-rs` code into the browser with very little additional work, otherwise the library would have needed to be completely rewritten in JavaScript.

This repository uses `rush` [6] to manage the packages. The only issue I had with this, was the very peculiar Node.js version requirement [7], see the footnote for specifics. Getting around this requirement was very simple using the Node Version Manager `nvm`. I chose to

[4] GitHub Repository: `https://github.com/contentauth/c2pa-js`
[5] WASM mdn web docs: `https://developer.mozilla.org/en-US/docs/WebAssembly`
[6] rush Homepage: `https://rushjs.io/`
[7] Rush Node.js version specification: `https://github.com/ArckyPN/c2pa-js/blob/main/rush.json#L127`

use the highest allowed version supported by this configuration, version 18. Every time I needed to do anything that involved `rush`, namely building the packages to test the changes I made and also committing the changes to the git repository, I had to switch node version 18. Using `nvm` this is as simple as running `nvm install 18` or `nvm use 18` once the version has been installed.

The first step was replacing the `c2pa` npm dependency from the official, remotely hosted version to the version I have forked. This was a very simple task, as it only required a simple change in the `package.json`: changing the associated value of the `"c2pa"` entry of the `"dependencies"` object from the original version denotation `"0.18.0-fmp4-alpha.1"` to the path to the local clone of the forked repository `"file:../c2pa-js/packages/c2pa"`.

Since I forked the most recent edition of the repository and not the branch that contains the alternative version, which includes the fragmented BMFF validation, the next step was bringing this validation into the version. There was a very big version difference between the two versions and in the meantime the `c2pa-rs` Rust library had moved to the v2 API and deprecated the original API, which the fragmented BMFF version happened to still be using. This required to firstly update the Rust part of this library to the new API. After that I could simply copy the remaining code changes from the alternative version into the forked version. This proved to be a very good approach, because while going through the changes, I learnt how the WASM integration works and by the time I brought the fragmented BMFF validation back, I knew what I needed to do to add the Rolling Hash validation.

The first half of change had to be done in TypeScript and began with adding the Rolling Hash validation as a new function definition to the `C2PA` interface:

```
export interface C2pa {
    ...
    readRollingHash(
        fragment: ArrayBuffer,
        rollingHash: Uint8Array,
        anchorPoint?: Uint8Array,
        options?: C2paReadOptions,
    ): Promise<Uint8Array>;
    ...
}
```

Listing 5.3: New Rolling Hash Validation Function

This function validates a fragment, given as data buffer, using the Rolling Hash and an optional Anchor Point. The Rolling Hash is the result expected by hashing the fragment and the Anchor Point. This hash is retrieved from the DASH MPD or the C2PA Manifest from the initialization fragment. The Anchor Point is optional, because the very first fragment doesn't have one and when joining the live stream not from the very beginning, the Anchor Point may not be known. When no Anchor Point is given, the validator will attempt to read the Anchor Point from the "uuid" BMFF Box of the fragment. Finally, on success this function will return the Rolling Hash result, this hash will be the

Anchor Point of the next fragment validation. When the validation was not successful, this function will throw an error, necessitating wrapping this function in a `try/catch` block.

The next step was actually implementing the defined function by adding it to the `createC2pa` function. This function calls a newly added Worker Pool function specific to Rolling Hash validation, which in-turn also required a new function on Worker. This final Worker function is calling the actual function that is executing the WASM Rust code. All these changes were very straightforward and analogue to the other already existing validation functions.

The second half the changes needed to be done in Rust code. They started out with adding the TypeScript function prototypes to the provided string definition. Then I could implement the function that gets bound by WASM to be callable from JavaScript. This was again essentially the same for the other functions with the only difference being calling a Rolling Hash specific validation function. This function calls the actual `c2pa-rs` validation function I implemented for Rolling Hash validation.

The final step after the implementation was building the changes into usable packages using `rush` by running `rush build` in a terminal. Once this command had run its course, the code changes were ready to use.

5.3.3 Styling the Video Element

The people behind the `c2pa-rs` code also have a fork of the `dash.js` video player [8], where they have created a video player combining `dash.js` and `video.js` [9] that is able to validate the C2PA Manifests of the video playing and show the status of the validation. `video.js` is a JavaScript library that enables customizing a HTML video element.

This video player shows the validation status of each media fragment on the seekbar of the video element. Valid sections are shown in blue, invalid in red and unknown section are colored in gray. An example of this seekbar is shown in Figure 5.1. For presentation purposes the stream contains two invalid fragments. The same image also shows the second addition: a Content Credentials icon in the bottom left corner. This icon shows the overall validation status of the entire stream. It also doubles as a clickable button to open an overlay over the video to display a selection of metadata from the C2PA Manifest, like the issuer of the signing certificate, the tool or device used to create the Manifest, the name of the signer and their website and social media links, the validation status and possibly more. This overlay is shown for a fully valid stream in Figure 5.2 and for a stream with a few invalid sections in Figure 5.3.

I have adapted the sample code from the above linked GitHub repository into this testbed. I kept everything in the same visual display style. However, I have modified the code itself into TypeScript and my style of coding to make it fit into the website framework I am using. During this implementation I came across three major challenges.

The first problem was the proper integration of `video.js` into my Svelte website. Since

[8] C2PA dash.js Player: `https://github.com/contentauth/dash.js/tree/c2pa-dash/samples/c2pa`
[9] video.js Homepage: `https://videojs.com/`

the documentation of `video.js` is not that good, in my opinion, it took me a while to understand that it is imperative to include the library provided CSS file and give the HTML video element the class `video-js`.

The next issue was the filling of the overlay menu with the information from the C2PA Manifest. The sample code is using messy string manipulation to set the `innerHTML` property of the menu's list elements to build this menu. However, doing it this way didn't not trigger re-renders in the Svelte framework and the menu always remain in the initial state. I overcame this problem by creating the required HTML elements using the document JavaScript API, namely `document.createElement(...)`. Then set all the required parameters of those elements and then explicitly inserting the elements into the document. The sample code also re-created the menu with every timeline update, which was no big problem when they constantly overwrite the inner HTML of the menu elements. My approach, however, required to build this menu once for the first time a menu element has a value and then only update the value when it changes.

Finally, I had the issue that the seekbar itself of the timeline would not show up when the live stream starts. Sometimes it showed later into the running stream with wildly different delays and sometimes it would never show up. When using a VoD stream the seekbar showed up right away. Eventually, I came across two GitHub issues regarding this dilemma [10] [11] on the `video.js` repository. The problem was that for live streams `video.js` usually shows no seekbar at all, instead it shows an indicator that a live stream is being watched. However, there is an options for the initial setup to enable a live UI which would show a seekbar. I was using this options from the start. The developers of `video.js` are of the opinion, though, that it makes little sense to show a seekbar when only very little of the live stream has been loaded into the browser and have configured the live UI to only show the seekbar once at least 30 seconds of media has been loaded. Fortunately, it is possible to configure this parameter and I have set the value to zero seconds, which makes the seekbar appear right away.

5.3.4 Reading the `c2pa.hash.bmff.v2` Assertion

Reading the `"c2pa.hash.bmff.v2"` assertion proved to be a spiteful challenge. Initially, when I was done with the implementation and wanted to test it out, I would always get the result that none of the fragments would correctly validate with trustworthy C2PA manifests.

The culprit ended up being the definition of the `BmffHash` Rust structure. It uses the popular Rust crate `serde` (from **Ser**ialize and **De**serialize) to define how the structure is converted to and from binary data and in this definition the data field signifying the version is being ignored from the conversion. This resulted that when deserializing this assertion from the previously signed output the version would always be left uninitialized on the structure. This in-turn had the effect that the required version 2 would not be used for all the data hashing, resulting in the erroneous validations.

[10] `video.js` Issue #7095: `https://github.com/videojs/video.js/issues/7095`
[11] `video.js` Issue #6630: `https://github.com/videojs/video.js/issues/6630`

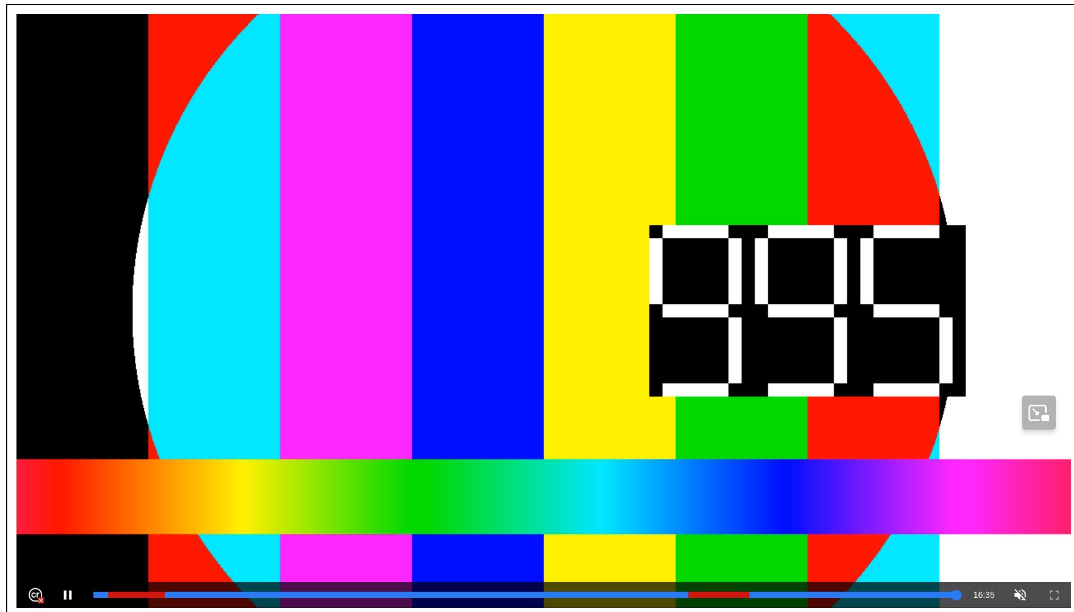


Figure 5.1: Seekbar Video Styling

Figure 5.2: Video Content Credentials Overlay

Figure 5.3: Video Content Credentials Overlay With Errors

I fixed this problem by making the existing method `BmffHash::set_bmff_version` public and manually setting the version to `2` after the assertion had been extracted.

5.3.5 CDN Caching

The Distributor is simple HTTP server built with the `Rocket`[12] Rust crate. The media distribution is handled by two endpoints: `"/ingest/uri..."` and `"/digest/uri..."`. The ingest endpoint is used by the Signer to post the live stream after signing. The digest endpoint is used by the media players on the Consumer to get the published live stream.

To keep the forwarding delay to a minimum the most recent fragments are kept in memory of the Distributor as part of a proxy cache. This cache consists of two components a `DashMap` and a `ReplayChannel`. The `DashMap` is provided by the `dashmap`[13] Rust crate and is essentially a standard Rust `HashMap` that has been highly optimized for concurrency. The `ReplayChannel` is taken from the `replay-channel`[14] Rust crate and works just like normal broadcast data channels in Rust with the added caveat that subscribers to that channel will always receive all data starting at the beginning of the channel. With this cache the CDN is able to write fragments to the cache, identified by their URIs, as they arrive. When the Consumer requests a new fragment, the CDN can check if the requested URI is cached. If it is cached I can subscribe to the corresponding

[12]Rocket GitHub: `https://github.com/rwf2/Rocket`
[13]dashmap GitHub: `https://github.com/xacrimon/dashmap`
[14]replay-channel GitHub: `https://github.com/freenet/replay-channel`

channel and send the data as response. When a fragment wasn't cached the CDN will instead read the fragment from local storage.

I make use of the `FFmpeg` arguments `-window_size <num>` and `-extra_window_size <num>` to prevent the cache from caching unnecessarily many fragments and stop it from growing too large. The window size argument denotes the number of fragments kept in the DASH manifest and HLS media playlist. The extra window size option defines the number of fragments in addition to window size that are kept saved on the disk. Since I am using an HTTP output, instead of directly deleting the fragments, FFmpeg will instead send an HTTP DELETE request for the corresponding fragment. The Signer forwards these delete requests to the CDN where the denoted fragment is removed from the cache. Additionally, the CDN should not directly delete the fragment from the cache, instead it should save the data to the attached storage. However, since I configured the Signer and CDN to use the same local directly as storage, I am skipping this part in the CDN to prevent unnecessary redundant IO operations.

5.3.6 Producer Inputs

The Producer is capable of accepting a variety of input types. The input can be configured using the CLI arguments as shown in Section 4.1.1.

Remote

Remote inputs can be anything that is addressable using an IP-compliant address. Examples for this are URLs pointing to server hosted DASH MPDs or HLS Playlists which denote VoDs or live streaming, IP-cameras like the Android app DroidCam [15], which turns your smartphone into a camera or anything else that is supported as input by `FFmpeg`.

Local

Video files are the local input type. They can be either used as a static input to created a limited time stream or they can be used a infinite loop to create a endless live stream.

Standard Input

This input option expects raw images on the standard input pipe, which turn these images into a video stream. In this use case `FFmpeg` expects a definition of the image resolution to properly interpret the received data. This type was needed by a different project where the images were created by the Unity Game Engine[16] and then used to create a live stream for a remote rendering scenario.

[15] DroidCam Homepage: `https://droidcam.app/`
[16] Unity Homepage: `https://unity.com/`

Device

On Linux video devices are addressed using a path, for example contents of the `/dev/` directory, and they can also be used as an input. For example on a laptop with an integrated webcam that webcam is usually the device `/dev/video0`.

When using this input I had to use the tool `video4linux2` to extract metadata about the device, like supported formats, resolutions and framerates which are required by `FFmpeg` to properly parse the input data.

External Camera

This method uses the same **Device** input as detailed in the previous section but since getting this to work required more work, I am giving it its own section.

For this thesis I was provided with a Sony Alpha 7 III, so these steps apply only to this specific camera (every vendor likely has different implementations of getting this working).

Normally, Sony provides the software tool Imaging Edge Webcam [17], which turns the camera into a webcam when connected to a PC. Unfortunately, this program is only supported on Windows and macOS operating systems, as described on the download page, and I am using a Linux operating system. However, I was able to find a Reddit thread, which had a comment detailing a workaround to get an alternative working [18].

This approach uses the tool `gphoto2` [19], an open source application for digital cameras and media players on Unix-like systems. The first step is installing the required tools `gphoto2`, `video4linux2 loopback-utils` and `FFmpeg` by running `sudo apt install gphoto2 v4l2loopback-utils ffmpeg`, followed by plugging in the camera using an USB cable. If the file system automatically mounts the camera, it is important to unmount it. It is important to then select the option to use the camera as capture device when prompted to on the camera's display, instead of using it to transfer files to the connected device. The next step is creating a new video device using the video feed from the camera. Before the device can be created it is important to be able to determine which device is being created by this process: first ensure no other loopback service is running by stopping the service using the command `sudo rmmod v4l2loopback`. Next, determine the currently available video devices by remembering the list given by `ls /dev/video*`. Now start the loopback service using the camera's video feed with the command `sudo modprobe v4l2loopback exclusive_caps=1 max_buffers=2` to create the new video device. The designation of this device can be determined by listing the available video device again (with the same command as before). Whatever device that is now on this list that wasn't there before is the device that was created by this process. Whether this process has worked can be verified by running `gphoto2 -auto-detect`. If the process was successful, the terminal should now display the camera model name and the port it is con-

[17] Imaging Edge Webcam Product Page: `https://support.d-imaging.sony.co.jp/app/webcam/en/`
[18] Reddit - Sony Imaging Edge Webcam in Ubuntu: `https://www.reddit.com/r/Ubuntu/comments/s4zjo7/comment/kwsoxzm/`
[19] gPhoto Homepage: `http://gphoto.org/`

nected to. Finally, the last step is actually piping the video feed from the camera to the video device by running the final command `photo2 -stdout -capture-movie | ffmpeg -i - -vcodec rawvideo -pix_fmt yuv420p -threads 0 -f v4l2 <video device>`.

Once this command is running the camera feed is now accessible from the specified video device.

I have automated this process in the helper shell script. I split this process up into two additional shell scripts in the Producer repository: setup and run-capture. The setup encompasses everything after the dependency installation up until the final piping command. It stops the possibly running loopback services, reads the currently running video devices and keeps them in memory, then it starts the loopback service, reads the available devices again, then it filters out all video devices that are new in the second reading. Finally, when multiple new devices have been found, it prompts the use to choose which device to use. Otherwise, it will automatically use the one device found. The determined video device path will be printed to the standard output. The run-capture script expects the video capture device path argument and will simply run the final piping command.

The automation starts by calling the setup script and saving the returned video device to a variable. Then it starts the run-capture script in a separate terminal using `gnome-terminal - ./run-capture $VIDEO_DEVICE`. Finally, I am waiting five seconds to make sure that the video device is saturated.

5.4 Documentation

To effortlessly run this testbed during development, I have created four helper shell scripts. Each of these scripts corresponds to one of the four components of this testbed. They are pre-configured with options, which allow running the testbed without needing specific knowledge of the CLI options of all the components.

However, it is still possible to overwrite some of the default values, specifically only for the Signer and CDN scripts. This can be achieved using environment variables. For example on Linux, if one wants to use port 7000 instead of the default port of the CDN, the script command would like the following: `PORT=7000 ./scripts/cdn`. When setting multiple overwrites the individual environment variable declarations are separated by a whitespace.

5.4.1 Producer

The `./scripts/producer` does not use environment variables to overwrite default values. Instead it uses a similar CLI interface to the actual Rust programs. By default, if no arguments given, the producer will use the test image as input. To configure it, the first argument is a subcommand followed by an optional input value.

All available options are:

- Test:

  - Command: test
  - Input: none
  - Description: use a test image as input (default behavior if no command given)
- Webcam:
  - Command: webcam
  - Input: none
  - Description: use `/dev/video0` as input device
- Device:
  - Command: device
  - Input: capture device path
  - Description: same webcam command, but with a specific capture device
- Local:
  - Command: local
  - Input: path to local video file
  - Description: use the local video file as input
- Remote:
  - Command: remote
  - Input: URL to remote resource
  - Description: use the remote resource as input
- DroidCam:
  - Command: droidcam
  - Input: none
  - Description: use the default address of the DroidCam Android app as input
- External:
  - Command: external
  - Input: none
  - Description: use an external cameras as input (this will run the script described in Section 5.3.6)
- Help:
  - Command: help
  - Input: none

- Description: print a help page describing these exact commands

This script must be launched after both the Signer and CDN are running, otherwise the output will point to a non-existing target and FFmpeg will crash.

5.4.2 Signer

The `./scripts/signer` script will run the `c2patool` in live signing mode. Configuration options are the address of the CDN ingest URL, which should be the address the CDN is using, the output directory where to save the data to, the path to the C2PA Manifest JSON definition file and the window size for the optimize Merkle Tree signing. This script must be running before starting the Producer.

Variable	Value	Description
CDN	URL	CDN ingest URL
OUTPUT	Path	Path to save directory
MANIFEST	Path	Path to C2PA Manifest JSON definition file
WINDOW	usize	Window size for optimized Merkle Tree signing

Table 5.2: Signer Environment Variables

5.4.3 CDN

The `./scripts/cdn` starts the CDN server. It can be configured with a different localhost port or the address can be overwritten entirely (in this case the port variable has no effect) and the output directory can be specified. This script must be running before starting the Producer. The order of running the Signer or CDN first does not matter.

Variable	Value	Description
PORT	u16	Localhost Port on which to listen
BIND	Socket Address	he Socket Address on which the CDN will listen on
MEDIA	Path	The path to the output directory

Table 5.3: CDN Environment Variables

5.4.4 Consumer

The `./scripts/consumer` script is the easiest of all the scripts and does not require any configuration. This script will run the SvelteKit website and open it automatically in the default web browser. The will try to use to port `5173` of localhost and if that is not available, it will automatically increment the port by `1` until it finds an available port.

This script can be run whenever one wants.

6 Evaluation

In this chapter I will evaluate the different C2PA signing methods and present the benchmarks I put these through alongside their results and observations.

6.1 Test Environment

I ran the benchmarks on the Work Laptop 2 shown in Table 5.1. In total I ran two benchmarks to measure the performance of the original Merkle Tree, optimized Merkle Tree and Rolling Hash approaches. The first benchmark measures the time it takes to sign fragments and the second measures the amount of data needed to be uploaded to the CDN.

For these benchmarks I prepared 100 fragments for four different media representations: one audio track and three video tracks with vastly different file sizes. I chose to use only one audio quality, because from experience it is not that common to have multiple audio qualities, if there are multiple representations they are usually different languages with the same encoding. The audio setting I use are the same that are found in the default audio settings of the Producer, shown in Figure 4.1, with an average file size of `31 kilobyte`. The three video tracks represent a low, mid and high encoding quality each. The settings are taken from the default video settings of the Producer, shown in Figure 4.2. The low options is represented by the `240p` setting with an average file size of `64 kilobyte`, the mid option is taken from the `720p` (HD) setting with an average file size of `467 kilobyte` and the high option is the `2160p` (4k) setting with an average file size of `3,752 kilobyte`. All file sizes are shown in Figure 6.1.

6.1.1 Signing Duration

In this benchmark I measured how long the entire signing function runs for to sign a live stream with up to 100 fragments. For this benchmark I loop through all available representations then for each signing method I iterate from 1 to 100 and sign the live stream at each index in accordance to the method. For the Rolling Hash signing this means that only the current fragments at that index is being signed. For the case of the optimized Merkle Tree approach I provide the signing function with the paths too all fragments up to the current index, but only the up to window size (eight) number of fragments are actually signed and for the original Merkle Tree every single fragment is being signed up to the current index.

The lower the time it takes to sign the newly created fragment, the less the impact on the streaming latency is, lower is better.

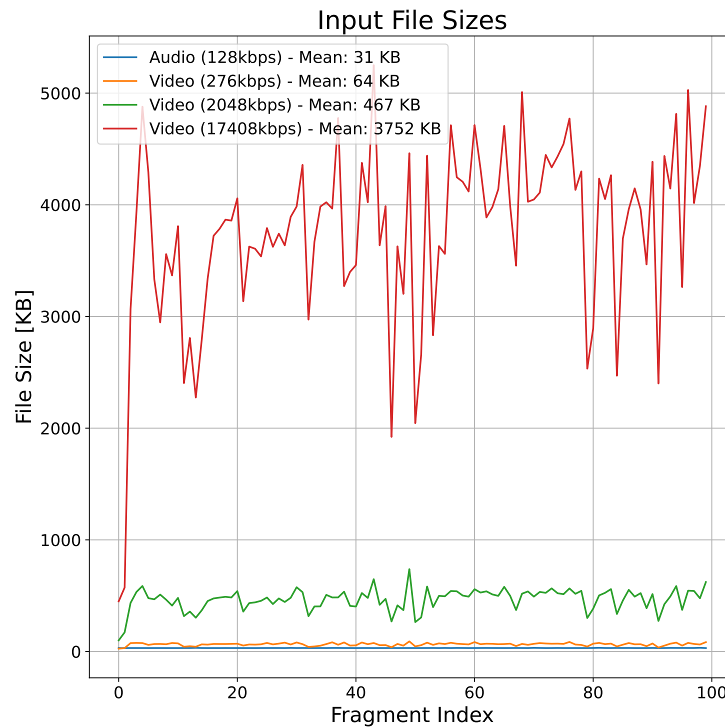


Figure 6.1: Input File Sizes

The time measuring can be described with the pseudo code from Algorithm 6.1. Essentially, I take a timestamp right before signing the fragments and right after the signing I take a second timestamp and the difference between the two timestamps is the time it took to sign the fragments. Then I save this value in a data structure in relation to the fragment index and representation the fragment belongs to.

Algorithm 6.1 Duration Measuring Method

1: $now \leftarrow Time.now()$
2: sign_fragments()
3: $time \leftarrow Time.since(now)$

In total I ran this benchmark ten times and averaged all results.

6.1.2 Data Upload Size Required

For this benchmark I looked at how much data has to be published to the CDN after each fragment has been signed. I measured this by simply adding up the file sizes of all the files

that were affected by the signing of the current fragment, which would subsequently be needed to be uploaded to the CDN. In case of the Rolling Hash approach that is only the initialization fragment and the fragment at the current index. For the optimized Merkle Tree approach that is the initialization fragment and all fragments of the current Merkle Tree, which are up to eight fragments with my configuration and again all fragments for the original Merkle Tree signing. The original Merkle Tree signing requires to upload every single fragmnet up to the current index.

Since this benchmark is merely adding up file sizes I ran this one in conjunction with the previous duration benchmark, so I don't have to sign everything multiple times for each benchmark.

In this benchmark a lower upload size equals to less overhead on the network and a better caching situation for the CDN, again lower is better.

6.2 Performance Measurements

In this section I will present the results of the above described benchmarks and give my evaluation and observations about them.

6.2.1 Signing Duration

I have split the data plots into two comparisons. One comparing each signing method in relation to the different representations, shown in Figure 6.2 and the other comparing the signing method to each other per representation, shown in Figure 6.3.

This signing procedure is the added step into the creation of this live stream which is why I equate the signing duration as the impact on the latency of the live stream.

Since the original Merkle Tree signing method signs every single fragment that has been created over the course of the live stream, I expected the overall signing duration to steadily increase as the live stream gets longer. This expectation is confirmed when looking at Figure 6.2a. The signing duration of all four representations linearly increases in proportion to the number of fragments that are being included in the signing process. It can also be deduced that the larger the input files are the stronger the duration increases.

I will score the performance of the original Merkle Tree signing method by looking at how many fragments can be signed within one second. Since the graphs are consistently linear, I will measure this score by taking the signing duration of the 100th fragment and extrapolating its signing duration to one second.

The audio representation has the lowest signing duration of all, which is expected since it also has the lowest input file sizes with an average of `31 kilobyte`. The score of this representation is `528` fragments, which can be signed within one second. The next fastest signing duration is the low video representations with an average file size of `64 kilobyte`. The score of this representation is `344` fragments. The mid video representation has the second slowest signing duration with average file sizes of `467 kilobyte` and a score of `82` fragment. The slowest signing duration is expectedly measured on the high video representation with average file sizes of `3,752 kilobyte` and a score of only `13` fragments.

The optimized Merkle Tree signing method uses essentially the same concepts of the original equivalent but with the added constraint that eight fragments are grouped into separate Merkle Trees. The expectations of this method is that the signing duration will look very similar for each individual Merkle Tree but reset every eight fragments. The plot in Figure 6.2b confirm this expectation.

The plot perfectly shows the sawtooth-signal-like graph of this approach. It starts with a low signing duration then it increases linearly like the original method and then peaks at every eighth fragment before it resets back down to the low duration. The important metric of this approach are the extreme signing durations, by which I will score the individual representations.

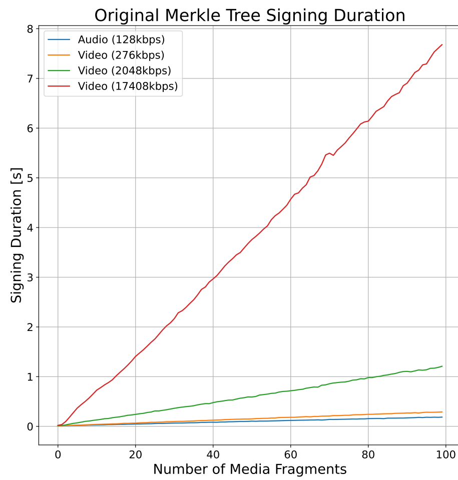
The order slowest to fastest is the same as before. The audio representation is the fastest with a minimum signing duration around `8` milliseconds and a peak of `25` milliseconds. Again followed by the low video representation with a minimum duration of about `8` milliseconds and a peak of roughly `35` milliseconds. The next slowest is the mid video representation with the low point signing duration being on average `10` milliseconds and a maximum of around `100` milliseconds. The slowest is again the high video representation with minimum signing duration about `100` milliseconds and peaks up to `700` milliseconds.

Looking at the plot of the Rolling Hash signing method in Figure 6.2c is clear that this is fast method overall by a good margin. Since this approach only every signs the current fragment, the signing duration directly correlates to the input file size. The three lowest representation have near constant signing duration. The audio presentation has an average of `10.5` milliseconds. The video low and mid video representations have average signing duration of `11.8` and `21.3` milliseconds, respectively. The high video representation fluctuates a lot more than the other representations but the majority of the time it remains in roughly the same interval between `80` and `100` milliseconds with an overall average of `86.5` milliseconds. When looking at the plots of Figure 6.2c and Figure 6.1 side by side, it is very obvious that these results are expected. The files sizes for the lower representations line up the with stability of the signing duration and the fluctuation of the higher representations equally lines up with the signing durations.

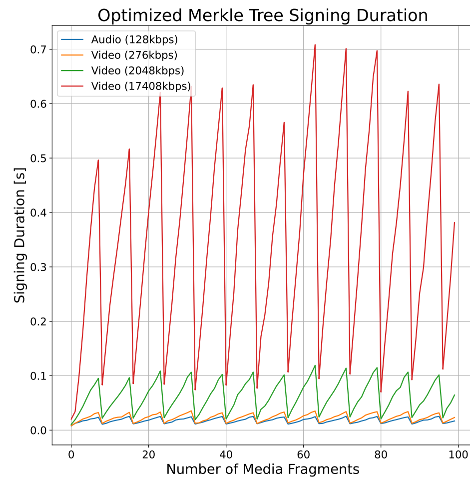
Now looking at the comparison of the signing methods to each other in Figure 6.3, it gets directly apparent that the signing methods all perform essentially the same in regards to what representation is being signed. However, these plots show very well that the original signing method is by a large margin the slowest signing method, with the optimized Merkle Tree signing method being much more competitive with the Rolling Hash method but the Rolling Hash method is still the best.

6.2.2 Data Upload Size Required

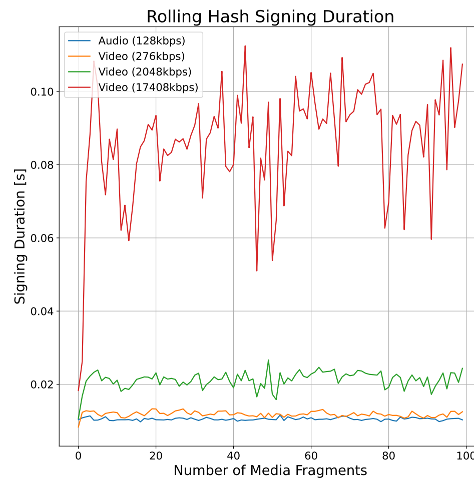
For this benchmark I have split the data plots into the same comparisons as before: comparing each signing method in relation to the different representations are shown in Figure 6.4 and the signing methods compared to each other per representation are shown in Figure 6.5.



(a) Original Merkle Tree

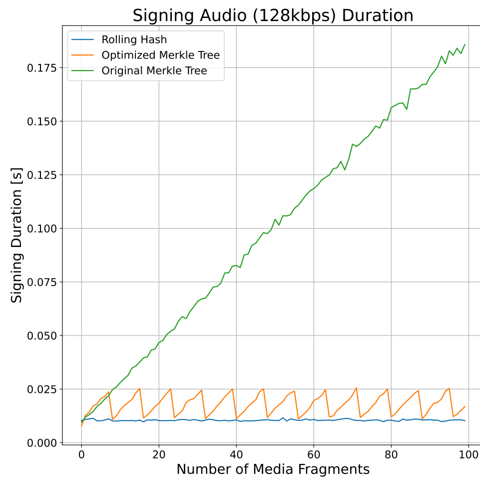


(b) Optimized Merkle Tree

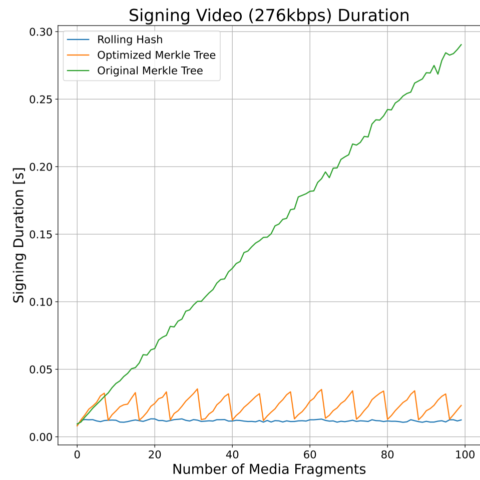


(c) Rolling Hash

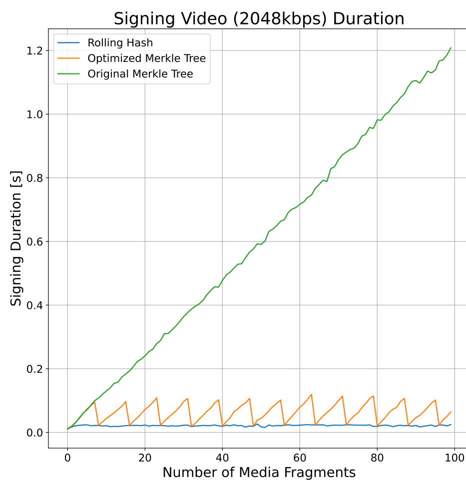
Figure 6.2: Signing Duration Results



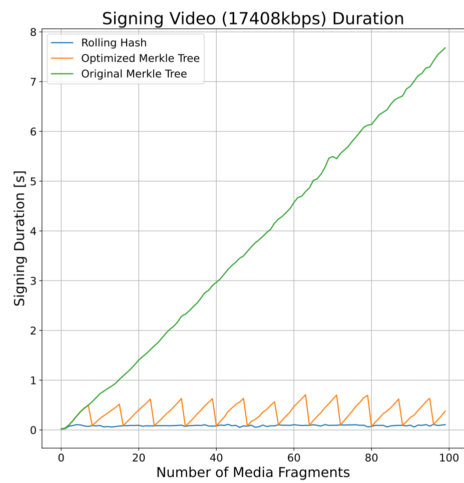
(a) Audio (128kbps)



(b) Video (276kbps)



(c) Video (2,048kbps)



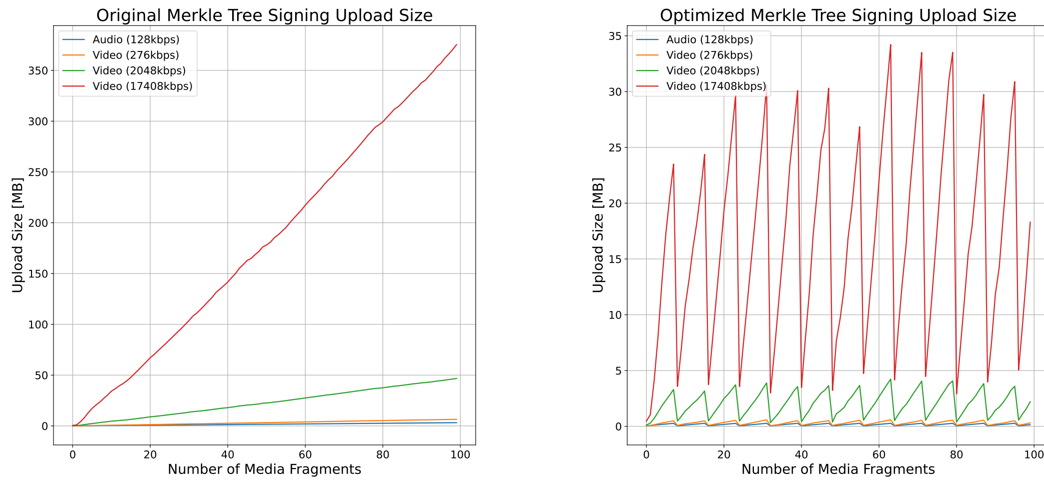
(d) Video (17,408kbps)

Figure 6.3: Signing Duration Results

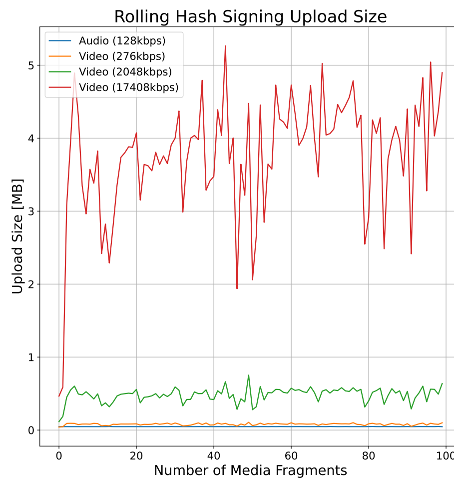
In essence, the results of the required data upload size is exactly the same as the signing duration benchmark. The original Merkle Tree method requires by far the most data upload, of which the majority is entirely redundant, because every time a new fragment is being signed, every single prior fragment is being signed as well and subsequently they have to be published anew to the CDN to have the most up-to-date data available. As before it also shows a linear increase as the number of fragment rises. All the representations have an gradient of slightly more then the average file size per fragment, as shown in Figure 6.1. The reason by this is that with every new fragment everything from before has to be uploaded again, as already mentioned and the gradient is a little more because the added C2PA data is now included in the files. These results are shown in Figure 6.4a.

The same goes for the optimized Merkle Tree signing approach, shown in Figure 6.4b. Again a sawtooth-signal-like graph can be observed which resets every eighth fragment. In all cases the needed upload is considerably lower than compared to the original method.

Lastly, the Rolling Hash results are equally the same, as seen in Figure 6.4c. The three lower representations again have a near constant value and the highest fluctuates slightly but overall still significantly lower than the other two methods.

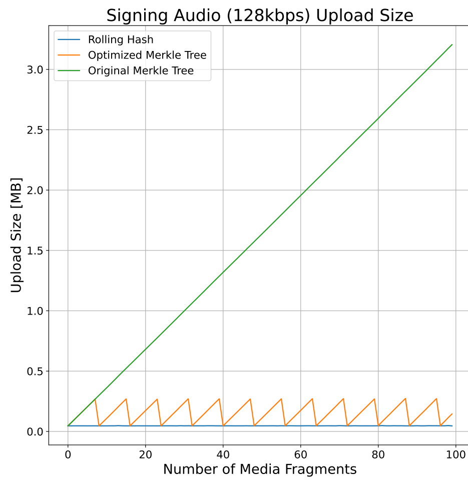


(a) Original Merkle Tree (b) Optimized Merkle Tree

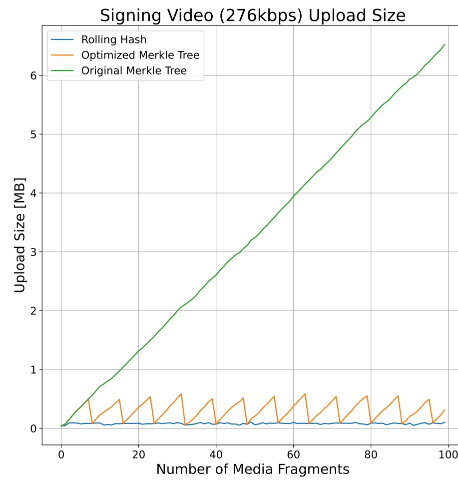


(c) Rolling Hash

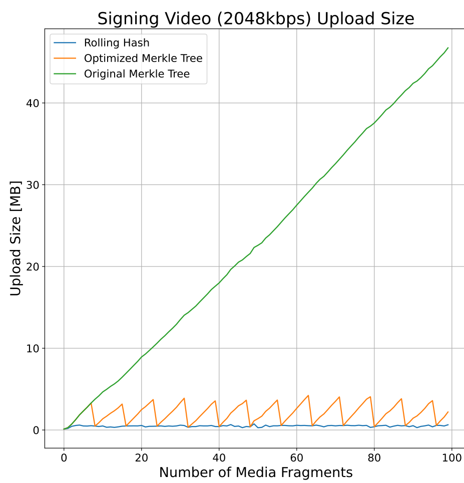
Figure 6.4: Upload Size Results



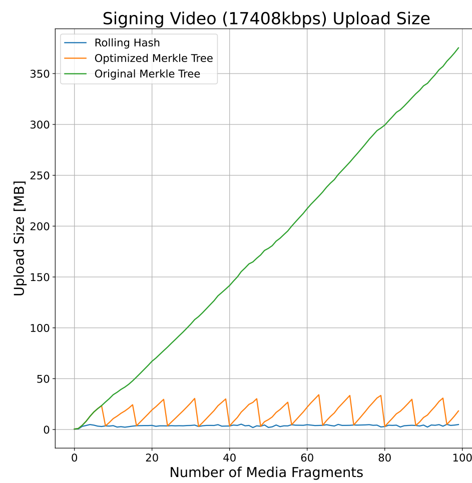
(a) Audio (128kbps)



(b) Video (276kbps)



(c) Video (2,048kbps)



(d) Video (17,408kbps)

Figure 6.5: Upload Size Results

7 Conclusion

I will begin the conclusion to this thesis with a summary of what I have presented in this thesis, followed by a dissemination of C2PA on what real world applications can benefit from it. Finally, I will come to a close with an outlook on C2PA, possibilities how the shown implementation can be expanded upon and further research into this area.

7.1 Summary

I began this thesis with a look into the C2PA technical specifications to get a sense of what C2PA is all about and to understand how to improve the existing fragmented BMFF signing method to make it more suitable for live streaming. In parallel I started with the implementation of the base frame of the testbed made up of the four components: Producer, Signer, Distributor and Consumer. The Producer is a configurable FFmpeg script generator, which is used to create the live stream. The Signer is a modification of the `c2patool` which acts an HTTP server, which receives the live stream from the Producer's FFmpeg command and then signs the live stream as it arrives and then publishes the live stream to the Distributor. The Distributor is a pseudo CDN HTTP server, which receives the live stream from the Signer and then hosts it for consumption. The Consumer is a website that uses the popular `dash.js` and `hls.js` video players to play the signed live stream, requested from the CDN, and also validates whether the received live stream is trustworthy according to the C2PA specifications. It also visualizes the validation status.

Once this basic frame was standing I continued with the optimization of the signing method. First, I tried out the existing implementation in this live streaming context and learnt that it technically works, but the required computational power become too large too fast. My first optimization approach was to use the Merkle Tree concept but make use of the possibility to use multiple Merkle Trees in the C2PA Manifest. I settled on grouping fragments in sets of eight and signing these into one Merkle Tree. This way I reduced the number of fragments needed to be signed from all existing fragments down to at most eight. This significantly lowered not only the time it took to sign new fragments but also the amount of data required to be published to the CDN, a lot of which was redundant data. I tried out two mutations of this approach as well, both of which decoupled the embedded C2PA from the file themselves and one of them put that data on a separate server and the other put the data into the DASH/HLS Manifests, namely the MPD for DASH and the MediaPlaylists for HLS. However, I had several issues with the implementation of these mutations and since this optimized Merkle Tree approach still required the uploaded of a lot of redundant data, I ultimately scrapped

these two mutations in favor of creating a completely new signing method. This method combined all the previous lessons I had learnt and also wasn't limited by adhering to the C2PA specifications. It uses a Rolling Hash to created a single chain of hashes, with each link being one media fragment of a representation of the live stream.

I measured the performance of all three of these signing method by looking at how long the actual signing process takes and how much data is required to be uploaded to the network. For this benchmark I created a live stream consisting of 100 media fragments at four different quality settings. One for a typical audio track and three at low, mid and high video encoding levels. The results confirmed that the original method was the worst method by a significant margin. Both metrics increased in a linear fashion as the number of fragments increased. The optimized approach improved both metrics significantly. They both still increased linearly, since it is still the same Merkle Tree concept, but every eighth fragment the duration and upload size reset, capping out the maximum at a much lower value compared to the original method. Finally, the Rolling Hash approach came out to be the best signing method of the bunch. It has the lowest signing duration as well as the lowest upload sized.

7.2 Dissemination

The application possibilities for C2PA are near endless. So much of our current world revolves around media and there is no way around seeing countless pieces of media every day. From images, videos and live streams on social media or any other website, PDFs at work, movies or music on streaming services to TV broadcasts, media is everywhere in this digital age.

The signing and validating of C2PA can be integrated into even more places, since media goes through so many steps of processing before it is being released to be viewed by the public. The validating can be integrated in any of the previously mentioned places where media can be viewed and so many more. The signing can be made part of digital tools like FFmpeg, editing software, any actors on the way of the journey media takes and many more. But it can also be integrated into physical devices, like cameras, scanners, smartphones and more.

The more places C2PA can get integrated, the better user are able to verify that the media they are viewing is from a trustworthy source, while also protecting less trained eyes from artificially generated or forged media.

7.3 Outlook

Probably the most important task for the future is the continued development of the C2PA specifications to not only increase the types of supported media but also mature the implementations to get as many people as possible to integrate C2PA into their products. For example this testbed could be significantly improved if FFmpeg would be able to sign the generated content directly with C2PA instead of require the complicated steps of sending it to a HTTP server to sign it and then forward it to the CDN.

It could also be an interesting experiment to integrate C2PA into the Media over QUIC transport (MoQT) protocol, because MoQT already breaks the data it sends into tiny chunks of data and the C2PA data could be separately transmitted on MoQT tracks rather than having the data embedded directly in the media fragments.

The `c2pa-rs` library currently also only supports the hashing algorithms SHA256, SHA394 and SHA512 and if would a worthwhile investigation to explore more hashing algorithms to potentially find one that is able to hash data faster, to further cut down the signing duration.

All the live streaming situation have all been exclusively been made with a single representation. It is imperative to expand the signing methods with the capability to signing a live stream with multiple representations to enable adaptive bitrate streaming. Important for these expansions is that there needs to be a way for the different qualities to be somehow connected with hashes to be able to provide a secure options to validate a a fragment even after having performed a quality switch. Equally important is that, despite the increased computation requirement, the signing still needs to remain fast to keep the latency impact as low as possible.

In conclusion, C2PA promises a highly intriguing procedure of embedding the provenance and authenticity of a medium into itself. This technology can be a powerful tool to enable transparency of the creation process of media to allows users to verify the origin and validity of the media they are viewing.

List of Acronyms

API	Application Programming Interface
BMFF	Base Media File Format
C2PA	Coalition of Content Provenance and Authenticity
CLI	Command Line Interface
CDN	Content Delivery Network
CSS	Cascading Style Sheets
DASH	Dynamic Adaptive Streaming over HTTP
FOKUS	Fraunhofer Institut fuer offene Kommunikationssysteme
GUI	Graphical User Interface
HLS	HTTP Live Streaming
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
MPD	Media Presentation Description
URI	Uniform Resource Identifier
VoD	Video on Demand
XML	Extensible Markup Language

Bibliography

[1] “Manifest store image source.”

[2] R. Pantos and W. May, “HTTP Live Streaming.” RFC 8216, Aug. 2017.

[3] M. A. García and G. Camarillo, “Extensible Markup Language (XML) Format Extension for Representing Copy Control Attributes in Resource Lists.” RFC 5364, Oct. 2008.

[4] F. Yergeau, “UTF-8, a transformation format of ISO 10646.” RFC 3629, Nov. 2003.

[5] R. C. Merkle, “Method of providing digital signatures,” US patent 4309569, Jan 5 1982.

[6] C. Bormann and P. E. Hoffman, “Concise Binary Object Representation (CBOR).” RFC 8949, Dec. 2020.

Annex

```
#!/usr/bin/env bash

# change directory to this file to
# make it runnable from anywhere
cd "$(dirname "$0")" || exit

# use ENV variables to override configured defaults
OUTPUT="${OUTPUT:-http://localhost:6262/ingest/bbb/source.mpd}"

ffmpeg \
    -loglevel error -stats -hide_banner \
    -f lavfi -re -i testsrc=size=1280x720:rate=25 \
    -profile main \
    -pix_fmt yuv420p \
    -map 0:v:0 -s:v:0 1280x720 -b:v:0 2048000 \
-maxrate:v:0 2200000 -bufsize:v:0 3300000 \
    -f dash \
    -dash_segment_type mp4 \
    -preset medium \
    -sc_threshold 0 \
    -r 25 \
    -keyint_min 48 \
    -g 48 \
    -c:v libx264 \
    -hls_playlist 1 \
    -master_m3u8_publish_rate 1 \
    -http_persistent 1 \
    -adaptation_sets "id=0,streams=v" \
    -seg_duration 1.920 \
    -update_period 1.920 \
    -use_template 1 \
    -use_timeline 1 \
    -utc_timing_url https://time.akamai.com/?iso \
    -write_prft 1 \
    -window_size 5 \
    -extra_window_size 0 \
    -flags +global_header \
    -init_seg_name \$RepresentationID\$/segment_init.\$ext\$ \
    -media_seg_name \
\$RepresentationID\$/segment_\$Number%09d\$.\$ext\$ \
    "$OUTPUT"
```

Listing 1: Example FFmpeg Shell Script

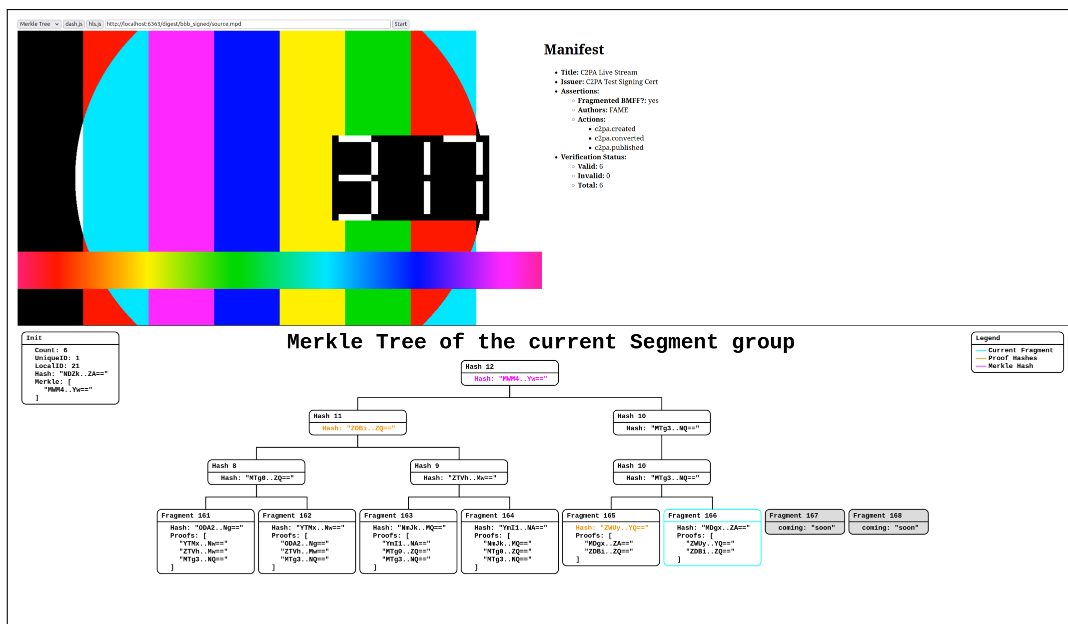


Figure .1: Example Incomplete Merkle Tree Visualization