ANALYSIS OF DATA CORRUPTION, PRESERVATION, AND RECOVERY OF MP4 MEDIA FILE PLAYABILITY

A Thesis

by

MATTHEW J. BARRY

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE, COMPUTER SCIENCE

May 2024

Major Subject: Computer Science and Engineering

ABSTRACT

Video media pervades our lives: movies and shows for entertainment, tutorials or documentaries for education, security footage, scientific monitoring, video calls for work or keeping in touch, recordings of cherished memories, etc. We capture this data in video files using various cameras and smartphones, but unfortunately, these file formats can be brittle and prone to corruption. Imagine opening an irreplaceable video only to see an error message, "Sorry, this file is damaged and cannot be played." What causes this to happen? How can this be avoided? Can such a file be repaired? This research answers these questions for the MP4 file format, the most commonly used format in commercial and consumer devices at the time of writing. We identify the likely cause of unplayable files, present a preservation technique that adheres to the MP4 specification, and identify the necessary steps to restore the playability of these damaged files.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professors John Andrew “Drew” Hamilton (advisor), Riccardo Bettati, and Jyh-Charn “Steve” Liu of the Department of Computer Science and Engineering, and Professor Jeyavijayan “JV” Rajendran of the Department of Electrical Engineering.

All other work conducted for this thesis was completed by the student independently.

Funding Sources

No other outside source of funding was provided.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Video media pervades modern life: it captures memories of important events, keeps records for later review, tells engaging stories for entertainment, and spreads news about current events. Much of this video data has great intrinsic or sentimental value and is worth preserving. Yet, some videos succumb to data corruption over time and eventually lose their ability to be viewed. Digital video files are saved in many different formats. At the time of writing, MPEG-4 (MP4 for short) is one of the most common formats, and a preliminary survey shows it is particularly susceptible to this type of corruption and loss. This study investigates the underlying cause of this problem and aims to mitigate against future data loss.

1.1 Background

Digital forensics, as the subject implies, brings the rigor of forensic investigation to digital applications. Any forensic science seeks to answer the question, "What happened here?" usually in the wake of some unknown or undesirable event such as a criminal investigation. Digital forensics frequently involves working with data assumed to contain valuable information, but that information may not be known ahead of time and must be uncovered through analysis and interpretation. Low-level sequences of bits stored in a computer may seem like random noise on initial inspection. These bits derive meaning from how they are interpreted, which, in turn, depends on the method used to encode the information they represent. The loss of this critical metadata might result in the effective loss of the data itself, akin to losing the key needed to decrypt a secure message. More fundamentally, even if the proper decoding method is known, the resulting interpretation is potentially meaningless if the data itself is corrupted.

Within the realm of digital forensics, data recovery methods provide a means of detecting and repairing data corruption or reverse engineering its interpretation. However, such *a posteriori* methods come into use only after the damage has been done. The task of hardening data against corruption or aiding in recovery *a priori* belongs to the field of data preservation. Both of these fields complement each other: improvements to data preservation tactics amplify the effectiveness of data recovery methods, and advances in data recovery improve the perceived reliability of current data formats and storage practices.

Media information collectively refers to audio, photographic, and video information. Encoding such information into files poses a particular challenge due to its complexity. By contrast, numeric and textual information can be encoded efficiently in compact formats that are relatively easy for both humans and computers to interpret. This is due to the symbolic nature of text and numbers, where information is conveyed through discrete finite values and glyphs. Media data carries much more information and is intended to reconstruct the signals that stimulate our senses directly. Consider the nature of this data. Our senses and the mind's interpretation leave much room for the data to vary: two similar-looking videos might have subtle variations in subject placement, colors, brightness, audio, timing, etc. While there is significant room for variation, media formats represent a form of ordered information that is distinct from randomness. Note the virtual impossibility that any such meaningful patterns could emerge from truly random data (e.g., white noise and "snow"). The combination of variation and non-randomness in this information implies a large amount of entropy and, therefore, a large amount of bandwidth required to encode it.

Media data are also highly dimensional, depending on the type. Audio data consists of a sequence of channels oriented along a single time dimension. Photographic media consists of color channels within a two-dimensional pixel array and reconstructs a visual stimulus at a single point in time. Video data is particularly interesting because it is a combination of both, with two-dimensional photographic data drawn out over time and usually accompanied by parallel audio data. If left in their raw formats, such video data would occupy an inordinate amount of storage space; so audio and visual *codecs* employ various compression methods to retain the original meaning or value of the data while reducing the amount of space required to store it. Some codecs are perfectly reversible and therefore *lossless*, whereas others represent an approximation of the original data and discard the residuals—i.e., *lossy* codecs.

Codecs encode a single stream of either audio or visual information, but not both. Therefore, to represent a video as a single file unit, the file format must contain a combination of encoded audio and visual *streams* related by a common timescale. These file formats are called *containers*. At the time of writing, MPEG-4 Part 14 (MP4), Matroska (MKV), and WebM (a derivative of MKV) represent some commonly used video container formats, with MP4 being highly favored by a vast majority of available hardware and software. Each container format supports one or more codecs for storing each contained audio and video stream. In the case of MP4, the most often used audio and video codecs are AAC and H.264, respectively. The Advanced Audio Coding (AAC) codec was first standardized in MPEG-2 Part 7, and the Advanced Video Coding (H.264) codec was introduced in MPEG-4 Part 10. Focusing on these most popular formats limits the scope of research while maximizing relevance to the general public.

1.2 The MP4 Container Format

1.2.1 Specification History

The specifications governing the MP4 file format have their origin in Apple's QuickTime File Format (QTFF), which was originally published in 2001 [1]. This format was standardized and promulgated in ISO/IEC 14496-1 [2] later that year by the Moving Picture Experts Group (MPEG) [3][1]. Since then, the format has been refined and updated to give rise to the following current standards:

- ISO/IEC 14496-12:2022 – ISO base media file format [5],
- ISO/IEC 14496-14:2020 – MP4 file format [6], and
- ISO/IEC 14496-15:2022 – Carriage of network abstraction layer (NAL) unit structured video in the ISO base media file format [7].

14496-12 defines the basic structure used by the MP4 file format, and then 14496-14 extends this basic structure to include additional support specifically for certain audio and video codecs. The contents of 14496-15 are beyond the scope of this study.

[1]The MPEG group has since been closed in favor of the *Moving Picture, Audio, and Data Coding by Artificial Intelligence (MPAI)* community [4].

It should be noted that these official standards are not freely available: each publication costs 208 CHF (232.96 USD) to purchase at the time of writing. However, the specification can be obtained in other ways. The MPEG website archive [3] provides older, withdrawn versions of each publication, while Apple maintains the current version of the QTFF specification on its website with the following notice:

> The QTFF has been used as the basis of the MPEG-4 standard and the JPEG-2000 standard, developed by the International Organization for Standardization (ISO). Although these file types have similar structures and contain many functionally identical elements, they are distinct file types. [8]

Despite the claim that the file types are distinct, the two file format specifications are similar enough from a technical perspective to develop a tool capable of reading files in either format. The structure of the MP4 file format can also be learned by inspecting the source code of open-source software that is capable of opening MP4 files, such as VLC [2], MPV [3], and ffmpeg [4], to name a few.

1.2.2 Basic Internal Structure

Internally, an MP4 file contains a composite hierarchy of data units arranged as a tree. Apple's QTFF Specification [8] refers to these as "atoms," whereas the ISO/IEC MPEG-4 specification refers to these as "boxes." The two definitions are practically identical, and the terms may be used interchangeably in this document. However, the latter will be preferred out of respect for the standard, even though the information presented here relies heavily on Apple's freely available specification instead of the paywall-restricted ISO/IEC specification.

Each box consists of a header and contents. The header includes a 4-byte discriminant value that helps to determine the type of data that the box contains. This discriminant, in combination with the box's position within the tree (i.e., "ancestor" boxes and their types), determines the exact field structure of a box. Apple gives the following example in their specification: "the *profile* atom

[2] `https://www.videolan.org/vlc/`
[3] `https://mpv.io/`
[4] `https://ffmpeg.org/`

inside a *movie* atom contains information about the movie, while the *profile* atom inside a *track* atom contains information about the track" (emphasis added to denote types). Given the above general description, all boxes contain at least the following two, maybe three, basic header fields in the provided order:

1. `size`: *unsigned 32-bit integer*, the number of bytes that the whole box occupies, including these fields. There are two special cases for this value:
   (a) The root box (and *only the root*) may contain a size of 0 if its contents continue until the end of the file.
   (b) A box whose size is over 2^{32} bytes in length will have a size of 1 and provide the actual size in an additional `extended size` field below.
2. `type`: *4-character string*, the box's type discriminant.
3. `extended size`: *unsigned 64-bit integer (optional)*, this field is present only if it is larger than what can fit in the `size` field. Otherwise, it is either omitted or filled with a placeholder `wide` box.[5]

Of the many types of boxes defined by the specification, the following types are of interest: `moov` and `mdat`. A movie file must contain a `moov` box because it contains metadata about the movie and how to interpret it. Such metadata includes the number of tracks in the file, the type (audio, video, subtitles, etc.), codec used, and time synchronization information for each track, references for converting time indices to data locations, etc. The data referenced by the `moov` is stored in `mdat` boxes elsewhere in the file and not in the `moov` box itself. `mdat` boxes are relatively simple: each contains a header with the `size`, a `type` of `mdat`, and optionally the `extended size`, followed directly by binary media data. The structure of a `moov` box is much more complicated, and additional details of its structure are available in Appendix A.

[5]A `wide` box is an "empty" box containing only a header with a `size` of 8 bytes and a type of `"wide"`. Filling the extended size with a `wide` box has the effect of allowing the box to grow larger than 2^{32} bytes without needing to rewrite or reallocate it.

Atoms deeper within the `moov` hierarchy are more tightly coupled with the individual pieces of the data that comprise the media information. For example, The `dinf` atom indicates where the data resides, i.e. the current file itself or in an external file, and the `stbl` atom provides a mapping from time and sample number to the offset/address where the sample's data can be found. According to the QTTF specification, "a sample is a single element in a sequence of time-ordered data," the quantum of data represented by an MP4 container.

1.3 Problem Statement

The MP4 video file format is susceptible to data corruption. The degree to which this corruption affects the encoded video varies greatly. In some cases, minor corruption may have little to no effect. Noticeable effects may consist of a benign reduction in media quality or alter the video in some way. These types of errors leave the file in a still-playable state and sometimes can be recovered to their original quality using existing video recovery methods. However, in some unfortunate cases, even small amounts of data corruption in the right (or wrong) place can render an entire MP4 video file unplayable even though the vast remainder of the file remains intact.

The goals of this research are to investigate why data corruption leads to unplayable MP4 video files, preserve files from this type of corruption, and to recover video files that have already been rendered unplayable.

1.3.1 Investigate

Identify the likely cause or causes of data corruption that can render MP4 files unplayable.

The data corruption process by which MP4 files are rendered unplayable may happen naturally over time without any human intervention, unintentionally through user error, or purposefully for some intended (possibly malicious) reason. Nonetheless, we hypothesize that a simple, common underlying mechanism exists by which MP4 videos are rendered unplayable. The findings of this investigation will determine how we approach the remaining research goals.

1.3.2 Preserve

Determine a way to harden MP4 files against this type of data corruption.

The MP4 file specification offers some flexibility in the structure, amount, and kind of data that can be stored. We hope to find a means of adjusting the file or embedding additional information that will reduce an MP4 file's susceptibility to the cause(s) identified in the first goal while still adhering to the specification.

1.3.3 Recover

Examine the feasibility of returning an unplayable MP4 file to a playable state.

The possibility of fully recovering unplayable MP4 files is tantalizing and could be extremely helpful. However, reconstructing lost data is incredibly challenging and even impossible depending on the nature of what was lost. Developing a general-purpose tool to perform MP4 file recovery might go beyond the scope of this research, so this goal will be to attempt recovery of isolated cases uncovered during the course of this research. The study findings for preservation may reduce or even eliminate the need for MP4 recovery.

2. LITERATURE REVIEW

Video formats, particularly the MP4 container format, have been the subject of much research in digital forensics due to the wealth of information they contain. Their widespread publication and availability on social Internet forums and the massive variety of recording devices, especially smartphones, ease the spread of compelling narratives. For this reason, videos are also of great value to any type of forensic investigation. Feeds from security cameras, dashcams, and even mounted GoPro-style cameras capture objective evidence of how historical events unfolded from the perspective of a first-person observer.

2.1 MP4 Metadata Structure and Interpretation

From an outside perspective, MP4 video files contain a tremendous amount of data: a single video that is only a few minutes in duration can have a file size approaching hundreds of megabytes, depending on the quality. Most of this, however, is occupied by the stream data. The metadata, in contrast, is only a few kilobytes in total and can be scattered anywhere throughout the file.

The MP4 file structure is hierarchical, and its prohibitive size prevents the entire tree from being loaded into memory directly. Traversing the tree in its entirety might require multiple passes through the file, which can severely impact read performance. An optimal approach for parsing MP4 metadata is to use an event-based parser that reads the file in a single pass, noting the file offsets where various metadata of interest occur [9]. This approach uses a stack structure to record its traversal state since it encounters nested nodes in a depth-first fashion. Similar methods exist for parsing other types of hierarchical data efficiently: namely, SAX for parsing XML [10].

2.2 Data Corruption

Digital forensics often requires sifting through incomplete, fragmented, or otherwise corrupted data. Such damage to a file's data stems from various causes and can occur when data is "in transit" between storage devices and when it is "at rest" on a single isolated storage device.

2.2.1 Corruption in Transit

Although often overlooked as such, copying data between locations within a single device is a form of data transfer. As with the scribes and historians of old, transcription errors can arise when copying data from one place to another. In a digital device, failures in the electrical hardware are likely causes of such errors. More often, however, data transfers occur between devices over a network.

The physical layer is the lowest level of Internet Protocol (IP) network infrastructure and has implementations using various wired (e.g., coaxial, Ethernet) and wireless (e.g., 802.11) technologies. At this level, network transmissions are prone to interference and noise from external sources and network hardware failures from within. These can introduce errors in the transmitted signal, thereby reducing the connection reliability. Transmission Control Protocol (TCP) offers a robust means of data transfer over these potentially unreliable connections at the expense of communication and computational overhead. On the other hand, User Datagram Protocol (UDP) provides a simple and fast way to relay data between two networked devices without the overhead. Still, these communications are susceptible to data corruption in transit.

2.2.1.1 Dropped Packets

When transmitting large amounts of data, the sending software splits the original data into multiple smaller segments or packets, which the receiver then reassembles. Some of these packets may be delayed or lost, and the data they contain will be out of order or missing entirely in the reassembly. TCP contains packet sequence numbers to enforce correct ordering and requires acknowledgment of each transmitted packet [11]. Failure to receive an acknowledgment will result in packet retransmission, and unexpected errors in sequence numbering will trigger a connection reset. In the end, the receiver can know with confidence that all packets will received and reassembled in the proper order, but this confidence comes at the cost of doubling the number of exchanges. Every transmitted data packet must have an accompanying acknowledgment returned to the original sender. This overhead prevents devices from taking advantage of the total available network bandwidth.

For live media streams, maximizing the amount of information bandwidth maximizes the perceived quality of the media, so applications generally prefer UDP for this purpose. The downside is that UDP does not detect or correct either issue, so the application layer must handle problems. Padmanaban and Ilow [12] explored ways to handle dropped network packets from H.264 video streams. Their method relies on the fact that H.264 already packetizes its stream data. In addition to sending the video packets, this approach sends interspersed parity packets from which the receiver can reconstruct dropped video packets if necessary. This manner of reconstruction requires support at both ends of the communication: the sender must encode the parity packets, and the receiver/player must know how to take advantage of them.

2.2.1.2 Random Transmission Errors (Bit Flips)

One of the mechanisms used by TCP to ensure reliability is a computed checksum for each packet [11]. This checksum determines whether any transmission errors occurred and, in some cases, enables the receiver to correct them. In the former case, the recipient could withhold acknowledgment for a packet containing a checksum mismatch and wait for the sender to reattempt transmission. However, computing and verifying checksums require even more overhead and further reduce usable bandwidth. This overhead is yet another reason why UDP is favored for live video streams. Padmanaban's method above provides a way to use parity data to identify and correct bit flips as part of a live stream.

2.2.1.3 Premature Stream Termination (Truncation)

All forms of network transmissions are susceptible to sudden termination. This connectivity issue can arise from several causes, such as the loss of an intermediate network device, removal or severing of transmission lines, termination of the software performing the sending or receiving, etc. In any case, the receiver has only a partial copy of the intended data. The worst-case scenario for this type of error is when it occurs silently, leading the user to believe the transmission is complete when, in fact, it has failed. In the case of media files, this can occur when recording data from a live stream (e.g., security footage or a digital broadcast) that is interrupted or otherwise ends abruptly.

This type of corruption is not limited to transit over a network. For example, when copying a large file to a removable storage device, the operating system typically copies the file into an in-memory buffer. Doing so helps to keep the operating system responsive to other tasks while data is flushed from this buffer to the storage device asynchronously. As part of the device unmounting or "safe removal" process, the operating system ensures that any such cache buffers are flushed completely. However, if the storage device is removed without being safely unmounted, the operating system may not have finished writing transferred files. Files that occupy a large amount of storage (a characteristic of most media files) are particularly prone to unsafe removal. Partially written copied files can be especially troublesome if the target destination is a backup that may need to be used to recover the files later. In the event of such a recovery, a user would be troubled to find that the supposedly backed-up files are, in reality, not recoverable.

2.2.2 Corruption at Rest

Data in transit is particularly vulnerable due to the number of problems that can arise when transcribing the data. Once data is written to a (non-volatile) storage medium, we might consider the data safe from any corruption. On the contrary, storage devices themselves are prone to data degradation.

How storage degrades over time is dependent on the storage technology in use. Here are some common storage methods and how they can fail with age:

- *Mechanical* storage devices, such as hard drive disks (HDDs), depend on finely tuned moving parts that can wear out, drift, or break due to physical shock, whereas *solid-state* storage devices are more durable in this regard.

- *Magnetic* storage, such as tapes, HDDs, and floppy disks, can weaken over time when exposed to heat or external magnetic fields (such as that of the earth) or other electromagnetic sources.

- *Electrical* storage, such as flash memory, often relies on keeping a capacitor at a specific voltage, but electrons can drain slowly as the dielectric material wears out with use and age. Flash memory is particularly vulnerable due to destructive write operations.

- *Optical* storage, such as compact disks (CDs), digital versatile disks (DVDs), and Blu-ray disks, are susceptible to warping and scratches. However, with the proper materials and care, this storage category is generally considered the most stable for single-volume, long-term digital storage. The mechanical drive or reader may wear out eventually, but the data storage medium tends to be robust. [1]

2.2.2.1 Bit Rot

Regardless of the storage technology in use, corruption at rest arises from the physical properties of the storage medium. Degradation over time eventually leads to errors during data retrieval: what was once a 0 might now be interpreted as a 1 or vice versa. The storage device is still operational, but its data might need to be refreshed to “full potency.” The term *bit rot* describes this general type of operational degradation. [13]

Many modern storage technologies use error correction to improve storage density and lifetime. As bit rot begins to occur, read performance will degrade as the device must correct any misread bits. However, older and simpler storage devices may not have any forms of error detection and correction: the bits read from the device are sent verbatim to the operating system.

2.2.2.2 Sector Read Errors

Storage devices, filesystems, and operating systems all tend to work with data in equal-sized chunks called *sectors*. This term originates in rotating storage media: as the medium rotates, the read/write head sweeps out an arc or sector along the surface as it reads the requested data. If a sector contains too many bit errors for the on-device error correction to handle, the device will report a read failure, leaving a gap in the retrieved data. Any damage or imperfections in the

[1]Optical storage is a class of other physical storage methods that can be "etched" into a physical medium. Such strategies tend to be “write-once, read-only.” By contrast, other energy-based storage methods rely on preserving some type of energy gradient. Such states have a natural tendency to return to an equilibrium. These strategies can be erased and rewritten.

storage medium can cause similar failures when reading the affected sectors. In the case of SCSI- and SATA-connected (magnetic) hard disk drives, vendors specify an error rate of one error per 10^{13} to 10^{16} bits read. [14] These read errors can also manifest as bit rot.

2.2.2.3 Fragmentation

While not a form of corruption *per se*, the block-based nature of filesystems can lead to fragmentation. This occurs when blocks containing a file's data cannot be stored contiguously in the storage medium. Fragmentation becomes problematic for interpreting video data when the filesystem metadata governing the location and ordering of blocks is damaged or no longer present. Such situations arise when recovering deleted files or quick-formatted storage volumes.

2.3 Data Integrity and Preservation

2.3.1 Transmission Validation

Savvy users know the potential for transmission errors or foul play when downloading large or important files from the Internet. Hence, a commonly recommended practice is to validate a file's integrity against a published checksum, hash, or cryptographic signature provided by the original file's author. A validation failure indicates a problem with the downloaded file, in which case the download should be reattempted, perhaps from an alternate mirror if multiple hosting servers are available.

2.3.2 Multiple Volume Storage: RAID

Fortunately, bit rot and sector read errors are relatively easy to prevent: RAID arrays (levels 2–6) maintain parity information that allows information to be recovered in the event of read errors or hardware failures [15]. All RAID systems have a limited redundancy threshold representing the number of disk failures that can be tolerated. RAID's error correction detects problems only when reading the data, and, as previously mentioned, errors can aggregate silently over time. If left unchecked, the data could still degrade to the point where it is no longer recoverable. Modern software RAID systems such as ZFS and BTRFS address this shortcoming by proactively and periodically "scrubbing" the data to maintain its integrity [16] [17] [18]. A scrub operation reads all stored data to check for and repair any errors, which prevents silent errFors from accumulating.

2.4 Data Recovery

2.4.1 Filesystem-Based Recovery: File Carving

Regardless of the type or source of corruption, recovering usable data from a corrupted source requires considerable effort and is not always successful. Methods such as file carving [19] [20] attempt to recover a file's original content from remnants of filesystem metadata. File carving recovers deleted files from the "free space" on a storage medium. When deleting a file, the operating system only marks the file's allocated blocks as unused, but the blocks themselves (and file data therein) remain intact until they are overwritten. Large files that span multiple blocks become fragmented when the blocks used to store a file are not contiguous within the storage medium. This is often the case for video files due to their size.

Early file carving tools like Scalpel [21] did not compensate for filesystem fragmentation. Poisel and Tjoa recognized this problem and outlined a "roadmap" to develop a file carving technique for fragmented multimedia files [22]. Casey and Zoun explored tradeoffs for methods that produce longer playable fragments versus many shorter fragments and found the total duration of recovered video varies by case and method [23]. Na *et al.* [24] propose using H.264 frame headers to identify video stream data across multiple blocks and then arrange those blocks into a valid video file. Other recent tools like VidCarve [25] similarly account for fragmentation and, in this case, have been tailored specifically for recovering video files.

File carvers for video data maintain a metadata index and attempt to match available movie data. Carvers generally discard segments of movie data without a matching header because they are unplayable. , data becomes spliced into another video file because of a false positive match in the metadata index. Generally, file carving assumes the presence of all video data and metadata needed to construct a playable video, but it often struggles to assemble video file fragments in the proper order. When only partial video data or metadata is available, file carving produces unexpected results.

2.4.2 Stream-Based Recovery Methods

2.4.2.1 Noise reduction

Theoretically, the media data contained in a stream—assuming it can be properly decoded from its container—represents an ordered signal, and any corruption could be classified as introducing some random interference. Stated differently, the signal represents some function we wish to recover, and the data corruption is an undesired alteration to the output of the signal function. From this perspective, data recovery takes the form of noise reduction within the realm of signal processing. Fourier transforms and wavelet decompositions are well-known tools in this area and have been used in the recovery of degraded audio [26] [27] and image [28] [29] data.

2.4.2.2 Frame Reconstruction

Zhang and Stevenson [30] proposed an alternate video encoding method consisting of parallel streams, allowing dropped frames in one stream to be reconstructed from nearby frames in an alternate stream. Chen *et al.* [31] explored using steganography to embed recovery data directly within an H.264 stream.

Corrupted video data can originate from online sources as well. Transmission errors from live-streamed video sources require live error detection, correction, and, if that fails, recovery.

More recently, Mary P. D. *et al.* [32] and Hoang *et al.* [33] have developed tensor completion techniques for filling in missing pixels in degraded video frames. However, as mentioned previously, all of these methods operate at the stream level, and their success depends on intact and well-formed MP4 container metadata.

2.5 Available MP4 Video Datasets

Datasets like the one developed by Gloe [34] as well as VISION [35] and FloreView [36] aggregate hundreds—even thousands—of media files from dozens of devices for comparative metadata analysis and research. The latter dataset even uses the same subjects across all media captures. The EVA-7K [37] dataset includes videos modified by various video editing software and shared online through common social platforms. These datasets are intended for comparative analysis to identify a video's origin and any possible manipulation.

2.6 Honorable Mention

A survey of the current state of digital forensics, especially that which is pertinent to media data, would be incomplete without including some significant areas of interest. These topics may not be readily or obviously applicable to the research presented here, but they are nonetheless vital to understanding current problems. Perhaps someone else will find the material presented here helpful.

2.6.1 Manipulation and Deepfake Detection

In the past, video evidence was regarded as indisputable. The underlying assumption was that video would be extremely difficult or impossible to fabricate. Video manipulation methods were primitive and left noticeable artifacts in the resulting videos. Today, video editing software is more widely available and has improved, making it more challenging to identify edits visually. Moreover, with the recent introduction of "deepfakes" created by artificial intelligence, verifying a video's authenticity is crucial before trusting its content. Manipulation detection is a valuable defense against the effects of disinformation campaigns in today's Internet-connected culture.

2.6.1.1 Pixel-based Methods

Early work in pixel-based manipulation detection focused primarily on detecting visual artifacts or comparing similarities with a known original video [38]. However, as editing software has improved, these artifacts are becoming more challenging to identify. In some cases, distinguishing between an original video and a modified version might not be possible from visual indicators alone.

The ubiquity of artificial intelligence for handling visual information and identifying subtle patterns has led to the irony of "fighting fire with fire:" the same tactics used to make some oTxf the most compelling manipulated or fabricated videos are being used to identify them. Rana *et al.* identified 112 studies between 2018 and 2020 for deepfake detection alone [39], 86 of which used a form of deep learning or machine learning. Facial recognition accounts for most of these strategies since generated videos are not likely to reconstruct an individual subject's facial features perfectly. The success of these pixel-based approaches depends on the machine learning model used. As evidenced by the large number of studies, researchers have developed an ever-widening selection of models from which to choose.

2.6.1.2 Metadata-based Methods

The metadata embedded within video files can provide an audit trail and clues about a video file's origin. This information can be direct, such as an embedded "Edited by such-and-such software" comment or marking, but an adversary with intent to deceive would likely strip out such apparent indicators. Hence, manipulation detection methods that rely on more subtle markings are invaluable for forensic investigation. MP4's metadata structure allows for ample variation while remaining true to the specification, and different recording devices or editing software tend to leave their own "fingerprint" in the metadata of the files they produce.

Xiang *et al.* developed methods for analyzing this metadata using nearest-neighbor classification [40] and self-supervised neural networks [41]. Metadata-based manipulation detection may provide a more reliable means of manipulation and deepfake detection over pixel-based methods as video manipulation tactics continue to improve.

2.6.2 Steganography

MP4 videos can also provide an enormous cover for surreptitiously embedding additional data. Hemalatha *et al.* developed a way to embed secret audio and image data in the wavelet domain of each of the cover video's streams [42]. Their method offers increased payload capacity and a better signal-to-noise ratio than prior video steganography methods.

Steganography greatly interests the broader digital forensics field but may seem less applicable to video recovery. On the contrary, there is some relationship between these pursuits. Steganography aims to embed data without any recognizable effect on the cover data. However, any changes to the cover data invariably result in artifacts or signal degradation to some degree since a certain amount of its former bandwidth is hijacked for use by the embedded signal. In the case of video cover data, the video recovery methods mentioned above could reverse the embedding process, returning the original video to a closer approximation of the original data. If the embedded data is also a video stream, the above video recovery methods could reduce the bandwidth needed to perform the embedding. Moreover, as previously discussed in [31], steganography can embed arbitrary recovery information in the data itself.

2.7 Summary

The existing literature demonstrates a need for further research into how intact MP4 video files might become unplayable over time. All of the identified prior research assumes that files remain consistently playable, but experience demonstrates that an MP4 file's viability is not guaranteed. Existing methods such as file carving offer some insight into how potentially fragmented video files might be reassembled, but this process itself often leads to unplayable videos. Moreover, file carving attempts to recover the file contents exactly, so a file that was already unplayable prior to carving will be unplayable even after a successful recovery. Other recovery methods operate at the codec level, repairing errors in the media stream rather than in the container itself. Fortunately, the general ways in which data can become corrupted are well-known and succinctly enumerated, but have not been applied to their effect on MP4 files. To our knowledge, the research presented in this thesis is the first to document and investigate this phenomenon in a rigorous academic setting.

3. METHODS

A review of current literature reveals an opportunity for research to identify why MP4 videos might become unplayable. It is possible that the answer to this question may be known already by some in the industry or reside in other protected knowledge repositories. Nonetheless, we hope to provide the answer in an open academic setting and begin to explore some solutions to this problem. This chapter outlines a series of experiments to investigate what types of data corruption can render MP4 files unplayable, evaluate a method to preserve files from these types of data corruption, and determine how to recover already unplayable files.

3.1 Investigation

The previous chapter highlighted several ways in which a file might become corrupted at rest or in transit. Although these are not exhaustive, they represent a significant set of common data corruption instances that a file might encounter with regular use over time. We can generalize these and similar instances of data corruption using the following data corruption model:

1. *Bit-flipping*, which accounts for minor network transcription errors or bit rot. This is assumed to occur with uniform randomness across all data contained in a file. The severity is parameterized as the number of bits k_B that are flipped from their original values.

2. *Segment/sector blanking*, which represents data loss from dropped packets or sector read errors. For this type of corruption, we model file data as being split across an ordered sequence of contiguous segments, all of which are of size b bytes except for the last segment. Given a file of size n bytes, it is unlikely to be the case that b divides n, so the last segment will be partially occupied with only $n - b\left\lfloor \frac{n}{b} \right\rfloor$ bytes of data. Similar to bit-flipping, this type of corruption is assumed to occur with uniform randomness across all segments in a file, and the severity is parameterized by the number of segments k_S with missing data.

3. *Truncation*, which primarily occurs from an interrupted file transfer but could also result

from partial recovery due to fragmentation or file carving. The severity of this type of corruption is parameterized as the number of bytes k_T missing from the end of the file.

We can induce the effects of this model in a controlled manner by intentionally modifying the binary data comprising sample MP4 video files. To accomplish this, we implement the model as a "mangler" utility program that subjects an original input MP4 file to one of the parameterized binary modifications. The following implementations and experimental conditions outline how we can determine the effects of these binary modifications on the input MP4 file's playability:

1. *Bit-flipping:* Seeking to a random byte offset within the file and flipping a random bit within that byte. This process will be repeated for $k_B = 1, 10, 100$ times to simulate k_B bit flips.

   Hypothesis 1. *Bit-flipping renders MP4 files unplayable with probable (i.e., $p > 0.5$) regularity.*

2. *Segment blanking:* We use a segment size of $b = 4$ KB, which is a typical sector size for filesystems on hard drives. For a random byte offset o within the file, seek to the starting byte offset $\lfloor \frac{o}{b} \rfloor$ of the containing segment and set b bytes to all 0's. Let n be the file size in bytes. If the selected segment is the last in the file, only $\min(b, n - o)$ bytes will be set to 0's so that the file's size will not change as a result of this induced corruption. This process will be repeated for $k_S = 1, 2, 3$ to simulate k_S segment errors.

   Hypothesis 2. *Segment blanking renders MP4 files unplayable with probable (i.e., $p > 0.5$) regularity.*

3. *Truncation:* Clipping the end of a file by reducing the file's size by k_T bytes, where $k_T = 2\text{ KB}, 200\text{ KB}, 500\text{ KB}$.

   Hypothesis 3. *Truncation renders MP4 files unplayable with probable (i.e., $p > 0.5$) regularity.*

The source code for the key components of this mangler utility is provided in Appendix B.

For these experiments, the video file contents do not matter as much as the MP4 container structure. As documented by Xiang and others [40] [41] [34], video files created from the same recording device or encoder will share the same structure, but this structure will vary between different devices and encoders. This variation is expected and permitted by the MP4 specification to which all compliant devices and players adhere. The FloreView dataset [36] contains videos from a wide array of recent popular smartphone recording devices. A single playable MP4 file with H.264 video from each of four popular devices represented in this dataset, as listed in Table 3.1, should yield an adequate dataset for experimentation. All videos have a resolution of 1920×1080 pixels.

Filename	Device	Size	Framerate	Bitrate
D14åç_L1S1C4.mov	Apple iPhone 13 mini	61.6 MB	30 fps	17.681 Mbps
D17_L1S1C4.mp4	Samsung Galaxy S21+	71.0 MB	59.93 fps	21.889 Mbps
D34_L1S1C4.mp4	Google Pixel 5	104.9 MB	60.07 fps	33.035 Mbps
D43_L1S1C4.mp4	OnePlus 8T	34.0 MB	30.15 fps	10.463 Mbps

Table 3.1: Selected sample videos for experimentation from FloreView dataset.

Each video will be subject to the set of "mangler" experiments above to produce a total of 36 corrupted videos: 12 per corruption method. These corrupted versions of the original video will be evaluated for playability using VLC[1] and mpv[2], both of which are versatile, open-source, cross-platform media players.

3.2 Preservation

Based on the findings of the previous section, we plan to proceed by identifying what can be done to prevent the identified corruption method(s) from becoming a problem. Ideally, such preventative measures should operate within the MP4 specification. This could take a proactive

[1]`https://www.videolan.org`
[2]`https://mpv.io`

approach, in which the modifications prevent the identified problematic method of corruption from having the same destructive effect, or it could be used to assist in recovery after the file has been found to be unplayable.

To evaluate the effectiveness of any developed preservation method, we will modify the original videos from the experiments above. MP4 files that have been modified according to this preservation method will be referred to as "preserved MP4 files." These files should remain playable according to the same VLC and mpv criteria.

Hypothesis 4. *Preserved MP4 files are playable without being subject to any data corruption.*

The preserved files will then be subject to the same induced corruption experiments above that previously rendered them unplayable and evaluated for playability. Success will be measured by whether the playability of the preserved and subsequently corrupted files improves over their original corrupted counterparts.

Hypothesis 5. *Preserved MP4 files remain playable when subject to bit rot, segment blanking, and truncation.*

Preliminary results indicate truncation as a likely candidate for causing unplayability, which means that offset consistency (e.g., box sizes matching the actual file size and not overflowing) or the data at the end of the MP4 file is key for maintaining playability. Since the MP4 container structure is tree-based, and child order does not matter in most circumstances, relocating or somehow replicating the data at the end of the video could provide an effective means of hardening the video against truncation.

3.3 Recovery

Next, we turn our attention to recovering MP4 videos that have already been rendered unplayable and seek to restore them to playable status. We begin by observing that the data comprising an unplayable video file is a subset of the original (playable) file's data. Recovery, therefore, entails reconstructing some or all of the missing data, adapting the remaining valid data to account for any missing data, or a combination of both strategies. In the former case, the reconstruction could match the original data exactly or be an approximation or placeholder for the original data.

MP4 files contain massive amounts of data, not all of which may be necessary for the file to be playable. The task of restoring playability should focus on only the essential pieces of data. Boxes are the basic unit of data storage in the MP4 specification. So, to determine which parts of an MP4 file are minimally necessary for playability, we develop another mangler utility to strip out specific boxes from an intact MP4 file. Subjecting an original file to successive removal of each box will produce a series of candidate videos that can be evaluated for playability using the methods above. By noting the boxes whose removal led to an unplayable file, we can enumerate the types of data that may need to be reconstructed as part of the recovery process.

Hypothesis 6. *A video is playable if and only if it contains a minimal subset of data/metadata.*

Using this information, we inspect an unplayable MP4 file produced by the mangling process in Section 3.1 to identify which pieces of this minimal subset are missing. The method used to restore any such missing data will depend on the nature of that data. The recovery process or processes developed here must use only contextual information already present in the corrupted file and heuristics from what is generally known. This precludes the trivial case of copying analogous data from a different file.

Hypothesis 7. *Missing essential data/metadata can be recovered without reference to privileged information.*

Modifying an unplayable MP4 file to include the reconstructed data should then return the MP4 file to playable status by the above playability criteria. Additionally, playback of the recovered MP4 file should match or at least resemble that of the original file. This requirement avoids the case where “restoring” a corrupted file could result in a playable MP4 devoid of any original value.

Hypothesis 8. *Previously unplayable MP4 files will be playable and contain content recognizable from the original video after restoring missing essential data/metadata.*

3.4 Summary

We anticipate the root cause of unplayable videos can be reduced to at least one of the following modeled data corruption methods: bit flips, lost segments, or truncation. By inducing these errors in a controlled manner, we can identify the nature of the affected areas and their necessity for MP4 files to be playable. We will develop a method for modifying the files that maintains adherence to the MP4 specification, and we expect this modification to preserve the files' playability when re-exposed to the same corruption that previously would have rendered them unplayable. Finally, we intend to reconstruct the missing data necessary to restore an unplayable file back to playable status. Figure 3.1 shows a graphical flowchart depicting all experimental procedures and findings involved in this research, and a narrative description of these procedures follows.

First, we subject an input MP4 file to a battery of induced corruption methods to produce a set of corrupted MP4 files. Each corrupted file is then tested for playability: unplayable corrupted videos are saved for additional evaluation, and corrupted videos that remain playable are ignored. Each unplayable video contributes a tally towards the corruption method that was applied. This experimentation process is repeated for each additional input MP4 file, and the sum total of tallies will identify the corruption method(s) that reliably render MP4 files unplayable.

Next, we examine the set of unplayable files to determine the underlying data that is being lost and develop a method to preserve that data from the corruption methods identified in the previous phase. We apply this preservation method to the original input MP4 file to produce a new hardened input MP4 file. This file must remain playable in order for the preservation method to be a valid mitigation. The hardened MP4 file is then subjected to the same corruption method(s) that rendered the original MP4 file unplayable. If the resulting hardened corrupted file(s) remain playable, or if playability can be recovered using any post-corruption steps associated with the preservation method, then we consider the proposed preservation method effective and successful.

Before approaching recovery of the unplayable corrupted files, we use the original (non-corrupted) MP4 file to determine a minimal set of data that is required for the file to be considered playable. This process only needs to happen once since all input MP4 files will share the same minimal subset of required data. Removing each successive atom/box in the MP4 data hierarchy produces a series of candidate MP4 files to test for playability: if a candidate file is unplayable, then that atom-/box is marked as required. The aggregate union of all required atoms/boxes in the MP4 hierarchy thus represents a minimal set of required data.

Finally, using this set of minimal data, we can inspect the unplayable corrupted MP4 files from the first phase to determine what parts of the data missing from the original file are (or were) required. Based on the nature of this data, we develop a method to reconstruct or approximate it from contextual information and intact data elsewhere in the file. Reconstructing only this minimal data eases the recovery task and improves the chances of success. Adding this reconstructed data to the unplayable MP4 file results in a candidate recovered file: if the file is playable, then we consider the proposed recovery methods effective and successful. Note that these methods are localized to the experimental data presented here and may not work to recover all unplayable MP4 videos in general. However, a comprehensive MP4 recovery tool might include them among its recovery strategies for similar cases.

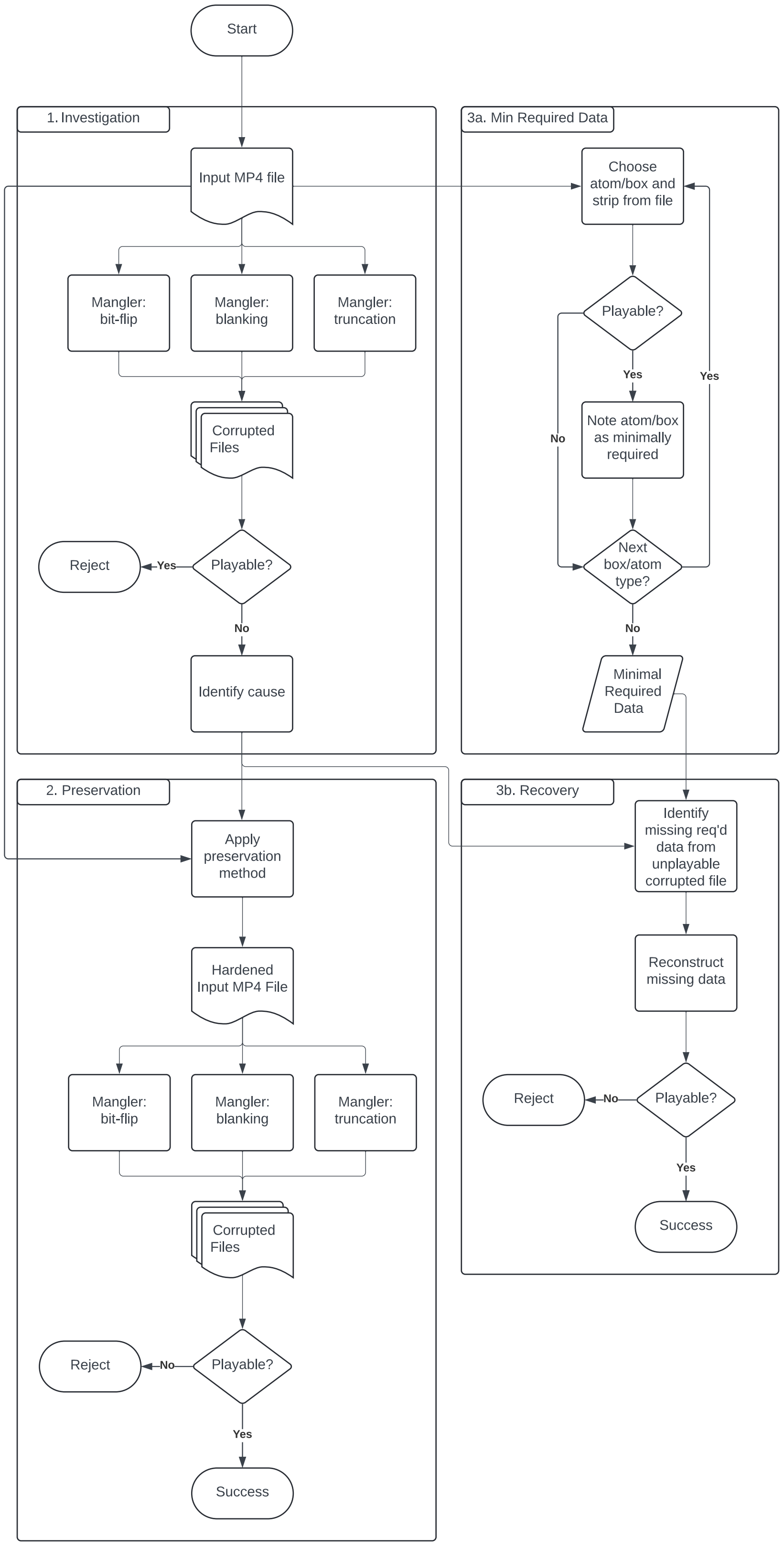

Figure 3.1: Overview of experimentation

4. RESULTS

4.1 Investigation: Effects of Corruption on File Playability

The videos selected from the FloreView dataset share the same “L1S1C4” suffix and will be referenced by their prefix for brevity: “D14,” “D17,” “D34,” and “D43.” Prefixes in this dataset correspond with the model of the recording device used to create them, whereas the suffix indicates the subject of the videos. The contents of these videos do not matter as much as the structure, which varies by device and manufacturer: a corruption method that breaks playability for a particular video is likely to affect different videos from the same device and potentially videos from devices by the same manufacturer. Hence, the selected videos capture the same subject across different devices: an Apple iPhone 13 mini, a Samsung Galaxy S21+, a Google Pixel 5, and a OnePlus 8T. These devices represent a diverse sampling of modern recording devices.

Table 4.1 shows the playability results for each of the files from the FloreView dataset that subjected to induced data corruption. Cases that retained full playability are indicated by the term “full,” cases that were partially playable indicate what parts of the file were affected (i.e., “no audio” or “no video”), and videos that could not be opened for playback are indicated as “unplayable.” These latter videos are of particular interest for this experiment.

Induced Corruption	**D14**	**D17**	**D34**	**D43**
Bit-flipping ($1\times$)	full	full	full	full
Bit-flipping ($10\times$)	full	full	full	full
Bit-flipping ($100\times$)	full	full	full	full
Blanking ($1 \times 4\,\mathrm{KiB}$)	full	full	full	full
Blanking ($2 \times 4\,\mathrm{KiB}$)	full	full	full	full
Blanking ($3 \times 4\,\mathrm{KiB}$)	full	no video	full	full
Truncation (2 KB)	full	no audio	no video	full
Truncation (200 KB)	unplayable	unplayable	unplayable	full
Truncation (500 KB)	unplayable	unplayable	unplayable	full

Table 4.1: Effects of induced data corruption methods on FloreView file playability

4.1.1 Bit-flipping

All videos retained full playability when subjected to bit-flipping, regardless of the severity. The only side effects of this induced corruption were increased artifacts and stream distortions. The severity of these distortions increased with the number of flipped bits.

Failure to produce a single unplayable video means that we reject Hypothesis 1 that bit-flipping renders videos unplayable with probable regularity: $p = 0$.

4.1.2 Segment Blanking

All but a single video retained full playability when subjected to blanking. Removing three 4 KiB data segments rendered D17 partially unplayable: playback of the file's video stream failed in both mpv and VLC.

In scoring probable regularity of segment blanking, the partially playable file contributes only a "half-point." So, out of 12 trials, 0.5 resulted in an unplayable outcome, yielding $p = \frac{0.5}{12} = 0.0417$. Thus, we reject Hypothesis 2 that segment blanking renders videos unplayable with probable regularity.

4.1.3 Truncation

The results of this induced corruption are more interesting. D43 retained full playability across all tested severities. D14 remained fully playable at the lowest severity, but higher severities rendered it unplayable. D17 and D34 both retained only partial playability at the lowest severity, and higher severities rendered both videos unplayable. Interestingly, D43 was the only video to remain playable across all severities.

In summary, out of 12 trials, 6 were unplayable, and 2 were only partially playable. The presence of a single unplayable video would demonstrate the possibility that truncation could result in an unplayable file. However, in this case, we find ample support for Hypothesis 3 that truncation renders MP4 files unplayable with probable regularity: $p = \frac{7}{12} = 0.583$. Given this result, our next step is to determine the underlying reason why truncation is so destructive and perhaps why some files like D43 seem resilient to it.

4.2 Preservation

4.2.1 Metadata Location

A closer inspection reveals similarities in the MP4 structure of D14, D17, and D34: the media stream data (i.e., the `mdat` box) precedes the metadata (i.e., the `moov` box) in all of these files. Placing the metadata at the end of the MP4 file leaves it vulnerable to truncation, and since the stream data cannot be decoded without the metadata, truncating these files renders them unplayable.

D43 differs in that its metadata is placed early in the file, leaving the stream data at the end: truncation affects only the trailing end of the stream data, leaving the metadata untouched. Stream data is written roughly chronologically, with data at earlier positions in the file corresponding to earlier time indices. In this case, the metadata provides the necessary information to decode the intact media data from the start. Playback simply ends where the stream data has been truncated.

4.2.2 Rationale

Appending the metadata at the end of the file appears to be a common practice, and this is likely because doing so makes the recording process more efficient. Encoded stream data can be written directly to the file as it becomes available, allowing the file to grow without adjustment, limitations, or knowing the total amount of data. Upon completion, the metadata that the encoder has generated during the recording process can be appended to the file without adjusting or recomputing any part of the file.

The contents of the `moov` atom are not known until after the recording has stopped, so how could a device produce a recording with the metadata at the beginning of the file? A closer inspection of the D43 file reveals how the OnePlus 8T likely accomplishes this: there is a `free` box between the `moov` box and the `mdat` box, which could be the result of "pre-allocating" a reserved space at the beginning for the `moov` atom. The `mdat` data is written to the file starting at a later offset in the file, and the `moov` atom is written to the pre-allocated space later. This places an upper bound on the amount of metadata, which translates to an upper bound on the duration of the video that can be recorded using this method. A larger metadata pre-allocation allows for a longer video but also results in more wasted padding space on shorter videos.

4.2.3 Preservation Methodology

As the D43 file demonstrates, placing the metadata at the beginning of the file is an effective means of preserving MP4 files in the face of truncation. This can be accomplished by pre-allocating space, as discussed above, or by relocating the `moov` atom in a later post-processing or archival phase. Relocation means recomputing offsets and shifting a significant portion of the file. Such operations may not be feasible on a recording device with limited I/O throughput, computational power, or energy resources. Performing these operations on a desktop machine or a cloud server seems more appropriate, but the files in their "as-recorded" state would be vulnerable until this conversion takes place.

Relocating metadata to the beginning of a file is actually a solution to a well-known problem in the field of media streaming. Suppose a user wishes to play a video stored as an MP4 file on a remote server. To do so, the MP4 file must be transmitted to the user's local machine, even if only temporarily. Moreover, suppose that the metadata of this MP4 file is placed at the end and that transmission of this file occurs sequentially from the start. Since the metadata is required to begin playback, the user must wait for the entire MP4 file to be transmitted before it can be played. File servers and HTTP(S) servers fall into this category, but a well-designed media streaming server could seek to the metadata, send it first, and subsequently start transmitting that so playback can begin immediately as the stream data continues in the background. This is known as "MP4 faststart." Rewriting MP4 files with their metadata at the beginning of the file permits faststart playback even if the file is transmitted sequentially.

4.2.4 "Sidecar Files:" An Alternate Approach

The MP4 specification also permits metadata to reference stream data located in another file. This means that the metadata for an MP4 file could be backed up in a "sidecar file" that accompanies the original file. If both files are undamaged, opening either file will result in the same media playback. If the original file's metadata becomes damaged due to truncation, the media can still be played by opening the sidecar file. Moreover, the original file can be recovered by restoring the metadata from the sidecar file.

This approach to MP4 preservation can be non-intuitive since users expect media files (and data files in general) to be self-contained and not depend on other files. Suppose, in the previous example, that the user realizes the original file is unplayable. Fortunately for the user, the sidecar file seems to work fine, so the user deletes the original file, mistakenly believing that both files merely contained the same media: after all, opening either file used to result in the same playback. More fundamentally, suppose the original file were intact, and the user recognized the files as "duplicates." In an effort to save disk space, the user might decide to keep only the smaller file and delete the original file. In any case, without access to the data contained in the original file, the sidecar file no longer works.

4.2.5 Experimental Outcome

We will move ahead in this study with the faststart approach to preservation. The ffmpeg utility includes a MOV/MPEG-4 muxer that supports a `+faststart` flag.[1] The presence of this flag results in an MP4 output file with the metadata at the beginning of the file. Use of this flag is demonstrated in Listing 4.1. Performing this manipulation on the D14, D17, and D34 videos results in playable files that retain their playability in the face of truncation, as shown in Table 4.2.

```
ffmpeg -i input.mp4 -c:a copy -c:v copy \
    -movflags +faststart \
    output.mp4
```

Listing 4.1: ffmpeg faststart usage example

In all cases, the faststart-modified files (columns labeled "FS" in Table 4.2) are playable before truncation. These results support Hypothesis 4 that preserved files are playable without being subject to any data corruption. Moreover, the preserved files retain full playability in all truncation cases. This supports Hypothesis 5, which validates the preservation method.

[1] `https://ffmpeg.org/ffmpeg-formats.html`

Induced Corruption	D14	D14 FS	D17	D17 FS	D34	D34 FS
Unmodified	(full)	full	(full)	full	(full)	full
Truncation (2 KB)	full	full	no audio	full	no video	full
Truncation (200 KB)	unplayable	full	unplayable	full	unplayable	full
Truncation (500 KB)	unplayable	full	unplayable	full	unplayable	full

Table 4.2: Faststart (FS) preservation effectiveness

4.3 Recovery

4.3.1 Minimum Required Data

MP4 files are composed of a selection of standard (and occasionally custom) boxes or atoms. The contents of these boxes may differ, but playability must be derived from the presence of a minimal set. We can narrow down which of these may be minimally required by intersecting the tree structures of all original files in our dataset. The detailed MP4 trees of these files and the derivation of their intersection are available in Appendix C.

In implementing the inspection capabilities of the mangler utility program, we observed three variations in the way MP4 boxes are used: empty, structural, and data-bearing. Empty boxes contained no data and typically had a type identifier of `free`. Structural boxes were always observed to contain only child boxes recursively, whereas data-bearing boxes contained raw data and no child boxes. Hence, in common tree data structure parlance, structural boxes serve as internal nodes, whereas empty and data-bearing boxes serve as leaf nodes.

Playability is derived from the data contained in these leaf nodes and, transitively, by the structure where this data is stored. Hence, we assume structural nodes are required for playability if and only if their children are required. In this experiment, we used the mangler utility to create several derived versions of the D17 video. D17 is among the simplest of the videos in the selected dataset, containing only two tracks and relatively few leaf nodes. For each derived video, the mangler removed a specific leaf node by changing the type of the target node(s) to `free` and setting all of its contained data to empty bytes (`0x00*`). Performing the removal in this way (as compared to excising the target nodes) preserves the same overall structure and data offsets and is, therefore,

Req'd?	Leaf Box Path
	`ftyp`
●	`mdat`
	`moov/mvhd`
	`moov/meta`
	`moov/trak[⋆]/tkhd`
?	`moov/trak[⋆]/mdia/mdhd`
	`moov/trak[⋆]/mdia/hdlr`
	`moov/trak[0]/mdia/minf/vmhd`
	`moov/trak[1]/mdia/minf/smhd`
	`moov/trak[⋆]/mdia/minf/dinf/dref`
●	`moov/trak[⋆]/mdia/minf/stbl/stsd`
●	`moov/trak[⋆]/mdia/minf/stbl/stts`
	`moov/trak[0]/mdia/minf/stbl/stss`
●	`moov/trak[⋆]/mdia/minf/stbl/stsz`
●	`moov/trak[⋆]/mdia/minf/stbl/stsc`

Table 4.3: Boxes required for playability as validated on D17

minimally invasive. The results of evaluating playability based on the presence (absence) of these leaf nodes are shown in Table 4.3.

The `mdat` box contains the stream data, so removing this box makes the video trivially unplayable due to the lack of any media data. The more exciting cases arise when media data is present, but the player cannot interpret it due to missing metadata under the `moov` atom and descendants.

In most cases, playability was readily apparent: either the video would play normally, or the player would fail to open the file. When the `mdhd` box was removed, the player would open the video, but the time scale was very distorted: some parts would play slowly, other parts would play quickly, and the audio and video streams were out of sync.

To summarize, an MP4 file will be playable if and only if it has an `mdat` box with readable stream data, an `moov` box containing at least one `trak` box, and all of the following metadata boxes for each `trak`: `stsd`, `stts`, `stsz`, and `stsc`. This is the minimal set of data predicted by Hypothesis 6.

File	Corruption	Playability	Missing Metadata
D14	Truncation (200 KB)	unplayable	entire moov box missing
D14	Truncation (500 KB)	unplayable	entire moov box missing
D17	Blanking (3 $\times$ 4 KiB)	no video	stts?, stsz? (video trak box only)
D17	Truncation (2 KB)	no audio	stsc, stsz? (audio trak box only)
D17	Truncation (200 KB)	unplayable	entire moov box missing
D17	Truncation (500 KB)	unplayable	entire moov box missing
D34	Truncation (2 KB)	no video	stss, stsc, stsz? (video trak box only)
D34	Truncation (200 KB)	unplayable	entire moov box missing
D34	Truncation (500 KB)	unplayable	entire moov box missing

Table 4.4: Missing required metadata from prior experimental results (includes files with partial playability). Metadata annotated with a question mark (?) indicates the presence of a required box with invalid contents.

4.3.2 Missing Required Metadata in Unplayable Files

For the case of non-trivially unplayable videos, we assume an intact mdat box and look instead for missing required metadata. Examining the unplayable videos from the investigation experiments (see Table 4.1) reveals what parts of the metadata were damaged by our induced corruption. The results of this examination are shown in Table 4.4.

4.3.3 Reconstruction of Missing Metadata

We found the missing metadata cannot be readily inferred from intact data and are therefore unable to determine the possibility for recovery (Hypothesis 7). A more detailed discussion is provided in the chapter(s) that follow. However, if we assume that such recovery is possible, the process should yield metadata that provides a valid substitute for the metadata in the original video file. Therefore, we implement a surrogate recovery method that reads the required metadata from the original file.

4.3.4 Recovery of Playability

Using the results of the surrogate reconstruction method above, we splice the "recovered" metadata into the unplayable corrupted file with the expectation that the resulting recovered file will be playable and contain the same content recognizable in the original video. The results of

File	Corruption	Corrupt State	Recovered Playability
D14	Truncation (200 KB)	unplayable	full
D14	Truncation (500 KB)	unplayable	full
D17	Blanking ($3 \times 4\,\mathrm{KiB}$)	no video	full
D17	Truncation (2 KB)	no audio	full
D17	Truncation (200 KB)	unplayable	full
D17	Truncation (500 KB)	unplayable	full
D34	Truncation (2 KB)	no video	full
D34	Truncation (200 KB)	unplayable	full
D34	Truncation (500 KB)	unplayable	full

Table 4.5: Metadata recovery effectiveness

this experiment are outlined in Table 4.5.

In all cases, restoring the missing metadata resulted in fully playable files, and the media presented during playback of the restored files were identical to those of the originals. This lends complete support to Hypothesis 8.

5. DISCUSSION

5.1 Truncation: The Likely Cause of Unplayable MP4 Files

Of all the methods tested under the corruption model, only truncation satisfied the requirements of rendering MP4 files unplayable with probable regularity. Further investigation revealed the underlying reason *why* the files become unplayable and established minimum criteria for playability: namely, the presence of a `moov` box with at least one required `trak` box containing `stsd`, `stts`, `stsz`, and `stsc` boxes. Truncation becomes problematic for files whose metadata is appended at the end of the file, and this structure appears to be common for many different recording devices.

The relative size of MP4 metadata is very small compared to the referenced media data contained in the `mdat` box. For example, the D17 file occupies roughly 71 MB of space, but its metadata constitutes only 21 KB of that space. Therefore, the induced corruption methods that involve randomization are vastly more likely to affect media data than metadata. As demonstrated by the results of our experiments, the media data is quite robust against corruption. The files remained playable in these cases, but the audio and video playback would become progressively more distorted when subjected to higher levels of corruption.

Note that it is conceivable, albeit unlikely, that the other corruption methods may render a file unplayable if this critical data is affected. We were fortunate to capture such a case in the 3×4 KiB blanking results for D17. This file occupies over 17,000 blocks, only six of which are metadata (potentially fewer contain any critical required data).

5.2 Effective Preservation Methods

As stated above, truncation has a devastating effect on files with metadata located at the end of the file. Relocating this metadata to the beginning of the file successfully preserves a file's playability even when subjected to significant amounts of truncation. This preservation method also happens to solve another known problem when playing MP4 files that are stored on remote systems. Loading the metadata ahead of the media data allows media players to begin playback immediately while the file transfer continues; hence the term “faststart.”

Another preservation technique could be to create “sidecar” files to replicate the metadata in the original file. These files would be valid MP4 files since the specification permits referencing media data stored in a separate file. However, these files could confuse users who are unaware of their dependent structure or intended purpose.

5.3 Metadata Reconstruction Challenges

After determining what metadata needed to be reconstructed to restore playability, we came to the disappointing realization that doing so requires much more effort to determine whether it is possible and, if so, how. The first challenge arises from the nature of the “sample table” boxes:

- `stsd` — *Sample Description Table*. Each entry contains metadata about “samples,” which are the smallest unit of stream data. The format of each sample depends on the stream type, and descriptor info must be looked up in the `stsc` table first.
- `stts` — *Time-to-Sample Table*. Each entry provides a mapping from a time index to a sample number.
- `stsz` — *Sample Size Table*. Each entry describes the size of each sample.
- `stsc` — *Sample-to-Chunk Table*. Samples are grouped into “chunks,” and this table contains the metadata for looking up descriptor info in the `stsd` table.

Unfortunately, none of this information is readily discernable from the contents of the `mdat` box. Moreover, the stream data typically consists of variable-sized packets representing samples and chunks described above. Each packet belongs to a different stream and is almost always interleaved with packets from other streams. The contents of these packets are indistinguishable from random noise due to their encoding, which almost always involves compression, so their boundaries are extremely difficult, if not impossible, to detect. Recovering metadata in these circumstances would be akin to uncompressing multiple compressed files with their contents intertwined and without a decompression index.

This brings up a second challenge, which is determining how many tracks are represented in the media data. An MP4 file can contain multiple streams, and the data for each stream is represented in the `mdat` atom. Even if the packet boundaries were known, we would have to "defragment" the stream data into the correct number and type of `trak` atoms.

5.4 A Hopeful Case for Recovery

Fortunately, if future research uncovers a way to reconstruct this vital metadata from partially intact media stream data, we have demonstrated that adding this reconstructed metadata to the corrupted files will fully restore the playability of the original contents. In addition, recovery of this data may be possible under ideal controlled circumstances: for example, an MP4 file with a single stream using a known codec and known stream parameters (e.g., bitrate and resolution), but even this requires more in-depth research and advanced techniques that are beyond the scope of this study.

6. SUMMARY AND CONCLUSION

The motivation for this research is the observation that MP4 files might become unplayable spontaneously over time due to data corruption. After external "black box" probing using a model for data corruption, we have determined that truncation is responsible for this spontaneous unplayability. Moreover, we find the underlying reason for this phenomenon is that many recording devices append the media metadata to the end of the file. This metadata is required for playability, and its position at the end of the file leaves it vulnerable to being damaged or completely stripped from the file due to truncation.

Armed with these findings, we take advantage of the flexibility afforded by the MP4 specification and relocate this metadata to the beginning of the file. This restructuring not only addresses the "faststart" problem but also effectively prevents truncation from rendering these media files unplayable. In this structure, the media data at the end of the file may be truncated, but the vital metadata remains intact. In this state, the files retain their playability, and playback is unaffected up to the point of truncation.

After opening the "black box" and probing the specific structure of MP4 media files, we identified the data structures MP4 players require for playback. Damaging this information will affect the file's playability, and in the event of its loss, restoring this data will also restore playability.

6.1 Future work

MP4 files are highly complex, and the encoded media data they contain is even more so: the modern audio and video codecs used by the MP4 container specification are very efficient and highly compressed. The accompanying metadata provides media players with the necessary details to decode this compressed data into something we can see and/or hear.

More research is needed to determine whether and, if so, how this metadata can be reconstructed by looking at the intact media data. Doing so would be akin to uncompressing a compressed file without its accompanying index or decrypting an encrypted file without its key.

MP4 files represent a single format used for storing media data. Additional research could help determine whether other media container formats, such as VP9 and Matroska MKV, have similar weaknesses or are prone to the same data corruption problems. If so, perhaps the preservation method identified in this research or similar methods could apply to them.

REFERENCES

[1] Apple Computer, Inc. “Classic Version of the QuickTime File Format Specification.” (Mar. 1, 2001), [Online]. Available: `https://developer.apple.com/standards/classic-quicktime/` (visited on 03/28/2023).

[2] “Information technology – Coding of audio-visual objects – Part 1: Systems,” International Organization for Standardization, Geneva, CH, Standard, Oct. 2001. [Online]. Available: `https://www.iso.org/standard/34903.html` (visited on 04/14/2023).

[3] Moving Picture, Audio, and Data Coding by Artificial Intelligence (MPAI) Community. “MPEG: The Moving Picture Experts Group [archive].” (Mar. 31, 2023), [Online]. Available: `https://mpeg.chiariglione.org` (visited on 04/14/2023).

[4] Moving Picture, Audio, and Data Coding by Artificial Intelligence (MPAI) Community. “MPAI.community.” (2021), [Online]. Available: `https://mpai.community` (visited on 04/14/2023).

[5] “Information technology – Coding of audio-visual objects – Part 12: ISO base media file format,” International Organization for Standardization, Geneva, CH, Standard, Jan. 2022. [Online]. Available: `https://www.iso.org/standard/83102.html` (visited on 04/14/2023).

[6] “Information technology – Coding of audio-visual objects – Part 14: MP4 file format,” International Organization for Standardization, Geneva, CH, Standard, Jan. 2020. [Online]. Available: `https://www.iso.org/standard/79110.html` (visited on 04/14/2023).

[7] “Information technology – Coding of audio-visual objects – Part 15: Carriage of network abstraction layer (NAL) unit structured video in the ISO base media file format,” International Organization for Standardization, Geneva, CH, Standard, Oct. 2022. [Online]. Available: `https://www.iso.org/standard/83102.html` (visited on 04/14/2023).

[8] Apple Inc. “QuickTime File Format Specification.” (Sep. 13, 2016), [Online]. Available: https://developer.apple.com/library/archive/documentation/QuickTime/QTFF/QTFFPreface/qtffPreface.html (visited on 03/28/2023).

[9] L. Zhao and L. Guan, “An optimized method and implementation for parsing MP4 metadata,” in *2010 IEEE International Conference on Progress in Informatics and Computing*, vol. 2, 2010, pp. 984–987. DOI: 10.1109/PIC.2010.5687866. (visited on 03/07/2024).

[10] L. Fegaras, “The Joy of SAX,” in *XIME-P*, Citeseer, 2004, pp. 61–66.

[11] V. Cerf and R. Kahn, “A protocol for packet network intercommunication,” *IEEE Transactions on Communications*, vol. 22, no. 5, pp. 637–648, 1974. DOI: 10.1109/TCOM.1974.1092259. (visited on 08/04/2024).

[12] M. K. Padmanaban and J. Ilow, “Evaluating packet erasure recovery techniques for video streaming,” in *2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2015, pp. 1115–1119. DOI: 10.1109/CCECE.2015.7129430. (visited on 03/16/2024).

[13] S. Gibson, *SpinRite: "What’s under the hood"*, 1993. [Online]. Available: https://www.grc.com/files/technote.pdf (visited on 02/07/2023).

[14] J. Gray and C. van Ingen, *Empirical measurements of disk failure rates and error rates*, Dec. 2005. arXiv: cs/0701166 [cs.DB]. (visited on 10/28/2024).

[15] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, “RAID: high-performance, reliable secondary storage,” *ACM Comput. Surv.*, vol. 26, no. 2, pp. 145–185, Jun. 1994, ISSN: 0360-0300. DOI: 10.1145/176979.176981. (visited on 08/16/2024).

[16] J. Bonwick and B. Moore, “ZFS, the last word in file systems,” 2008. [Online]. Available: https://www.snia.org/sites/default/orig/sdc_archives/2008_presentations/monday/JeffBonwick-BillMoore_ZFS.pdf (visited on 08/16/2024).

[17] Oracle, "Checking ZFS File System Integrity," in *Oracle Solaris ZFS Administration Guide*. 2006. [Online]. Available: `https://docs.oracle.com/cd/E18752_01/html/819-5461/gbbwa.html` (visited on 05/05/2023).

[18] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-Tree Filesystem," *ACM Trans. Storage*, vol. 9, no. 3, Aug. 2013, ISSN: 1553-3077. DOI: `10.1145/2501620.2501623`.

[19] A. Pal and N. Memon, "The evolution of file carving," *IEEE Signal Processing Magazine*, vol. 26, no. 2, pp. 59–71, 2009. DOI: `10.1109/MSP.2008.931081`. (visited on 03/17/2024).

[20] R. Poisel and S. Tjoa, "A comprehensive literature review of file carving," in *2013 International Conference on Availability, Reliability and Security*, 2013, pp. 475–484. DOI: `10.1109/ARES.2013.62`. (visited on 03/17/2024).

[21] G. G. Richard III and V. Roussev, "Scalpel: A frugal, high performance file carver.," in *DFRWS*, 2005.

[22] R. Poisel and S. Tjoa, "Roadmap to approaches for carving of fragmented multimedia files," in *2011 Sixth International Conference on Availability, Reliability and Security*, 2011, pp. 752–757. DOI: `10.1109/ARES.2011.118`. (visited on 03/22/2024).

[23] E. Casey and R. Zoun, "Design tradeoffs for developing fragmented video carving tools," *Digital Investigation*, vol. 11, S30–S39, 2014, Fourteenth Annual DFRWS Conference, ISSN: 1742-2876. DOI: `10.1016/j.diin.2014.05.010`. (visited on 08/21/2024).

[24] G.-H. Na, K.-S. Shim, K.-W. Moon, S. G. Kong, E.-S. Kim, and J. Lee, "Frame-based recovery of corrupted video files using video codec specifications," *IEEE Transactions on Image Processing*, vol. 23, no. 2, pp. 517–526, 2014. DOI: `10.1109/TIP.2013.2285625`. (visited on 02/07/2023).

[25] K. Alghafli and T. Martin, "Identification and recovery of video fragments for forensics file carving," in *2016 11th International Conference for Internet Technology and Secured*

Transactions (ICITST), 2016, pp. 267–272. DOI: 10.1109/ICITST.2016.7856710. (visited on 03/17/2024).

[26] S. Godsill and P. Rayner, "Frequency-based interpolation of sampled signals with applications in audio restoration," in *1993 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 1, 1993, 209–212 vol.1. DOI: 10.1109/ICASSP.1993.319092. (visited on 02/09/2023).

[27] L. Simon, J. C. Valiere, and C. Depollier, "New contribution on noise reduction using wavelet techniques: Application to the restoration of old recordings," in *Audio Engineering Society Convention 94*, Mar. 1993. [Online]. Available: http://www.aes.org/e-lib/browse.cfm?elib=6675 (visited on 02/09/2023).

[28] J. B. Weaver, Y. Xu, D. M. Healy Jr., and L. D. Cromwell, "Filtering noise from images with wavelet transforms," *Magnetic Resonance in Medicine*, vol. 21, no. 2, pp. 288–295, 1991. DOI: https://doi.org/10.1002/mrm.1910210213. (visited on 02/09/2023).

[29] A. M. John, K. Khanna, R. R. Prasad, and L. G. Pillai, "A review on application of fourier transform in image restoration," in *2020 Fourth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, 2020, pp. 389–397. DOI: 10.1109/I-SMAC49090.2020.9243510. (visited on 02/09/2023).

[30] G. Zhang and R. Stevenson, "Efficient error recovery for multiple description video coding," in *2004 International Conference on Image Processing, 2004. ICIP '04.*, vol. 2, 2004, 829–832 Vol.2. DOI: 10.1109/ICIP.2004.1419427. (visited on 03/16/2024).

[31] H.-b. Chen, Y.-n. Dong, and H. Shi, "Data hiding based error recovery for h.264 video streaming over wireless networks," in *2011 Visual Communications and Image Processing (VCIP)*, 2011, pp. 1–4. DOI: 10.1109/VCIP.2011.6115931. (visited on 03/16/2024).

[32] G. M. P. D., B. Madathil, and S. N. George, "Entropy-based reweighted tensor completion technique for video recovery," *IEEE Transactions on Circuits and Systems for Video Tech-*

nology, vol. 30, no. 2, pp. 415–426, 2020. DOI: 10.1109/TCSVT.2019.2892848. (visited on 03/16/2024).

[33] P. M. Hoang, H. D. Tuan, T. T. Son, and H. V. Poor, "Qualitative HD Image and Video Recovery via High-Order Tensor Augmentation and Completion," *IEEE Journal of Selected Topics in Signal Processing*, vol. 15, no. 3, pp. 688–701, 2021. DOI: 10.1109/JSTSP.2020.3042063. (visited on 03/16/2024).

[34] T. Gloe, A. Fischer, and M. Kirchner, "Forensic analysis of video file formats," in *Digital Investigation*, Proceedings of the First Annual DFRWS Europe, vol. 11, 2014, S68–S76. DOI: 10.1016/j.diin.2014.03.009. (visited on 03/16/2024).

[35] D. Shullani, M. Fontani, M. Iuliani, O. A. Shaya, and P. Alessandro, "VISION: a video and image dataset for source identification," *EURASIP Journal on Information Security*, vol. 2017, no. 1, p. 15, Oct. 2017, ISSN: 2510-523X. DOI: 10.1186/s13635-017-0067-2. (visited on 03/12/2024).

[36] D. Baracchi, D. Shullani, M. Iuliani, and A. Piva, "FloreView: An Image and Video Dataset for Forensic Analysis," *IEEE Access*, vol. 11, pp. 109 267–109 282, 2023. DOI: 10.1109/ACCESS.2023.3321991. (visited on 03/12/2024).

[37] P. Yang, D. Baracchi, M. Iuliani, *et al.*, "Efficient Video Integrity Analysis Through Container Characterization," *IEEE Journal of Selected Topics in Signal Processing*, vol. 14, no. 5, pp. 947–954, 2020. DOI: 10.1109/JSTSP.2020.3008088. (visited on 03/08/2024).

[38] S. Milani, M. Fontani, P. Bestagini, *et al.*, "An overview on video forensics," *APSIPA Transactions on Signal and Information Processing*, vol. 1, e2, 2012. DOI: 10.1017/ATSIP.2012.2. (visited on 03/16/2024).

[39] M. S. Rana, M. N. Nobi, B. Murali, and A. H. Sung, "Deepfake detection: A systematic literature review," *IEEE Access*, vol. 10, pp. 25 494–25 513, 2022. DOI: 10.1109/ACCESS.2022.3154404. (visited on 08/04/2024).

[40] Z. Xiang, J. Horváth, S. Baireddy, P. Bestagini, S. Tubaro, and E. J. Delp, “Forensic Analysis of Video Files Using Metadata,” in *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2021, pp. 1042–1051. DOI: `10.1109/CVPRW53098.2021.00115`. (visited on 03/05/2024).

[41] Z. Xiang, A. K. Singh Yadav, P. Bestagini, S. Tubaro, and E. J. Delp, “MTN: Forensic Analysis of MP4 Video Files Using Graph Neural Networks,” in *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2023, pp. 963–972. DOI: `10.1109/CVPRW59228.2023.00103`. (visited on 03/05/2024).

[42] S. Hemalatha, U. D. Acharya, and Shamathmika, “MP4 video steganography in wavelet domain,” in *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 2017, pp. 1229–1235. DOI: `10.1109/ICACCI.2017.8126010`. (visited on 03/05/2024).

[43] O. Tange, *GNU Parallel 20241022 (’Sinwar Nasrallah’)*, GNU Parallel is a general parallelizer to run multiple serial command line programs in parallel without changing them., Oct. 2024. DOI: `10.5281/zenodo.13957646`. (visited on 11/01/2024).

APPENDIX A

ADDITIONAL MP4 METADATA STRUCTURE

A.0.1 Movie Metadata: The `moov` Box/Atom

As with the details already stated, this reference draws heavily from Apple's QTFF Specification [8]. Whereas the previously presented information only discussed general format structures that are practically identical between QTFF and MP4, there may be slight variations between the two formats with regard to the details presented here. Hence, this section will use Apple's "atom" terminology rather than the MP4 standard's "box" term.

The `moov` atom serves as a container for other types of atoms, each of which contains additional metadata about the movie represented in a file:

- Profile (`prfl`) - deprecated
- Movie header (`mvhd`) - required
- Clipping (`clip`)
- Track (`trak`) - at least one must be present
- User data (`udta`)
- Color table (`ctab`)
- Compressed Movie (`cmov`) - used when the movie is stored in compressed format
- Reference movie (`rmra`) - used for movies that reference data in separate files

A.0.2 Tracks: The `trak` Atom

Of these possible child atom types, `trak` atoms are the most relevant since they specify the number and types of each media stream contained in the file. Each `trak` atom in turn contains the following types of child atoms:

- Profile (`prfl`) - deprecated
- Track header (`tkhd`) - required
- Track Aperture Mode Dimensions (`tapt`)
- Clipping (`clip`)
- Track matte (`matt`)
- Edit (`edts`)
- Track reference (`tref`)
- Track exclude from autoselection (`txas`)
- Track loading settings (`load`)
- Track input map (`imap`)
- Media (`mdia`) - required
- User data (`udta`)

A.0.3 Streams: The `mdia` Atom

The `mdia` atom brings us closer to understanding the actual stream referenced by the track. This type of atom contains the following additional types of atoms:

- Media header (`mdhd`) - required
- Extended language tag (`elng`)
- Handler reference (`hdlr`) - determines the kind and how media data is to be interpreted
- Media information (`minf`) - handler-specific information
- User data (`udta`)

A.0.4 Raw Codec Metadata: The `minf` Atom

For the purposes of this research, we focus only on video `minf` atoms:

- Video media information (`vmhd`) - required
- Handler reference (`hdlr`) - required
- Data information (`dinf`) - used for data reference
- Sample table (`stbl`)

APPENDIX B

SELECTED SOURCE CODE FOR MANGLER UTILITY

The “mangler” utility for modifying and inspecting MP4 video files was developed in Rust and exists as a public GitHub repository at `https://github.com/komputerwiz/mp4-mangler`.

The source code is too long to include in its entirety, so the relevant parts pertaining to the binary file modification are included here. The code listings in this chapter use the 2021 edition of the Rust programming language, compile on version 1.80.1 of the `rustc` compiler, and depend on the following crates and corresponding versions:

- `log 0.4.20`
- `mp4 0.14.0`
- `rand 0.8.5`

All other imports are from the Rust standard library.

B.1 Data Corruption Methods

B.1.1 Bit-Flipping

```
use std::fs::OpenOptions;
use std::io;
use std::os::unix::prelude::FileExt;
use std::path::Path;

use rand::{self, distributions::Uniform, prelude::Distribution};

/// flips random bits in the given file
pub fn flip_bits(file: &Path, num_bits: usize) -> io::Result<()> {
  log::info!("opening file {}", file.to_str().unwrap_or("???"));
  let f = OpenOptions::new().read(true).write(true).open(file)?;
  let size = f.metadata()?.len();

  let mut rng = rand::thread_rng();
  let size_dist = Uniform::new(0, size);
  let bit_dist = Uniform::new(0, 8);

  for _ in 0..num_bits {
    // read byte at random byte offset
    let byte_offset = size_dist.sample(&mut rng);
    let mut buf: [u8; 1] = Default::default();
    f.read_exact_at(&mut buf, byte_offset)?;

    let orig = buf[0];

    // flip bit at random bit offset
    let bit_offset = bit_dist.sample(&mut rng);
    buf[0] ^= 1 << bit_offset;
    println!("Flipped bit {} at offset 0x{:x}: 0x{:02x} => 0x{:02x}",
      8 - bit_offset,
      byte_offset,
      orig,
      buf[0]);

    // write byte back to file
    f.write_at(&buf, byte_offset)?;
  }

  Ok(())
}
```

B.1.2 Segment Blanking

```rust
use std::fs::OpenOptions;
use std::io;
use std::os::unix::prelude::FileExt;
use std::path::Path;

use rand::{self, distributions::Uniform, prelude::Distribution};

/// zeroes random contiguous blocks of data
pub fn blank_blocks(
  file: &Path,
  num_blocks: usize,
  block_size_bytes: u64
) -> io::Result<()> {
  log::info!("opening file {}", file.to_str().unwrap_or("???"));
  let f = OpenOptions::new().read(true).write(true).open(file)?;
  let size = f.metadata()?.len();

  let bad_block = vec![0; block_size_bytes as usize];
  let size_in_blocks = size / block_size_bytes;

  let mut rng = rand::thread_rng();
  let block_dist = Uniform::new(0, size_in_blocks);

  for _ in 0..num_blocks {
    let block_offset = block_dist.sample(&mut rng);

    let end = if block_offset == size_in_blocks - 1 {
      // don't go beyond end of last block
      size % block_size_bytes
    } else {
      block_size_bytes
    } as usize;

    println!("Blanked block {} ({} bytes starting at offset 0x{:x})",
      block_offset,
      block_size_bytes,
      block_offset * block_size_bytes);

    f.write_at(&bad_block[0..end], block_offset * block_size_bytes)?;
  }

  Ok(())
}
```

B.1.3 Truncation

```rust
/// truncates the given number of bytes from the end of the file.
pub fn truncate(file: &Path, num_bytes: u64) -> io::Result<()> {
  log::info!("opening file {}", file.to_str().unwrap_or("???"));
  let f = OpenOptions::new().read(true).write(true).open(file)?;
  let size = f.metadata()?.len();

  f.set_len(size - num_bytes)?;

  println!("Truncated {} bytes from end of file.", num_bytes);

  Ok(())
}
```

B.2 Box/Atom Stripping

The following code depends on the parsing framework in Section B.2.1 and is used for finding the minimum required data for playability.

```rust
use std::fs::File;
use std::io::{self, Write};
use std::path::Path;

use mp4::{BoxHeader, BoxType};

/// strips the given box type(s) from the given input file
/// and writes the resulting MP4 to the given output file
pub fn strip(
  input: &Path,
  output: &Path,
  ignore: Vec<BoxType>
) -> io::Result<()> {
  let in_file = File::open(input)?;
  let in_file_size = in_file.metadata()?.len();
  let reader = io::BufReader::new(in_file);

  let out_file = File::create(output)?;
  let mut writer = io::BufWriter::new(out_file);

  let mut visitor = StripVisitor::new(&mut writer, ignore);
  read_box(reader, in_file_size, &mut visitor)?;

  Ok(())
}

struct Mp4Box {
  name: BoxType,
  data: BoxData,
}

impl Mp4Box {
  pub fn new(name: BoxType) -> Self {
    Self {
      name,
      data: BoxData::Empty,
    }
  }

  pub fn write_to(
    &self,
    writer: &mut impl io::Write
  ) -> io::Result<u64> {
    let mut data_buf = Vec::new();
    self.data.write_to(&mut data_buf)?;

    let mut size = 8 + data_buf.len() as u64;
    if size > u32::MAX as u64 {
      size += 8;
    }

    let name_id: u32 = self.name.into();

    if size > u32::MAX as u64 {
      writer.write_all(&1u32.to_be_bytes())?;
      writer.write_all(&name_id.to_be_bytes())?;
      writer.write_all(&size.to_be_bytes())?;
    } else {
      writer.write_all(&(size as u32).to_be_bytes())?;
      writer.write_all(&name_id.to_be_bytes())?;
    }

    writer.write_all(&data_buf)?;

    Ok(size)
  }
}

enum BoxData {
  Empty,
  Raw(Vec<u8>),
  Children(Vec<Mp4Box>),
}

impl BoxData {
  pub fn write_to(
    &self,
    writer: &mut impl io::Write
  ) -> io::Result<u64> {
    match self {
      Self::Empty => Ok(0),
      Self::Raw(bytes) => {
        writer.write_all(&bytes)?;
        Ok(bytes.len() as u64)
      },
      Self::Children(children) => {
        let mut sum = 0;
        for child in children {
          sum += child.write_to(writer)?;
        }
        Ok(sum)
      }
    }
  }
}

struct StripVisitor<'a> {
  writer: &'a mut dyn io::Write,
  ignore: Vec<BoxType>,
  stack: Vec<Mp4Box>,
}

impl<'a> StripVisitor<'a> {
  fn new(
    writer: &'a mut impl io::Write,
    ignore: Vec<BoxType>
  ) -> Self {
    Self {
      writer,
      ignore,
      stack: Vec::new(),
    }
  }

}

impl<'a> Mp4Visitor for StripVisitor<'a> {
  fn start_box(
    &mut self,
    header: &BoxHeader,
    corrected_size: Option<u64>
  ) -> io::Result<()> {
    if let Some(size) = corrected_size {
      log::warn!("Correcting size mismatch in {} box: header says {} B, but should actually be {} B", header.name, header.size, size);
    }

    // maintain context for all box types
    // the "ignore" step will happen later
    self.stack.push(Mp4Box::new(header.name));

    Ok(())
  }

  fn data(&mut self, reader: &mut impl io::Read) -> io::Result<()> {
    let current_box = self.stack.last_mut().unwrap();

    if self.ignore.contains(&current_box.name) {
      log::info!("skipping data for ignored box type {}", current_box.name);
      return Ok(());
    }

    // Attempt to do some recovery in the form of
    // box offset/size consistency adjustments
    let mut data: Vec<u8> = Vec::new();
    match current_box.name {
      BoxType::CttsBox => {
        let mut version_flags = [0u8; 4];
        // version (1 B) and flags (3 B)
        reader.read_exact(&mut version_flags)?;

        // entry count (u32)
        let mut entry_count_bytes = [0u8; 4];
        reader.read_exact(&mut entry_count_bytes)?;
        let entry_count = u32::from_be_bytes(entry_count_bytes);

        // read entries ({sample_count: u32, composition_offset: u32})
        let mut entries = Vec::new();
        let mut entry_bytes = [0u8; 8];
        while let Ok(_) = reader.read_exact(&mut entry_bytes) {
          entries.push(entry_bytes);
        }

        if entry_count as usize != entries.len() {
          log::warn!("correcting ctts table length: metadata says {} entries, but should actually be {} entries", entry_count, entries.len());
        }

        data.write(&version_flags)?;
        data.write(&(entries.len() as u32).to_be_bytes())?;
        for entry in entries {
          data.write(&entry)?;
        }
      },

      BoxType::StszBox => {
        let mut version_flags = [0u8; 4];
        // version (1 B) and flags (3 B)
        reader.read_exact(&mut version_flags)?;

        // sample size (u32)
        let mut sample_size_bytes = [0u8; 4];
        reader.read_exact(&mut sample_size_bytes)?;
        let sample_size = u32::from_be_bytes(sample_size_bytes);

        // entry count (u32)
        let mut entry_count_bytes = [0u8; 4];
        reader.read_exact(&mut entry_count_bytes)?;
        let entry_count = u32::from_be_bytes(entry_count_bytes);

        let mut entries = Vec::new();
        if sample_size == 0 {
          // samples have variable sizes that are stored in the table
          // read entries ({sample_size: u32})
          let mut entry_bytes = [0u8; 4];
          while let Ok(_) = reader.read_exact(&mut entry_bytes) {
            entries.push(entry_bytes);
          }
        }

        if entry_count as usize != entries.len() {
          log::warn!("correcting stsz table length: metadata says {}
  entries, but should actually be {} entries", entry_count, entries.
  len());
        }

        data.write(&version_flags)?;
        data.write(&sample_size_bytes)?;
        data.write(&(entries.len() as u32).to_be_bytes())?;
        for entry in entries {
          data.write(&entry)?;
        }
      },

      BoxType::StcoBox => {
        let mut version_flags = [0u8; 4];
        // version (1 B) and flags (3 B)
        reader.read_exact(&mut version_flags)?;

        // entry count (u32)
        let mut entry_count_bytes = [0u8; 4];
        reader.read_exact(&mut entry_count_bytes)?;
        let entry_count = u32::from_be_bytes(entry_count_bytes);

        // read entries ({chunk_offset: u32})
        let mut entries = Vec::new();
        let mut entry_bytes = [0u8; 4];
        while let Ok(_) = reader.read_exact(&mut entry_bytes) {
          entries.push(entry_bytes);
        }

        if entry_count as usize != entries.len() {
          log::warn!("correcting stco table length: metadata says {} entries, but should actually be {} entries", entry_count, entries.len());
        }

        data.write(&version_flags)?;
        data.write(&(entries.len() as u32).to_be_bytes())?;
        for entry in entries {
          data.write(&entry)?;
        }
      },

      BoxType::StscBox => {
        let mut version_flags = [0u8; 4];
        // version (1 B) and flags (3 B)
        reader.read_exact(&mut version_flags)?;

        // entry count (u32)
        let mut entry_count_bytes = [0u8; 4];
        reader.read_exact(&mut entry_count_bytes)?;
        let entry_count = u32::from_be_bytes(entry_count_bytes);

        // read entries ({first_chunk: u32, samples_per_chunk: u32, sample_description_id: u32})
        let mut entries = Vec::new();
        let mut entry_bytes = [0u8; 12];
        while let Ok(_) = reader.read_exact(&mut entry_bytes) {
          entries.push(entry_bytes);
        }

        if entry_count as usize != entries.len() {
          log::warn!("correcting stsc table length: metadata says {} entries, but should actually be {} entries", entry_count, entries.len());
        }

        data.write(&version_flags)?;
        data.write(&(entries.len() as u32).to_be_bytes())?;
        for entry in entries {
          data.write(&entry)?;
        }
      },

      BoxType::SttsBox => {
        let mut version_flags = [0u8; 4];
        // version (1 B) and flags (3 B)
        reader.read_exact(&mut version_flags)?;

        // entry count (u32)
        let mut entry_count_bytes = [0u8; 4];
        reader.read_exact(&mut entry_count_bytes)?;
        let entry_count = u32::from_be_bytes(entry_count_bytes);

        // read entries ({sample_count: u32, sample_duration: u32})
```

```
        let mut entries = Vec::new();
        let mut entry_bytes = [0u8; 8];
        while let Ok(_) = reader.read_exact(&mut entry_bytes) {
          entries.push(entry_bytes);
        }

        if entry_count as usize != entries.len() {
          log::warn!("correcting stts table length: metadata says {} entries, but should actually be {} entries", entry_count, entries.len());
        }

        data.write(&version_flags)?;
        data.write(&(entries.len() as u32).to_be_bytes())?;
        for entry in entries {
          data.write(&entry)?;
        }
      },

      // copy all other box types verbatim
      _ => {
        io::copy(reader, &mut data)?;
      },
    }

    current_box.data = BoxData::Raw(data);

    Ok(())
  }

  fn end_box(&mut self, _typ: &mp4::BoxType) -> io::Result<()> {
    if let Some(exit_box) = self.stack.pop() {
      if self.ignore.contains(&exit_box.name) {
        log::info!("skipping ignored box type {}", exit_box.name);
        return Ok(())
      }

      if let Some(parent_box) = self.stack.last_mut() {
        // "write" data to parent box
        match &mut parent_box.data {
          BoxData::Empty => {
            parent_box.data = BoxData::Children(vec![exit_box]);
          },
          BoxData::Raw(_) => {
            log::warn!("overriding raw data in parent with child node");
            parent_box.data = BoxData::Children(vec![exit_box]);
          },
          BoxData::Children(children) => {
            children.push(exit_box)
          }
        }
      } else {
        // exiting root; write data to file
        exit_box.write_to(&mut self.writer)?;
      }
    }

    Ok(())
    }
}
```

B.2.1 MP4 Parsing Framework

The following code takes inspiration from the method proposed by Zhao and Guan [9] and by the event-based SAX XML parser for Java [10].

```rust
use std::io::{self, Read, Seek, ErrorKind, SeekFrom};
use mp4::{BoxHeader, BoxType};

/// A SAX-style visitor/parser for reading MP4 boxes
pub trait Mp4Visitor {
  fn start_box(
    &mut self,
    _header: &BoxHeader,
    _corrected_size: Option<u64>
  ) -> io::Result<()> {
    Ok(())
  }

  fn data(
    &mut self,
    _reader: &mut impl Read
  ) -> io::Result<()> {
    Ok(())
  }

  fn end_box(
    &mut self,
    _typ: &BoxType
  ) -> io::Result<()> {
    Ok(())
  }
}

pub fn read_box<R: Read + Seek>(
  mut reader: R,
  end: u64,
  visitor: &mut impl Mp4Visitor
) -> io::Result<R> {
  // A box is simply a header followed by content.
  // The header includes the size (in bytes) and type of the box, and
  // has 2 different forms depending on the size:
  //
  // if size <= u32::MAX  (8-byte header),
  //
  //   box {
  //     header {
  //       size: u32,       // big endian
  //       type: [u8; 4],   // four ASCII chars
  //     },
  //     content: [u8; header.size],
  //   }
  //
  // if size > u32::MAX  (16-byte header),
  //
  //   header {
  //     size: u32,        // big endian, set to 1
  //     type: [u8; 4],    // four ASCII chars
  //     largesize: u64,  // big endian
  //   },
  //   content: [u8; header.largesize]
  //
  // The box size declared in the header includes the size of the
  // header itself, so the header of an empty box (i.e., with no
  // content) will have a declared size of 8 bytes

  // The stream is currently positioned at the start of the header,
  // and `current` captures this position.
  //
  // `end` captures the end of the current context, which could be the
  // file itself or the parent box.
  //
  //   .- current                     end -.
  //   |                                    |
  //   | header | content | other stuff... |
  //   ^
  let mut current = reader.stream_position()?;

  // Boxes are composite, meaning the contents of a box can be
  // additional (sub-)boxes. Hence, read them iteratively and
  // recursively to catch all of them.
  while current < end {
    log::debug!("reading box header");
    let header = read_header(&mut reader)?;

    // The stream is now positioned at the start of the content.
    // In a corrupted file, the size declared in the header could
    // potentially overflow the end.
    // The following situation is expected and desirable...
    //
    //   .- current         .- box_end   end -.
    //   |                  |                  |
    //   | header | content | other stuff...   |
    //            ^
    // But we want to catch the following case to correct for it:
    //
    //   .- current   end? -.  box_end? -.
    //   |                  |            |
    //   | header | content |      oops! |
    //            ^
    let mut box_end = current + header.size;
    let corrected_size = if box_end > end {
      log::error!("declared box size overflows container by {} B",
   box_end - end);
      box_end = end;
      Some(box_end - current)
    } else {
      None
    };

    visitor.start_box(&header, corrected_size)?;

    match header.name {
      BoxType::FreeBox
        | BoxType::ElstBox
        | BoxType::FtypBox
```

```
        | BoxType::HdlrBox
        | BoxType::MdatBox
        | BoxType::MdhdBox
        | BoxType::MvhdBox
        | BoxType::TkhdBox
        | BoxType::VmhdBox
        | BoxType::DrefBox
        | BoxType::StsdBox
        // ...
        => {
          // limit visitor's reader to just the contents of this box
          let content_start = reader.stream_position()?;
          let mut sub_reader = reader.take(box_end - content_start);
          visitor.data(&mut sub_reader)?;
          reader = sub_reader.into_inner();

          // skip to the end of this box
          log::trace!("not recursing into 'data-only' box");
          reader.seek(SeekFrom::Start(box_end))?;
        },
      _ => {
        // traverse all other boxes recursively
        reader = read_box(reader, box_end, visitor)?;
      }
    }

    visitor.end_box(&header.name)?;

    current = reader.stream_position()?;
  }

  Ok(reader)
}

fn read_header<R: Read>(reader: &mut R) -> io::Result<BoxHeader> {
  let mut buf = [0u8; 8];
  match reader.read(&mut buf) {
    Ok(sz) => {
      log::trace!("read {} bytes for header: {:0x?}", sz, buf);
      if sz == 0 {
        log::error!("reached EOF");
        return Err(std::io::Error::new(ErrorKind::UnexpectedEof, "reached EOF"));
      }
    },
    Err(e) => {
      log::error!("unable to read header: {}", e);
      Err(e)?;
    },
  };

  // Get size.
  let s = buf[0..4].try_into().unwrap();
  let size = u32::from_be_bytes(s);

  // Get box type string.
  let t = buf[4..8].try_into().unwrap();
  let typ = u32::from_be_bytes(t);

  // Get largesize if size is 1
  if size == 1 {
    reader.read_exact(&mut buf)?;
    let largesize = u64::from_be_bytes(buf);

    Ok(BoxHeader {
      name: BoxType::from(typ),

      // Subtract the length of the serialized largesize,
      // as callers assume `size - HEADER_SIZE` is the length of the
      // box data. Disallow `largesize < 16`, or else a largesize
      // of 8 will result in a BoxHeader::size
      // of 0, incorrectly indicating that the box data extends to the
      // end of the stream.
      size: match largesize {
        0 => 0,
        1..=15 => return Err(std::io::Error::new(ErrorKind::InvalidData
 , "64-bit box size too small")),
        16..=u64::MAX => largesize - 8,
      },
    })
  } else {
    Ok(BoxHeader {
      name: BoxType::from(typ),
      size: size as u64,
    })
  }
}

pub trait BoxHeaderExt {
  fn set_size(&mut self, content_size: u64);
}

impl BoxHeaderExt for BoxHeader {
  fn set_size(&mut self, content_size: u64) {
    self.size = content_size + 8;

    if self.size > u32::MAX as u64 {
      // writing will need to use longsize variant
      self.size += 8;
    }
  }
}
```

APPENDIX C

MP4 TREES FOR EXPERIMENTAL DATASET

The listings in this chapter contain textual representations of the MP4 box/atom trees that comprise the files used for experimentation in this study. Structures that are common to all files are indicated by an asterisk (⋆).

C.1 D14

```
ftyp (20 B) ⋆
wide (8 B)
mdat (61475099 B) ⋆
moov (26406 B) ⋆
  mvhd (108 B) ⋆
  trak (12055 B) ⋆
    tkhd (92 B) ⋆
    tapt (68 B)
      clef (20 B)
      prof (20 B)
      enof (20 B)
    edts (36 B)
      elst (28 B)
    mdia (11851 B) ⋆
      mdhd (32 B) ⋆
      hdlr (49 B) ⋆
      minf (11762 B) ⋆
        vmhd (20 B) ⋆
        hdlr (56 B) ⋆
        dinf (36 B) ⋆
          dref (28 B) ⋆
        stbl (11642 B) ⋆
          stsd (167 B) ⋆
          stts (24 B) ⋆
          ctts (6120 B)
          cslg (32 B)
          stss (120 B) ⋆
          sdtp (775 B)
          stsc (976 B) ⋆
          stsz (3072 B) ⋆
          stco (348 B)
  trak (5770 B) ⋆
    tkhd (92 B) ⋆
    edts (36 B)
      elst (28 B)
    mdia (5634 B) ⋆
      mdhd (32 B) ⋆
      hdlr (49 B) ⋆
      minf (5545 B) ⋆
        smhd (16 B) ⋆
        hdlr (56 B) ⋆
        dinf (36 B) ⋆
          dref (28 B) ⋆
        stbl (5429 B) ⋆
          stsd (159 B) ⋆
          sgpd (26 B)
          sbgp (28 B)
          stts (24 B) ⋆
          stsc (556 B) ⋆
          stsz (4412 B) ⋆
          stco (216 B)
  trak (593 B) ⋆
    tkhd (92 B) ⋆
    edts (36 B)
      elst (28 B)
    mdia (457 B) ⋆
      mdhd (32 B) ⋆
      hdlr (52 B) ⋆
      minf (365 B) ⋆
        gmhd (32 B)
          gmin (24 B)
        hdlr (56 B) ⋆
        dinf (36 B) ⋆
          dref (28 B) ⋆
        stbl (233 B) ⋆
          stsd (133 B) ⋆
          stts (24 B) ⋆
          stsc (28 B) ⋆
          stsz (20 B) ⋆
          stco (20 B) ⋆
  trak (2914 B) ⋆
    tkhd (92 B) ⋆
    edts (36 B)
      elst (28 B)
    mdia (2778 B) ⋆
      mdhd (32 B) ⋆
      hdlr (52 B) ⋆
      minf (2686 B) ⋆
        gmhd (32 B)
          gmin (24 B)
        hdlr (56 B) ⋆
        dinf (36 B) ⋆
          dref (28 B) ⋆
        stbl (2554 B) ⋆
          stsd (578 B) ⋆
          stts (120 B) ⋆
          stsc (232 B) ⋆
          stsz (1472 B) ⋆
          stco (144 B)
  trak (4326 B) ⋆
    tkhd (92 B) ⋆
    edts (36 B)
      elst (28 B)
    mdia (4190 B) ⋆
      mdhd (32 B) ⋆
      hdlr (52 B) ⋆
      minf (4098 B) ⋆
        gmhd (32 B)
          gmin (24 B)
        hdlr (56 B) ⋆
        dinf (36 B) ⋆
          dref (28 B) ⋆
        stbl (3966 B) ⋆
          stsd (602 B) ⋆
          stts (24 B) ⋆
          stsc (40 B) ⋆
          stsz (3072 B) ⋆
          stco (220 B)
  meta (632 B) ⋆
free (77430 B)
```

C.2 D17

```
ftyp (24 B) ⋆
mdat (70946324 B) ⋆
moov (20913 B) ⋆
  mvhd (108 B) ⋆
  udta (66 B)
  meta (177 B) ⋆
  trak (14743 B) ⋆
    tkhd (92 B) ⋆
    mdia (14643 B) ⋆
      mdhd (32 B) ⋆
      hdlr (44 B) ⋆
      minf (14559 B) ⋆
        vmhd (20 B) ⋆
        dinf (36 B) ⋆
          dref (28 B) ⋆
        stbl (14495 B) ⋆
          stsd (171 B) ⋆
          stts (7768 B) ⋆
          stss (120 B) ⋆
          stsz (6164 B) ⋆
          stsc (40 B) ⋆
          co64 (224 B)
  trak (5811 B) ⋆
    tkhd (92 B) ⋆
    edts (48 B)
      elst (40 B)
    mdia (5663 B) ⋆
      mdhd (32 B) ⋆
      hdlr (44 B) ⋆
      minf (5579 B) ⋆
        smhd (16 B) ⋆
        dinf (36 B) ⋆
          dref (28 B) ⋆
        stbl (5519 B) ⋆
          stsd (91 B) ⋆
          stts (328 B) ⋆
          stsz (4816 B) ⋆
          stsc (52 B) ⋆
          co64 (224 B)
```

C.3 D34

```
ftyp (28 B) ⋆
```

```
mdat (104914320 B) ⋆
moov (28745 B) ⋆
  mvhd (108 B) ⋆
  udta (37 B)
  meta (124 B) ⋆
  trak (7837 B) ⋆
    tkhd (92 B) ⋆
    mdia (7737 B) ⋆
      mdhd (32 B) ⋆
      hdlr (43 B) ⋆
      minf (7654 B) ⋆
        nmhd (12 B)
        dinf (36 B) ⋆
          dref (28 B) ⋆
        stbl (7598 B) ⋆
          stsd (58 B) ⋆
          stts (3968 B) ⋆
          stsz (3052 B) ⋆
          stsc (304 B) ⋆
          co64 (208 B)
  trak (6099 B) ⋆
    tkhd (92 B) ⋆
    mdia (5999 B) ⋆
      mdhd (32 B) ⋆
      hdlr (44 B) ⋆
      minf (5915 B) ⋆
        smhd (16 B) ⋆
        dinf (36 B) ⋆
          dref (28 B) ⋆
        stbl (5855 B) ⋆
          stsd (91 B) ⋆
          stts (472 B) ⋆
          stsz (4752 B) ⋆
          stsc (316 B) ⋆
          co64 (216 B)
  trak (14532 B) ⋆
    tkhd (92 B) ⋆
    mdia (14432 B) ⋆
      mdhd (32 B) ⋆
      hdlr (44 B) ⋆
      minf (14348 B) ⋆
        vmhd (20 B) ⋆
        dinf (36 B) ⋆
          dref (28 B) ⋆
        stbl (14284 B) ⋆
          stsd (180 B) ⋆
          stts (7360 B) ⋆
          stsz (6084 B) ⋆
          stsc (316 B) ⋆
          co64 (216 B)
          stss (120 B) ⋆
```

C.4 D43

```
ftyp (24 B) ⋆
moov (21793 B) ⋆
  mvhd (108 B) ⋆
  meta (177 B) ⋆
  trak (16049 B) ⋆
    tkhd (92 B) ⋆
    edts (36 B)
      elst (28 B)
    mdia (15913 B) ⋆
      mdhd (32 B) ⋆
      hdlr (44 B) ⋆
      minf (15829 B) ⋆
        vmhd (20 B) ⋆
        dinf (36 B) ⋆
          dref (28 B) ⋆
        stbl (15765 B) ⋆
          stsd (165 B) ⋆
          stts (6144 B) ⋆
          ctts (5944 B)
          stss (124 B) ⋆
          stsz (3088 B) ⋆
          stsc (76 B) ⋆
          co64 (216 B)
  trak (5451 B) ⋆
    tkhd (92 B) ⋆
    mdia (5351 B) ⋆
      mdhd (32 B) ⋆
      hdlr (44 B) ⋆
      minf (5267 B) ⋆
        smhd (16 B) ⋆
        dinf (36 B) ⋆
          dref (28 B) ⋆
        stbl (5207 B) ⋆
          stsd (91 B) ⋆
          stts (24 B) ⋆
          stsz (4808 B) ⋆
          stsc (52 B) ⋆
          co64 (224 B)
free (383387 B)
mdat (33584287 B) ⋆
```

C.5 Common intersection

The following (alphabetized) tree structure is shared across all MP4 files and, therefore, must be a superset of the minimum information required for playability.

```
ftyp
mdat
moov
  meta
  mvhd
  trak
    mdia
      hdlr
      mdhd
      minf
        dinf
          dref
        smhd
        stbl
          stsc
```

```
            stsd
            stss
            stsz
            stts
         vmhd
   tkhd
```