



Javier Domínguez Gómez

jdg@member.fsf.org

Fingerprint: 94AD 19F4 9005 EEB2 3384 C20F 5BDC C668 D664 8E2B

v0.03 - Noviembre 2021

<https://github.com/JavDomGom/videostego>

Contents

1	Introducción	2
2	Análisis de archivos MP4 por bloques	3
2.1	ftyp	4
2.2	mdat (info)	5
2.3	mdat (data)	5
2.4	moov	6
2.4.1	mvhd	6
2.4.2	drm	8
2.4.3	trak	9
3	Ocultación de información	10
3.1	Espacio de trabajo	11
3.2	Algoritmo de escritura	12
3.2.1	Cálculo de la longitud de M en bits	13
3.2.2	Definición y escritura de la cabecera	14
3.2.3	Conversión del mensaje a binario	15
3.3	Algoritmo de lectura	16
3.4	Capacidades	19
3.5	Debilidades	19

1 Introducción

MPEG-4 Part 14 o MP4 es un formato multimedia digital basado en contenedores que comúnmente se usa para almacenar vídeo y audio, pero también se puede usar para almacenar otros datos como subtítulos e imágenes fijas. MP4 permite la transmisión a través de Internet. La única extensión de nombre de archivo oficial para los archivos MPEG-4 Parte 14 es .mp4, pero muchas tienen otras extensiones, más comúnmente .m4a y .m4p. La especificación del formato de archivo MPEG-4 se basó en la especificación del formato QuickTime. Los archivos MP4 se pueden utilizar para almacenar o transmitir contenido audiovisual completo en un solo archivo. Debido a que se puede dividir fácilmente en partes más pequeñas, se usa ampliamente en servicios de transmisión de vídeo basados en tecnologías de tasa de bits adaptables como Smooth Streaming, MPEG-DASH y HLS.

Los archivos MP4 constan de varios bloques de datos consecutivos también llamados cajas o contenedores. Cada contenedor tiene un encabezado de 8 bytes: tamaño de fragmento de 4 bytes (big-endian, byte alto primero) y tipo de fragmento de 4 bytes, una de las siguientes firmas predefinidas:

- | | | | | |
|--------|--------|--------|--------|--------|
| • chap | • imap | • meta | • pnot | • ssrc |
| • clip | • jP2 | • mfra | • sapt | • sync |
| • crgn | • kmat | • moof | • skip | • tmed |
| • ctab | • load | • moov | • stts | • udta |
| • free | • matt | • pdin | • stsc | • uuid |
| • ftyp | • mdat | • PICT | • stsz | • wide |

Estos a su vez suelen tener otros contenedores de segundo nivel, también con su encabezado correspondiente de 8 bytes. El contenedor `moov` es probablemente el más importante de todos los descritos anteriormente. Contiene pistas de vídeo y audio, información del encabezado de la película como escala de tiempo, duración, características de visualización de la película. Es bastante habitual encontrar el contenedor `moov` al final del archivo `mp4`, después de todos los demás contenedores para garantizar que todos los metadatos se lean antes de `moov` cuando el archivo se reproduce desde el servidor.

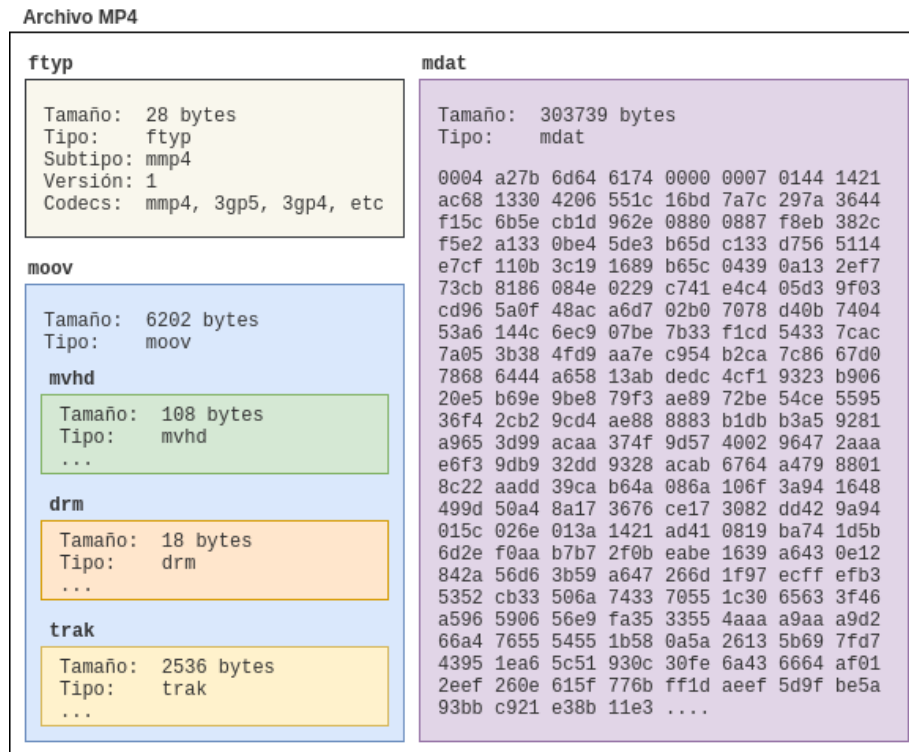


Figure 1: Ejemplo de arquitectura básica en un archivo MP4.

El bloque o contenedor **moov** no almacena información sobre audio/video en bruto. En su lugar, almacena todas las referencias a ubicaciones en el contenedor **mdat**, que es el contenedor donde realmente se almacena toda la información en bruto. El contenedor **mdat** tiene toda su información dividida en varios segmentos. Así pues, las referencias que almacena el contenedor **moov** apuntan a cada uno de esos segmentos para decodificarlos correctamente mediante un software de reproducción adecuado. Este software necesitará leer adicionalmente los contenedores **stts** (*sample-to-time table*) y **stsc** (*sample-to-chunk table*) para obtener la información de sincronización correcta.

2 Análisis de archivos MP4 por bloques

Para ilustrar este documento con ejemplos se ha utilizado el archivo **sample_1.mp4** que se puede encontrar en el repositorio de GitHub del proyecto.¹

En términos generales, los archivos MP4 se dividen en tres secciones principales: **ftyp** (tipo de contenedor), **mdat** (media data) y **moov** (metadatos),

¹https://github.com/JavDomGom/videostego/blob/main/mp4/sample_1.mp4

donde los metadatos contienen información general, información sobre la duración de cada frame de vídeo/audio, los offsets de cada frame de audio/vídeo, etc.

2.1 ftyp

En el primer bloque se encuentra la firma o *signature* del archivo. Nos servirá para identificar el tipo de archivo con el que estamos trabajando y otros datos adicionales como los formatos que es capaz de reconocer o con los que es compatible.

ftyp	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00000000	00	00	00	1c	66	74	79	70	6d	6d	70	34	00	00	00	01
0x00000010	6d	6d	70	34	33	67	70	35	33	67	70	34				

Figure 2: Detalle del bloque "ftyp".

- **Tamaño:** Los 4 primeros bytes del bloque indican el tamaño del mismo, en este caso 28 bytes (0x0000001c).
- **Tipo:** Los siguientes 4 bytes indican un código de cuatro caracteres. Se utiliza para identificar el tipo de contenedor principal empelado, la compatibilidad o el uso previsto de un archivo multimedia. En este caso el tipo es **ftyp** (0x66747970). Este tipo "ftyp" solo se aplica a los formatos de archivo contenedor MP4 o QuickTime (.mov) más nuevos.
- **Subtipo:** Los siguientes 4 bytes indican el subtipo o codec principal con otro código de 4 caracteres. En este caso **mmp4** (0x6d6d7034), quiere decir que el codec principal es *MPEG-4/3GPP Mobile Profile (.MP4/.3GP)*.
- **Versión:** Los 4 bytes que vienen a continuación representan el número de la versión del codec principal anterior, en este caso la versión es **1** (0x00000001).
- **Codecs compatibles:** Los últimos bytes hasta llegar al final del bloque se pueden dividir en secciones de 4 bytes. Cada una de estas secciones, representa el codec con el que el archivo es compatible, en este caso **mmp4** (0x6d6d7034), **3gp5** (0x33677035) *3GPP Media (.3GP) Release 5* y **3gp4** (0x33677034) *3GPP Media (.3GP) Release 4*. 3GP es el formato estándar requerido para los archivos multimedia enviados mediante el Servicio de Mensajería Multimedia (MMS) y los Servicios Multimedia Broadcast Multicast (MBMS).

El lector puede encontrar en la red una lista completa de todas las designaciones MP4/QuickTime ftyp conocidas hasta la fecha², así como un extracto de los diferentes formatos de archivo 3GPP detallado³.

2.2 mdat (info)

En este archivo **sample.1.mp4** el segundo bloque es simplemente informativo, nos indicará información acerca del siguiente bloque.

<u>mdat (info)</u>	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0x00000010														00	00	00	08
0x00000020				6d	64	61	74										

Figure 3: Detalle del primer bloque "mdat".

- **Tamaño:** Los 4 primeros bytes de este bloque nos indica el tamaño del mismo, en este caso **8** bytes (0x00000008).
- **Tipo:** Y los siguientes 4 bytes nos indican el tipo de contenedor que encontraremos en el siguiente bloque, que es **mdat** (0x6d646174).

2.3 mdat (data)

En este bloque encontraremos los el contenedor "mdat", que contendrá los todos los frames de vídeo y audio.

<u>mdat (data)</u>	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00000020					00	04	a2	7b	6d	64	61	74	00	00	00	07
0x00000030	01	44	14	21	ac	68	13	30	42	06	55	1c	16	bd	7a	7c
0x00000040	29	7a	36	44	f1	5c	6b	5e	cb	1d	96	2e	08	80	08	87
	...															
0x0004a290	0b	0e	27	20	ff	ff	e7	5e	4b	ff	ff	ff	ff	ff	ff	7f

Figure 4: Detalle del segundo bloque "mdat".

- **Tamaño:** Los 4 primeros bytes de este bloque nos indica el tamaño del mismo, en este caso **303739** bytes (0x0004a27b).

²<http://www.ftyps.com/>

³<http://www.ftyps.com/3gpp.html>

- **Tipo:** Y los siguientes 4 bytes nos indican el tipo de contenedor en el que estamos, **mdat** (0x6d646174).
- **El resto de bytes:** Todos los bytes que vienen a continuación hasta el final del bloque son los media-data.

2.4 moov

El bloque **moov** se compone de varios sub-bloques. En general, **moov** seguirá a **ftyp**. Moov contiene metadatos e información básica relacionada con el audio y el vídeo. A continuación una lista de los principales sub-bloques o contenedores de **moov**:

- **drm**
- **iods**
- **mvhd**
- **trak**
- **udta**

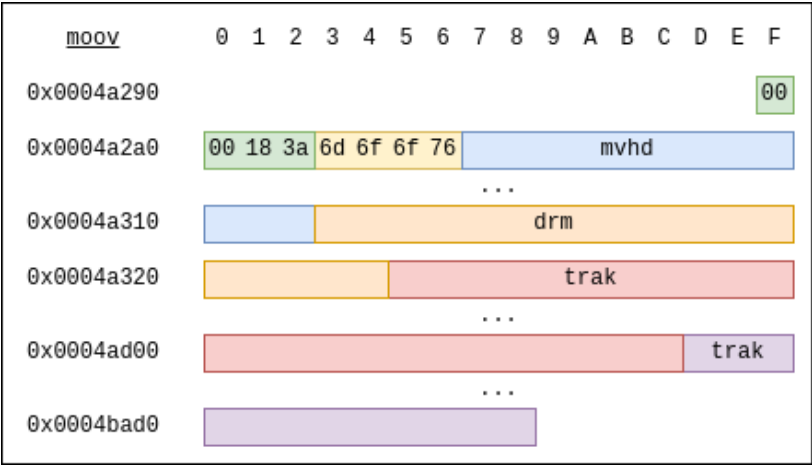


Figure 5: Detalle del bloque "moov".

2.4.1 mvhd

El bloque **mvhd** (Movie Header Box) contendrá los datos de la cabecera o header de la película.

<u>mvhd</u>	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x0004a2a0								00	00	00	6c	6d	76	68	64	00
0x0004a2b0	00	00	00	c1	02	17	10	c1	02	17	63	00	00	02	58	00
0x0004a2c0	00	4a	b3	00	01	00	00	01	00	00	00	00	00	00	00	00
0x0004a2d0	00	00	00	00	01	00	00	01	00	00	00	00	00	00	00	00
0x0004a2e0	00	00	00	00	00	01	00	00	00	00	00	00	00	00	00	00
0x0004a2f0	00	00	00	00	40	00	00	00	00	00	00	00	00	00	00	00
0x0004a300	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x0004a310	00	00	03													

Figure 6: Detalle del bloque "mvhd".

- **box size:** 4 bytes. Indica el tamaño en bytes del bloque, en este caso **108** bytes (0x0000006c).
- **box type:** 4 bytes, es el tipo de bloque, en este caso **mvhd** (0x6d766864).
- **version:** 1 byte. Versión del bloque, puede ser 0 o 1, generalmente **0** (0x00).
- **flags:** 3 bytes, flags, en este ejemplo no hay (0x000000).
- **creation time:** 4 bytes. Hora de creación relativa a la hora UTC 1904-01-01 00:00:00 en segundos. En este ejemplo es **3238139664** (0xc1021710) que equivale a la fecha y hora 2006-08-11 11:14:24.
- **modification time:** 4 bytes. Hora de la última modificación (0xc1021763) que equivale a la fecha y hora 2006-08-11 11:15:47.
- **time scale:** 4 bytes. La escala de tiempo. En este ejemplo tiene un valor de **600** (0x00000258).
- **duration:** 4 bytes. La duración y el valor del campo time scale se pueden usar para calcular la duración real de la pista en segundos dividiendo el valor de duration entre el valor de time-scale. En este ejemplo duration tiene un valor de **19123** (0x00004ab3), y time duration tiene un valor de 600 que ya habíamos calculado antes, por lo tanto la pista tendrá una duración total de $19123/600 = 31.87$ segundos.
- **rate:** 4 bytes. La velocidad de reproducción recomendada, los 16 bits superiores y los 16 bits inferiores son la parte entera y la parte decimal

del punto decimal respectivamente, es decir, formato [16.16], el valor de **1.0** (0x00010000) significa reproducción normal hacia adelante.

- **volume:** 2 bytes. Similar al campo rate, el formato es [8.8], **1.0** (0x0100) significa volumen máximo.
- **reserved:** 10 bytes, simplemente una cantidad de bytes reservados para uso del propietario del formato (QuickTime - Apple), en este caso no se le da uso (0x00000000000000000000).
- **matrix:** 36 bytes. Matriz de transformación de vídeo. Esta matriz define cómo mapear puntos de un espacio de coordenadas a otro. Modificando el contenido de una matriz de transformación, se pueden realizar varias operaciones de visualización de gráficos como traducción, rotación y el escalado. La matriz utilizada para lograr las transformaciones bidimensionales se describen matemáticamente mediante una matriz de 3 por 3.⁴
- **preview time:** 4 bytes. El valor de tiempo de la película en el que comienza la vista previa, en este caso no se le da uso (0x00000000).
- **preview duration:** 4 bytes. La duración de la vista previa de la película en unidades de escala de tiempo de la película.
- **poster time:** 4 bytes. El valor temporal del tiempo del cartel de la película, en este caso no se le da uso (0x00000000).
- **selection time:** 4 bytes. El valor de tiempo para la hora de inicio de la selección actual, en este caso no se le da uso (0x00000000).
- **selection duration:** 4 bytes. La duración de la selección actual en unidades de escala de tiempo de película, en este caso no se le da uso (0x00000000).
- **current time:** 4 bytes. El valor de tiempo para la posición de tiempo actual dentro de la película, en este caso no se le da uso (0x00000000).
- **next track id:** 4 bytes. Indica un valor que se utilizará para el número de identificación de la siguiente pista agregada a la película. En este caso el valor es **3** (0x00000003). 0 no es un valor de ID de pista válido.

2.4.2 drm

Debido a que a los proveedores de contenido y a los servicios de transmisión les gusta ganar dinero, el cifrado se usa generalmente para controlar quién puede reproducir un archivo de vídeo y cómo. Esto también se denomina "Gestión de Derechos Digitales" o "Digital Rights Management" (DRM).

⁴https://developer.apple.com/library/archive/documentation/QuickTime/QTFF/QTFFChap4/qtff4.html#/apple_ref/doc/uid/TP40000939-CH206-18737

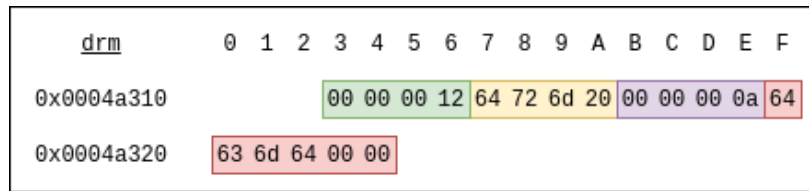


Figure 7: Detalle del bloque "drm".

- **box size:** 4 bytes. Indica el tamaño en bytes de este bloque, en este caso **18** bytes (0x00000012).
- **box type:** 4 bytes, es el tipo de bloque, en este caso "**drm**" (0x64726d20), nótese el espacio del final.
- **next box size:** 4 bytes. Indica el tamaño en bytes del bloque contenido dentro de este bloque drm, en este caso **10** bytes (0x0000000a).
- **file type:** 10 bytes. Indica el tipo de archivo, en este caso **dcmd** (Digital Cinema Master Digital) (0x64636d64), es el master digital DSM convertido ya a los estándares DCI, pero sin comprimir.

El método de cifrado más común y estandarizado para contenido encapsulado MP4 se llama Cifrado común. Cuando se utiliza Common Encryption, el cifrado solo se aplica a la carga útil mdat. Los metadatos permanecen sin cifrar.

También se agrega información a los metadatos para indicar que el contenido está encriptado y cómo. Esta información también incluye los ID de las claves necesarias para descifrar el contenido.

La forma de utilizar los ID de clave para obtener las claves reales depende de la solución DRM específica que se utilice, pero este es el único aspecto específico del proveedor. Qué algoritmos de cifrado usar y a qué aplicarlos está estandarizado.

2.4.3 trak

El bloque moov puede contener varios bloques de tipo **trak**. Cada uno define una pista individual de una película, y esta puede tener una o más. Cada pista es independiente de las otras pistas de la película y lleva su propia información temporal y espacial.

A continuación los propósitos que tienen los bloques **trak**:

- Para contener referencias y descripciones de media data.
- Para contener pistas modificadoras (interpolaciones, etc).

- Para contener información sobre la paquetización para protocolos de *streaming* (*hint tracks*). Estos *hint tracks* pueden contener referencias a *samples* de media data o copias de estos *samples*.⁵

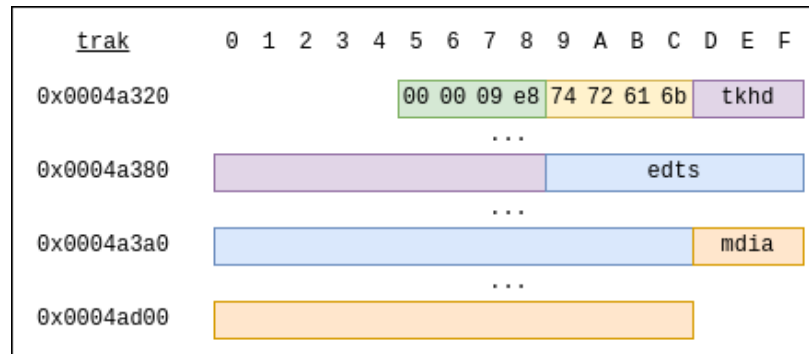
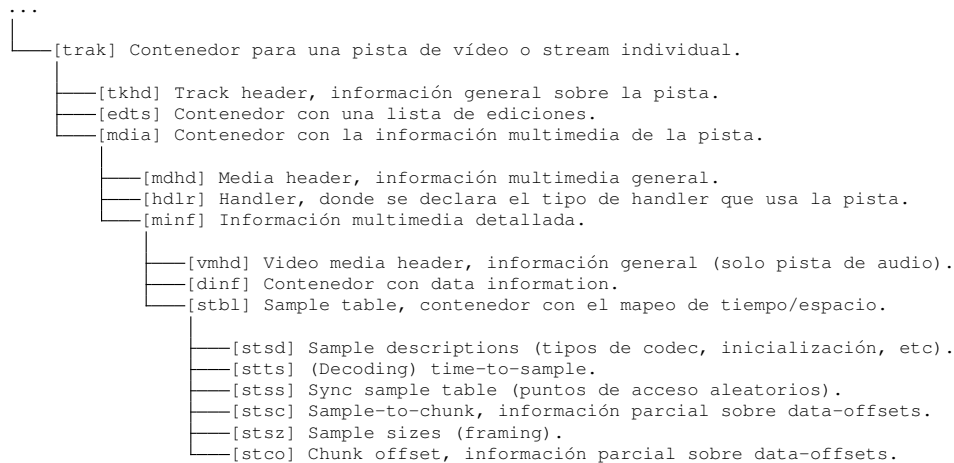


Figure 8: Detalle del bloque "trak".

Cada bloque trak contiene a su vez muchos otros bloques como tkhd, edts, mdia, etc. Y estos a su vez contienen más bloques como mdhd, hdlr, minf, etc. Véase un ejemplo en el siguiente esquema jerárquico:



3 Ocultación de información

Ahora que ya conocemos la arquitectura o la estructura de datos de la que se compone un archivo MP4 podremos realizar algunas modificaciones en algunos

⁵https://developer.apple.com/library/archive/documentation/QuickTime/QTFF/QTFFChap3/qtff3.html#/apple_ref/doc/uid/TP40000939-CH205-69871

bytes, en bloques y zonas muy concretas, no podremos modificar cualquier byte. En primer lugar hay que plantear un mensaje. Este mensaje M a ocultar puede ser una cadena de texto plano como "Hello world!" o una cadena de texto cifrado como "ce415a50bb09e42e7420d73196e47324". En todos los casos el mensaje se tratará como texto plano, es decir, una cadena de caracteres ASCII. Si el mensaje se hubiese cifrado previamente con algún algoritmo criptográfico o mediante alguna función *hash*, el receptor leerá el mensaje en su forma original, tal como se escribió en el fichero, no se descifrá.



Figure 9: Proceso de envío y recepción del mensaje "M".

3.1 Espacio de trabajo

Como se ha visto en puntos anteriores, en los que se ha realizado un análisis de cada una de las partes o contenedores que componen un archivo MP4, cada contenedor o bloque tiene un propósito específico. Esto quiere decir que no podremos escoger cualquier bloque de datos y hacer modificaciones en él. Si queremos realizar alguna modificación en el archivo un buen espacio de trabajo es el bloque "mdat" de datos, pues aquí residen los frames de las pistas de vídeo y datos relativos a las pistas de audio. Suele ser un bloque bastante amplio, con gran cantidad de bytes, lo que nos proporcionará cierta flexibilidad.

No se han de modificar ni los primeros ni los últimos bytes de este bloque, ya que dependiendo del tipo y subtipo de archivo MP4 estas zonas del bloque pueden contener algunos metadatos que es mejor no modificar, si no se podrían levantar sospechas e incluso corromper el bloque y como consecuencia el archivo, lo que podría provocar que no se reproduzca correctamente en algunos dispositivos y programas. Así pues el algoritmo de escritura dividirá el tamaño del bloque en 3 partes iguales y se establecerá el tercio central (2/3) como el espacio de trabajo, donde se realizarán las modificaciones oportunas para ocultar nuestro mensaje.

Para ilustrar este documento con ejemplos se ha utilizado un archivo MP4 llamado **sample_1.mp4** y que se puede encontrar en el repositorio de GitHub del proyecto.⁶ En este archivo, el bloque de datos "mdat" comienza en el offset "0x00000024". Si lo dividimos en tres partes iguales, el segundo tercio comienza en el offset 0x00018ba2, y se extiende hasta el offset "0x0003171f", lo que quiere decir que en este archivo MP4 tendremos un espacio de trabajo de 101245 bytes.

⁶<https://github.com/JavDomGom/videostego/blob/main/mp4/sample.1.mp4>

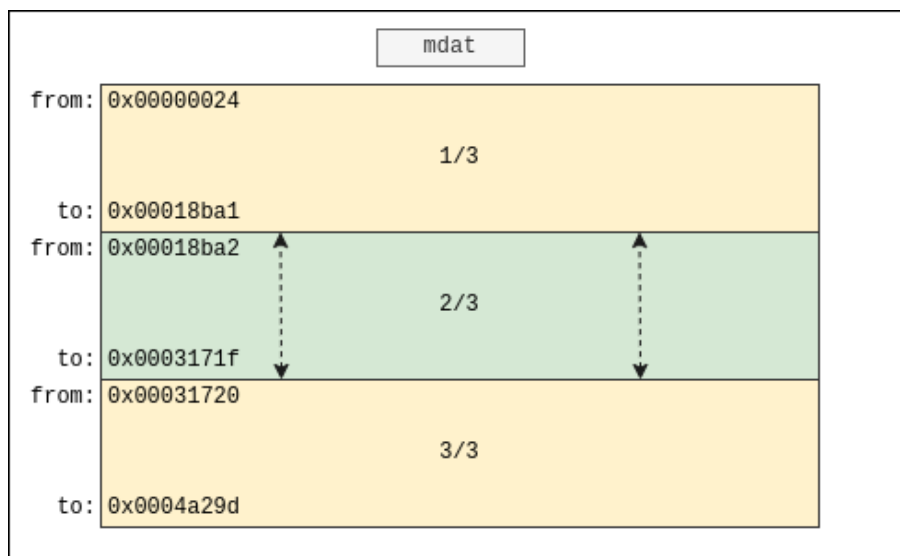


Figure 10: Zona central del bloque "mdat".

Según el archivo MP4 que utilicemos dispondremos de más o menos cantidad de bytes para nuestro espacio de trabajo. Así pues, cuanto más tamaño (duración de la película) tenga el archivo MP4 con el que se trabaje más grande será dicho espacio.

3.2 Algoritmo de escritura

Ahora que ya sabemos dónde se encuentra nuestro espacio de trabajo y qué capacidades podemos continuar con la escritura del mensaje.

Cada caracter ASCII del mensaje hará uso de 8 bytes del espacio de trabajo en el bloque de datos "mdat". Se ha escogido esta estrategia porque si se utilizase un byte para cada caracter del mensaje, este se podría ver fácilmente en una inspección del archivo MP4 si este se abre con algunos editores de texto. Por otro lado, sobrescribir un byte como 0xb7 por un byte como 0xf2 levantaría más sospechas que si se aumenta o disminuye únicamente un bit, el bit menos significativo (LSB)⁷, es decir, sobrescribir un byte como 0xb7 por un byte como 0xb8, o 0xff por 0xfe. Al fin y al cabo, un caracter ASCII tiene una representación binaria de 8 bits.

⁷https://en.wikipedia.org/wiki/Bit_numbering#Least_significant_bit

<u>Char</u>		<u>Hex</u>		<u>Bin</u>
<i>a</i>	\Rightarrow	0x61	\Rightarrow	01100001
<i>b</i>	\Rightarrow	0x62	\Rightarrow	01100010
<i>c</i>	\Rightarrow	0x63	\Rightarrow	01100011
...	\Rightarrow	...	\Rightarrow	...

La longitud del mensaje de entrada M en base 2 o binario es $|M_2|$. Cada bit (b_i) del caracter se utilizará para buscar paridad con cada byte (B_i) del bloque "mdat" de modo que 0 se utilizará marcar aquellos bytes que han de ser pares, y 1 para los impares, y si no coinciden en paridad o imparidad se modificará el byte (B_i) del bloque sumando o restando una unidad para hacer que coincidan. El algoritmo empleado para modificar los bytes que lo requieran se puede representar mediante la siguiente función recursiva definida por intervalos:

$$B_i + \sum_{i=0}^{|M_2|} \left\{ \begin{array}{ll} 0 & \text{if } ([B_i \wedge b_i] \bmod 2 = 0) \vee ([B_i \wedge b_i] \bmod 2 \neq 0) \\ \begin{cases} 1 & \text{if } B_i < 0\text{xff} \\ -1 & \text{otherwise} \end{cases} & \text{otherwise} \end{array} \right.$$

Es decir, si b_i y B_i ambos son pares, el byte B_i se queda como está, no se modifica, en caso contrario el byte B_i se incrementará una unidad siempre que sea un byte menor que 0xff, si no se le restará una unidad.

En los siguientes puntos se explica en detalle el funcionamiento del algoritmo con bytes reales, y los pasos a realizar en cada fase de escritura del mensaje M .

3.2.1 Cálculo de la longitud de M en bits

Para calcular la longitud de bits que tiene el mensaje de entrada M deberemos hacer una conversión a base 2 o binario. Lo más sencillo es multiplicar el número total de caracteres ASCII del mensaje por 8, ya que cada carácter ASCII equivale a 8 bits. Así pues la longitud de M en base 2 se definirá como $|M_2|$

<u>M</u>		<u>$Bits \times Chars$</u>		<u>Longitud</u>
Hello world!	=	8×12	\Rightarrow	$ M_2 = 96$
Quedamos a las 15:33 donde siempre	=	8×34	\Rightarrow	$ M_2 = 272$
No le busques 3 pies al gato	=	8×28	\Rightarrow	$ M_2 = 224$
ce415a50bb09e42e7420d73196e47324	=	8×32	\Rightarrow	$ M_2 = 256$

Para este ejemplo usaremos como valor de M el mensaje "Hello world!", que hará un uso de 96 bytes. La longitud del mensaje de entrada $|M|$ es un dato que se ha de reservar, ya que se utilizará para algunos cálculos más adelante.

3.2.2 Definición y escritura de la cabecera

La cabecera ocupará $2 + 32$ bytes, en total 34 bytes. Los dos primeros bytes se utilizarán para establecer el tamaño total de bytes que vamos a usar para ocultar nuestro mensaje M más los 34 bytes de la cabecera, y se escribirá en base 16 o hexadecimal. Por lo tanto estos dos bytes tendrán un valor mínimo de $0x0022$ (34 bytes de la cabecera) y un valor máximo de $0xffff$ (65535 bytes).

En primer lugar iremos al offset $0x00018ba2$, que es donde anteriormente habíamos calculado que comienza el segundo tercio del bloque "mdat". Ahí encontraremos como dos primeros bytes $0x40$ y $0xc4$. En el punto anterior calculamos para ocultar el mensaje de entrada M vamos a necesitar 96 bytes. A esta cifra de 96 le tenemos que sumar los 34 que ocupará siempre la cabecera, por lo tanto $96 + 34 = 130$, que en hexadecimal es $0x82$. Así pues sobrescribiremos los dos bytes $0x40c4$ por $0x0082$.

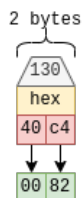


Figure 11: Escritura del tamaño total que se usará.

Los otros 32 bytes de la cabecera están reservados para escribir una etiqueta de 4 caracteres que servirá para identificar si el archivo MP4 ya ha sido modificado por este software. Esos cuatro caracteres son "vstg", que es un acrónimo de "videostego". Debemos convertir cada uno de estos caracteres a su valor en binario.

<u>Char</u>		<u>Hex</u>		<u>Bin</u>
<i>v</i>	\Rightarrow	$0x76$	\Rightarrow	01110110
<i>s</i>	\Rightarrow	$0x73$	\Rightarrow	01110011
<i>t</i>	\Rightarrow	$0x74$	\Rightarrow	01110100
<i>g</i>	\Rightarrow	$0x67$	\Rightarrow	01100111

Ahora que ya conocemos el valor binario que tiene cada caracter, emplearemos el algoritmo de escritura y utilizaremos cada bit de cada caracter para buscar paridad con cada byte de los actuales 32 bytes que estamos leyendo. Si el byte que se encuentra en el cursor de lectura es par, por ejemplo $0xca$, y nuestro bit es 0, el byte se deja como está, no se modificará. En caso contrario se le sumará una unidad, siempre que sea un byte inferior a $0xff$, si no se le restará una unidad. Aplica el mismo principio para los impares, es decir, si el

byte que se encuentra en el cursor de lectura es impar, por ejemplo 0xfb, y nuestro bit es 1, el byte se deja como está, y en caso contrario se modificará de la misma manera que los pares, sumando o restando una unidad para conseguir que estén alineados en una comparación de paridad.

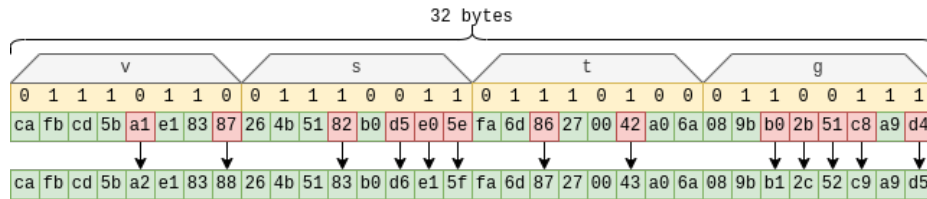


Figure 12: Escritura de la etiqueta "vstg" en la cabecera.

3.2.3 Conversión del mensaje a binario

Anteriormente decidimos que nuestro mensaje será "Hello world!". Lo primero que debemos hacer es convertir cada uno de los caracteres a binario.

<u>Char</u>	<u>Hex</u>	<u>Bin</u>
<i>H</i>	⇒ 0x48	⇒ 01001000
<i>e</i>	⇒ 0x65	⇒ 01100101
<i>l</i>	⇒ 0x6c	⇒ 01101100
<i>l</i>	⇒ 0x6c	⇒ 01101100
<i>o</i>	⇒ 0x6f	⇒ 01101111
	⇒ 0x20	⇒ 00100000
<i>w</i>	⇒ 0x77	⇒ 01110111
<i>o</i>	⇒ 0x6f	⇒ 01101111
<i>r</i>	⇒ 0x72	⇒ 01110010
<i>l</i>	⇒ 0x6c	⇒ 01101100
<i>d</i>	⇒ 0x64	⇒ 01100100
!	⇒ 0x21	⇒ 00100001

El proceso que viene ahora es exactamente igual al que hemos empleado en la cabecera para escribir la etiqueta "vstg", es decir, usar cada bit del mensaje para buscar paridad con cada byte que vaya leyendo el puntero de lectura, solo que ahora en vez de leer, comparar y sobrescribir sobre 32 bytes lo haremos sobre los siguientes 96 bytes, un dato que calculamos y reservamos anteriormente.

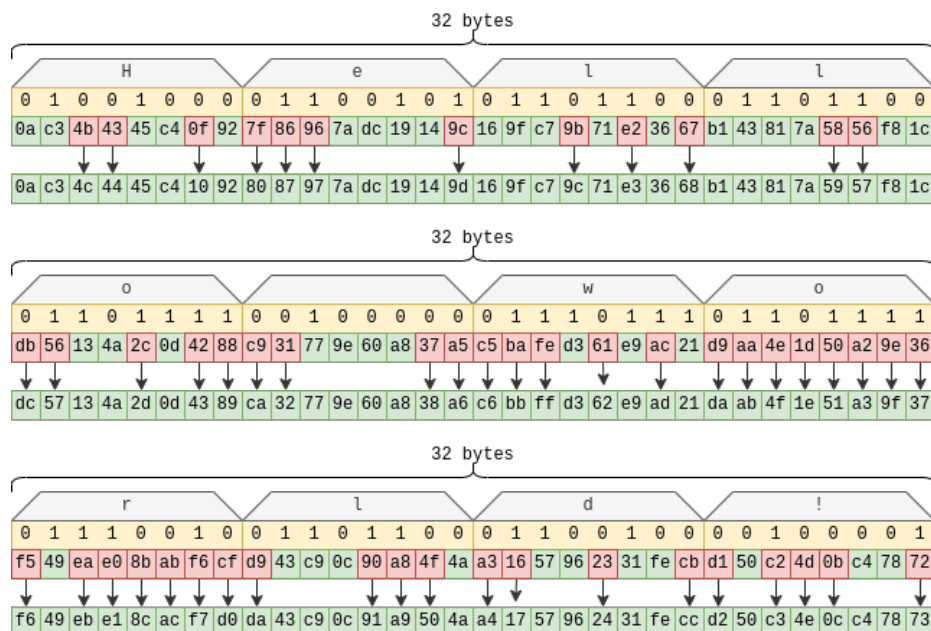


Figure 13: Escritura del mensaje sobre el archivo MP4 original.

3.3 Algoritmo de lectura

Dado un archivo de entrada, se deberán leer todas las cabeceras de cada bloque que lo compone. El objetivo es encontrar un bloque cuyo tipo sea "mdat" y que su tamaño sea superior a 8 bytes, pues existe la posibilidad de encontrar un bloque de tipo "mdat" que solo contenga información sobre el nombre del siguiente bloque, que podrá ser también de tipo "mdat", y este ocupará 8 bytes. Por ejemplo en el offset 0x0000001c del archivo **sample_1.mp4** utilizado de ejemplo.

- offset -	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0x00000000	0000	001c	6674	7970	6d6d	7034	0000	0001	...	ftyp	mp4	...					
0x00000010	6d6d	7034	3367	7035	3367	7034	0000	0008	mmp43gp53gp4	...							
0x00000020	6d64	6174	0004	a27b	6d64	6174	0000	0007	mdat	...	{	mdat	...				
0x00000030	0144	1421	ac68	1330	4206	551c	16bd	7a7c	.	D	.	!	h	.	0B	U	...

Figure 14: Bloque "mdat" solo con el nombre del siguiente bloque.

El bloque "mdat" con un tamaño superior a 8 bytes comienza en este caso en el offset 0x00000024. A partir de aquí, los primeros 4 bytes indican el tamaño total del bloque, que en este caso es de 303739 bytes (0x0004a27b).

<code>mdat (data)</code>	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
<code>0x00000020</code>					00	04	a2	7b	6d	64	61	74	00	00	00	07
<code>0x00000030</code>	01	44	14	21	ac	68	13	30	42	06	55	1c	16	bd	7a	7c
<code>0x00000040</code>	29	7a	36	44	f1	5c	6b	5e	cb	1d	96	2e	08	80	08	87
	...															
<code>0x0004a290</code>	0b	0e	27	20	ff	ff	e7	5e	4b	ff	ff	ff	ff	ff	ff	7f

Figure 15: Detalle del bloque "mdat" con los datos a leer.

El siguiente paso es dividir el tamaño total del bloque entre 3, pues si se ha escrito algún mensaje oculto en el archivo se habrá hecho a partir del segundo tercio del bloque. En este ejemplo el segundo tercio del bloque "mdat" comienza en el offset `0x00018ba2`, y termina en el offset `0x0003171f`, tal y como se muestra en la siguiente imagen.

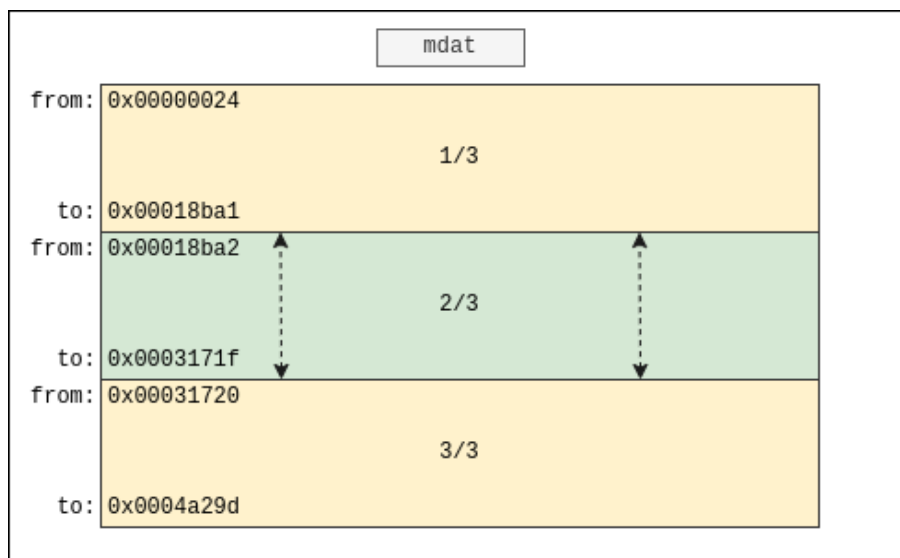


Figure 16: Zona central del bloque "mdat".

A partir de aquí, los dos primeros bytes nos indicarán el tamaño total de bytes que tenemos que leer y procesar para descifrar el mensaje oculto, si lo hubiera. En este caso los dos primeros bytes son `0x0082`, lo que quiere decir que vamos a tener que leer y procesar un total de 130 bytes, incluidos los dos bytes presentes.



Figure 17: Longitud de bytes a leer.

$$size = (B_0 + B_1)_{10}$$

A continuación hay que leer los siguientes 32 bytes y comprobar por cada uno de ellos su paridad. En caso de que el byte sea par, por ejemplo 0xca, se obtendrá un 0, y si el byte es impar, por ejemplo 0xfb, se obtendrá un 1, formando así una cadena binaria $|type|_2^{32}$ que se podrá leer a posteriori como una cadena de 32 bits (4 bytes).

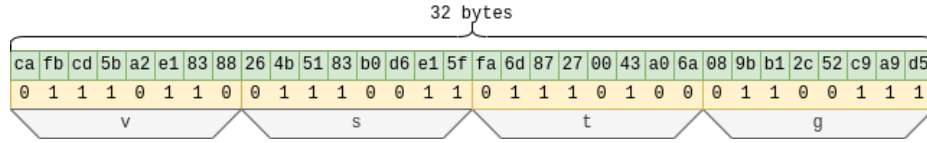


Figure 18: OP_CODE o type "vstg".

$$type = b_i \sum_{i=2}^{34} \begin{cases} 0 & \text{if } B_i \bmod 2 = 0 \\ 1 & \text{otherwise} \end{cases}$$

Si los 32 bits (4 bytes) resultantes representaran los 4 caracteres ASCII `vstg` se consideraría que el archivo contiene un mensaje oculto en los siguientes bytes hasta completar los 130 bytes que leímos en la cabecera. En caso contrario, no se continúa y se descarta la posibilidad de que en el archivo exista algún mensaje oculto escrito con este software. Hay que tener en cuenta que ya hemos leído 2+32 bytes, 2 del tamaño y 32 de este flag u OP_CODE `vstg`, esto quiere decir que aún quedan por leer 96 bytes.

Los siguientes 96 bytes se leerán empleando el mismo algoritmo de lectura que acabamos de usar en la lectura del flag `vstg`.

$$\text{if } (type)_{ASCII} = \text{"vstg"} ; M = b_i \sum_{i=34}^{size} \begin{cases} 0 & \text{if } B_i \bmod 2 = 0 \\ 1 & \text{otherwise} \end{cases}$$

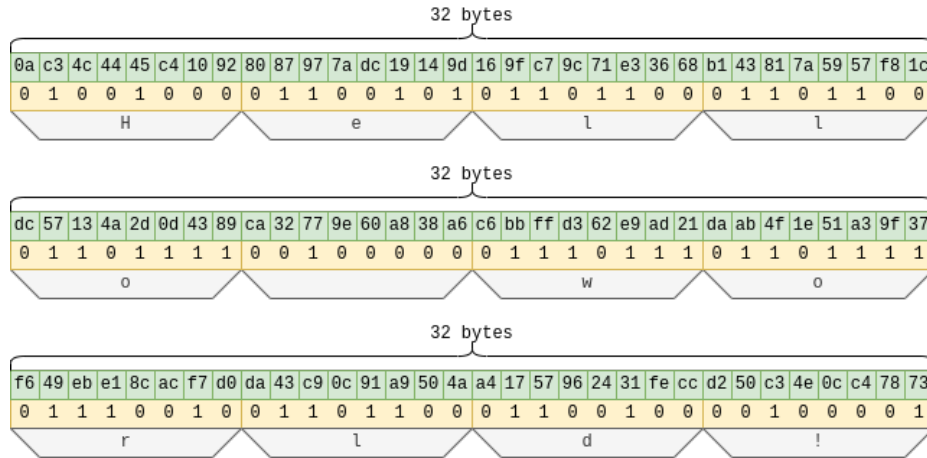


Figure 19: Longitud de bytes a leer.

En este ejemplo la cadena de bits 01001000 01100101 01101100 01101100 01101111 00100000 01110111 01101111 01110010 01101100 01100100 00100001 equivale al mensaje "Hello world!" en texto plano.

3.4 Capacidades

El mensaje M que se quiera ocultar tendrá una longitud variable, pero en ningún caso puede superar la longitud de 65535 ($0xffff$) menos 2+32 bytes de la cabecera, es decir, podrá tener una longitud máxima de 65501 bytes. Si cada caracter del mensaje va a ocupar 8 bytes, la longitud máxima del mensaje será de 8187 caracteres.

Si en futuras actualizaciones de Videostego se aumentara la cantidad de bytes reservados al tamaño de lectura, por ejemplo de los 2 bytes actuales a 3 bytes, el tamaño máximo sería de 16777215 ($0xffffffff$) menos 3+32 bytes de la cabecera, en total 16777180 bytes, que serían 2097147 caracteres, pero por encima de todo hay que tener en cuenta que el tamaño máximo de bytes a escribir o leer no debe superar nunca un tercio de la cantidad total de bytes del contenedor "mdat", y en el caso del archivo usado de ejemplo es una capacidad total de 101245 bytes, equivalente a 12655 caracteres.

3.5 Debilidades

- Si se evalúa la paridad de todos los bytes del bloque "mdat" se extrae una cadena binaria de considerable longitud. Esta se podría convertir a caracteres ASCII, y entre una cadena de texto sin sentido podría encontrarse alguna subcadena que sí lo tenga, siempre y cuando el mensaje M que se escribiera fuera una cadena de texto plano.

- La lectura de posibles mensajes ocultos previamente escritos con videostego no funciona si los archivos MP4 son procesados a través de plataformas como Twitter, LinkedIn o Youtube. Como se puede ver en las siguientes imágenes, estas plataformas modifican la información de los frames añadiendo gran cantidad de bytes, los cambian por completo y además eliminan y añaden metadatos en varios contenedores.

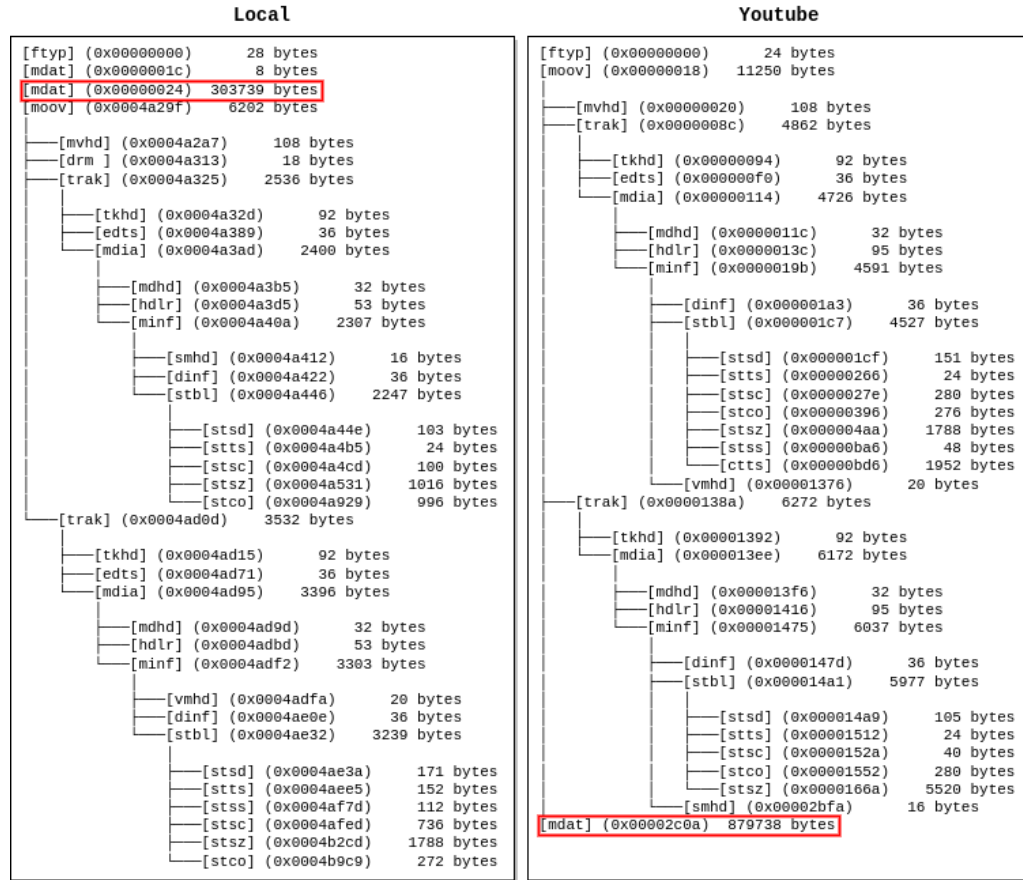


Figure 20: Diferencia tras subirlo a Youtube.

- offset -	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0x00000140	6864	6c72	0000	0000	0000	0000	0000	7669	6465								hdlr.....vide
0x00000150	0000	0000	0000	0000	0000	0000	0000	4953	4f20							ISO
0x00000160	4d65	6469	6120	6669	6c65	2070	726f	6475									Media file produ
0x00000170	6365	6420	6279	2047	6f6f	676c	6520	496e									ced by Google In
0x00000180	632e	2043	7265	6174	6564	206f	6e3a	2031									c. Created on: 1
0x00000190	312f	3238	2f32	3032	312e	0000	0011	ef6d									1/28/2021.....m
0x000001a0	696e	6600	0000	2464	696e	6600	0000	1c64									inf...\$dinf....d

Figure 21: Metadatos en el bloque "hdlr" tras subirlo a Youtube.