南開大學
Nankai University

南　　开　　大　　学

网 络 空 间 安 全 学 院

信息隐藏实验报告

实验 2：MP4 媒体文件分析

艾明旭 戴伊涵 刘璞睿 温博淳

年级：2021 级

指导教师：李朝晖

2024 年 4 月 26 日

目录

一、 实验要求

1. 任选一种媒体文件，进行格式剖析（建议用 UltraEdit）；
2. 针对该类型的文件，讨论可能的隐藏位置和隐藏方法；
3. 实现秘密信息的隐藏和提取。

二、 mp4 格式剖析

(一) MP4 文件格式分析

MP4，全称为 MPEG-4 Part 14，是一种数字多媒体容器格式，广泛用于存储视频和音频内容，也可以包含其他数据，如字幕和图片。MP4 格式基于 ISO 基础媒体文件格式（ISO/IEC 14496-12），这一标准自身源于 Apple 的 QuickTime 文件格式。它是由 Moving Picture Experts Group（MPEG）开发的，作为 MPEG-4 国际标准的一部分被发布。

1. MP4 文件的主要特点

1. 通用性和灵活性：MP4 可以存储视频、音频、文字和图片等多种类型的媒体内容，支持互联网、广播和存储媒体中的多种应用。
2. 高度压缩：MP4 使用高效的编码算法，如 H.264 或 H.265，为视频提供高质量的压缩。这使得 MP4 文件在保持较高画质的同时，文件大小相对较小。
3. 广泛兼容性：MP4 是一个非常流行的格式，被大多数软件和硬件播放器所支持。

2. MP4 文件结构概述

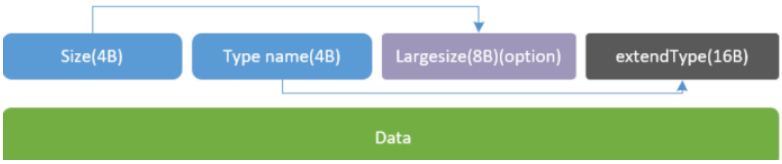
一个 mp4 文件通常由音频和视频两部分组成（当然有些还包含字幕和一些自定义的信息），音频或视频数据：一段在时间上相关联的 sample 序列（sample：对于视频而言是一帧压缩后的图像数据（如 264/h265 数据包），对于音频而言是：一小段语音信号采样编码后的数据（aac 编码数据包）），我们将这种 sample 序列定义为一个 track，于是就有了 video track 和 audio track 的说法。

一段数据协议通常由两部分组成：HEADER + DATA，其中 Header 一般具有一种固定的格式，它的作用是描述其后的 Data 部分。例如我们熟知的 TCP/IP 协议，MP4 协议也不外如是。mp4 协议定义了一种称之为 Box 的数据结构，一个 mp4 文件就是一个个 Box 拼接而成，如下图所示。解析 MP4 文件其实就是对 Box 的读取和解析。

(二) Box 的基本结构

1. Basic Box

Box 结构其本质上也是 HEADER+DATA 的模式，如下图所示：其中前 4 个字节用来表示 Box 的大小 (size)（包括了 header 和 data），紧接着 4 个字节用来表示 Box 类型 (type)，由于 size 字段 4 个字节的限制，当其后的 data 内容很大时，可能会超出 size 字段所能表示的范围，



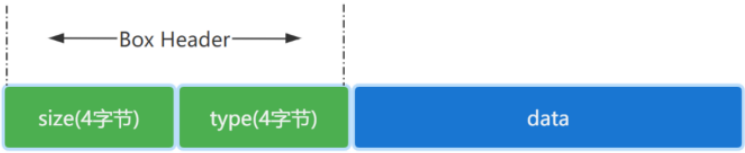
此时 size 字段将会被设置为 1，如果一个 Box 的 size == 1，则会在 type 字段后补充 8 个字节的空间用于表示该 Box 的大小；size 字段还有可能是 0，它表示该 Box 的内容一直到文件结束。对于有些 box，可能还会有一些扩展数据存放入 header 中，一般用 16 个字节来存放，header 数据后跟着的就是 Box 的 data 部分，其大小是 header 中 size 字段或者 largesize 字段所表示的大小减去其 header 自身所占用的大小。

下面是标准协议文档对 Box 的语法定义

Listing 1: Box 类

```
aligned(8) class Box (unsigned int(32) boxtype, optional unsigned int(8)[16]
    extended_type) {
   unsigned int(32) size;
   unsigned int(32) type = boxtype;
   if (size==1) {
      unsigned int(64) largesize;
   } else if (size==0) {
      // box extends to end of file
   }
    if (boxtype== 'uuid' ) {
        unsigned int(8)[16] usertype = extended_type;
    }
}
```

MP4 协议文档中定义了很多种类型的 Box，从上面的语法中可看出，除去一些超大的 Box 和一些用户扩展的 Box 除外，很多 Box 基本结构其实如下图所示。



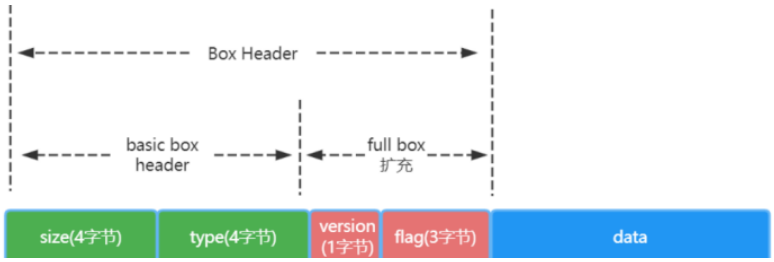
2. Full Box

基于上面的 Box 结构，标准协议在其 Header 基础上扩充了两个字段：1 个字节的 version 字段和 3 个字节的 flags 字段；我们将扩充后的 Box 称之为 Full Box，而将没有扩充这两个字段的 Box 称之为 Basic Box, 下面是标准协议对 Full box 的定义：

Listing 2: FullBox 类

```
aligned(8) class FullBox(unsigned int(32) boxtype, unsigned int(8) v, bit(24)
    f)
   extends Box(boxtype) {
     unsigned int(8) version = v;
     bit(24) flags = f;
```

```
}
```



(三) 部分 MP4 Box 解析

1. File Type Box

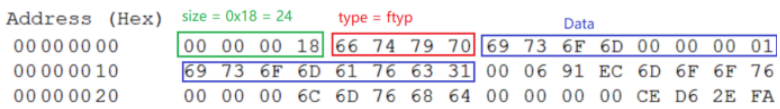
- 类型：ftyp
- 父容器：File（表示 Box 没有嵌套，直接位于文件层，属于最顶层的 box）
- 是否必须有：yes
- 数量：有且仅有一个

虽然 mp4 协议文档中定义了很多种类型的 Box，但对于一个 mp4 文件而言，只需要用到其中的必需的几种 box 即可，其他的 box 是为了满足某些特定需求的场景，用户可以根据需求选择性插入。

Listing 3: File Type Box 类

```
aligned(8) class FileTypeBox extends Box('ftyp')
    unsigned int(32) major_brand;
    unsigned int(32) minor_version;
    unsigned int(32) compatible_brands[];
}
```

1. **major_brand:** 因为兼容性一般可以分为推荐兼容性和默认兼容性。这里 major_brand 相当于是推荐兼容性。一般而言都是使用 isom 即可。如果是需要特定的格式，可以自行定义。
2. **minor_version:** 指最低兼容版本。
3. **compatible_brands:** 和 major_brand 类似，兼容协议，通常是针对 MP4 中包含的额外格式，比如:AVC，AAC 等, 相当于的音视频解码格式，每四个字节为一组。

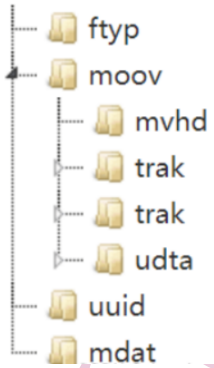


ftyp box 通常位于 mp4 文件的起始位置，上图是某一个 mp4 文件的二进制数据，由上图可知该 box 的大小占用 24 个字节，type 是 ftyp, 根据协议定义，ftyp 是一个 Basic Box，因此除去头部的 8 个字节剩余 16 个字节的内容为其 data 部分

2. Movie Box

- 类型：moov
- 父容器：File
- 是否必须有：yes
- 数量：有且仅有一个

moov box 是一个非常重要的 box：主要用来存放描述 mp4 文件媒体数据的 metadata 信息，包括视频的宽，高，总时长，音频的采样率，通道数，总时长等，以及如何查找真正的媒体数据信息。moov box 是一个容器 box，其内容分布到各个子 box 中。如下图所示：



moov 中的子 box 有很多种，下面只针对必需的 mvhd、trak 这两种 box 做分析。

3. Movie Header Box

- 类型：mvhd
- 父容器：moov
- 是否必须有：yes
- 数量：仅有一个

```
aligned(8) class MovieHeaderBox extends FullBox('mvhd', version, 0) {
    if (version == 1) {
        unsigned int(64) creation_time;
        unsigned int(64) modification_time;
        unsigned int(32) timescale;
        unsigned int(64) duration;
    } else { // version==0
        unsigned int(32) creation_time;
        unsigned int(32) modification_time;
        unsigned int(32) timescale;
        unsigned int(32) duration;
    }
    template int(32) rate = 0x00010000; // typically 1.0
    template int(16) volume = 0x0100; // typically, full volume
    const bit(16) reserved = 0;
```

```
    const unsigned int(32)[2] reserved = 0;
    template int(32)[9] matrix = { 0x00010000, 0, 0, 0, 0x00010000, 0, 0, 0,
        0x40000000 }; // Unity matrix
    bit(32)[6] pre_defined = 0;
    unsigned int(32) next_track_ID;
}
```

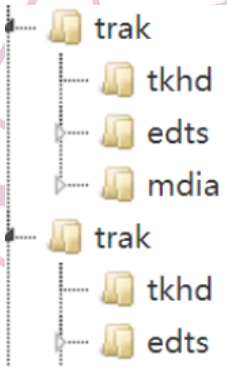
根据标准定义，moov box 属于 Full Box，其 Header 部分除了 size（4 字节），type(4 字节) 外，还额外包含了 version(1 字节), flags(3 字节)。

4. Track Box

- 类型：trak
- 父容器：moov
- 是否必须有：yes
- 数量：一个或者多个

Listing 4: Track Box 类

```
aligned(8) class TrackBox extends Box('trak') {
}
```



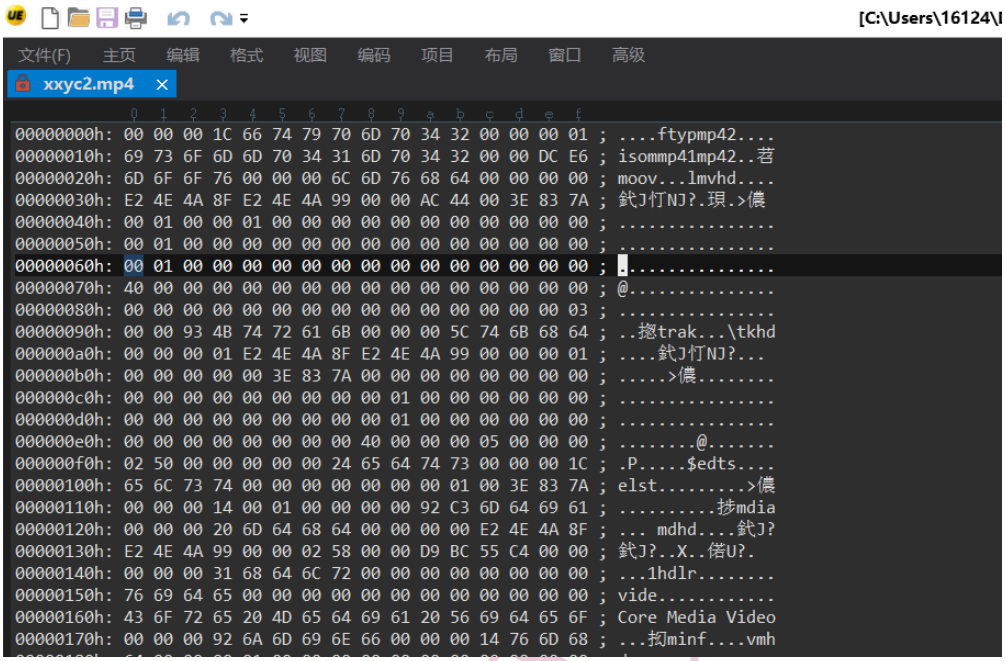
一个 mp4 媒体是由一个或者多个 track 组成，如音频 track，视频 track，每一个 track 都携带有自己的时间和空间信息，每个 track 都相互独立于其他 track。video track：包含了视频 Sample，audio Track 包含了 audio sample，hint track 稍有不同，它描述了一个流媒体服务器如何把文件中的媒体数据组成符合流媒体协议的数据包，如果文件只是本地播放，可以忽略 hint track，他们只与流媒体有关。

track 有两种用途：a. 包含媒体数据 (media tracks)。b. 用来存放分包信息，用于流传输协议 (hint track)。

在一个标准的 mp4 文件中，至少应该有一个 media track，同时所有有助于 hint track 的 media track 都应该保留在文件中，即使这些 media track 没有被 hint track 引用。本文不对 hint track 进行讨论。

（四） 实例分析

用 UltraEdit 打开实验用的 mp4 文件。



查看 mp4 文件属性：

下面详细分析 UE 中的字节流，ftyp 盒子和 moov 盒子是 mp4 文件中必不可少的盒子：

ftyp 盒子

1. `00 00 00 1c`: 这是 ftyp 盒子的大小，表示紧随其后的 ftyp 盒子的总长度是 28 个字节。
2. `66 74 79 70` (`ftyp`): 这是 ftyp 盒子的类型标识，ASCII 码为 ftyp。
3. `6D 70 34 32` (`mp42`): 这是文件的主要品牌，ASCII 码为 mp42，表示这个 MP4 文件遵循 MPEG-4 Part 14 版本 2 的规范。
4. `00 00 00 01`: 这是文件的次要版本号。
5. `69 73 6F 6D` (`isom`): 这表示一个兼容的品牌，ASCII 码为 isom。
6. `6D 70 34 31` (`mp41`): 又一个兼容品牌，ASCII 码为 mp41。

moov 盒子

1. `6D 6F 6F 76` (`moov`): 这是 moov 盒子的类型标识，ASCII 码为 moov。
2. `00 00 00 6C`: 这是 moov 盒子的大小，表示紧随其后的 moov 盒子的总长度是 108 个字节。
3. `6D 76 68 64` (`mvhd`，子盒子类型 `1`): 这是 mvhd 盒子的类型标识，ASCII 码为 mvhd，它是 moov 盒子中的一个子盒子。
4. `80 00 00 E2 4E 4A 8F E2 4E 4A 99`: 这部分是 mvhd 盒子的数据，通常包含了影片的创建时间和修改时间等信息。
5. `00 60 AC 44` 和 `00 3E 83 7A`: 这部分数据通常包括影片的时间尺度和时长，但具体的值需要转换成标准的时间格式才能理解。
6. 后续的 `00` 可能表示填充或者其他盒子的开始。
7. `74 72 61 6B` (`trak`，子盒子类型 `2`): 这个是 trak 盒子的类型标识，ASCII 码为 trak，所分析的文件中共有两个 trak 盒子。
8. 此后的数据不再赘述，是 trak 盒子中包含的 tkhd、edts 以及 mdia 盒子。

三、 可能的隐藏位置和隐藏方法

(一) 隐藏位置

1. 采样点中的最低有效位（LSB）

在一个 16 位的音频样本中，可以使用其中的最低有效位，来嵌入隐藏信息。这种方法在不改变音频质量的同时，提供了很高的隐蔽性。

2. 时间差异

采样点之间的时间差异（Time-Domain）：在音频数据的时间域中，可以通过微调每个采样点之间的时间差异，来嵌入隐藏信息。这种方法可以提供相对较高的容量和隐蔽性，但会对音频的质量产生一定的影响。

3. 频谱成分

在音频数据的频域中，可以嵌入隐藏信息，比如使用小幅度的频率偏移或频率增强，来表示隐藏信息。这种方法可以提供相对较高的容量和隐蔽性，但需要一些高级的信号处理技术。

4. 频率颠簸隐藏

在音频文件中插入隐藏数据的一种方法是通过微调音频信号中的频率来实现。这种方法通常会在音频信号的高频段中插入隐藏信息，这样就可以避免影响听觉质量。

(二) 隐藏方法

1. 基于 DCT 的方法

基于 DCT 方法的图像信息隐藏和提取是一种比较高级的方法。它的原理是将要隐藏的信息嵌入到载体图像的 DCT 系数中，从而实现信息的隐藏。这种方法的优点是安全性较高，嵌入的信息容量较大，但是实现较为复杂，需要对图像进行 DCT 变换和量化。

2. 基于文件 IO 的图像信息隐藏和提取

基于文件 IO 的图像信息隐藏和提取是一种比较简单的方法。它的原理是将要隐藏的信息直接写入到图像文件中，从而实现信息的隐藏。这种方法的优点是实现简单，但是安全性较低，易被攻击者检测到和破解。

3. 基于 LSB 方法

最低有效位（LSB）算法是一种最简单、易于实现的音频信息隐藏算法。它利用了音频信号的容错性，将要隐藏的信息转换为二进制格式，然后将每个二进制位插入到音频采样值的最低有效位中，即将最低位的值替换为要嵌入的信息。由于音频的每个采样值都是由多个位组成的，因此通过替换最低有效位，可以实现较小的嵌入容量，但几乎不会对音频质量产生明显的影响。

该算法的优点是易于实现和嵌入容量较大，但缺点也显而易见，容易被检测和破解，而且如果嵌入量过大，会导致音频质量下降。

4. 回声隐藏算法

回声隐藏算法是一种利用了人类听觉局限性的算法，它将要隐藏的信息分成多个部分，并将它们插入到音频信号的反射声中。由于人类听觉系统会过滤掉较弱的反射声，因此这些隐藏的信息在听觉上不易察觉。该算法的优点是隐蔽性高，不易被检测，但缺点是嵌入容量较小。

5. 相位隐藏算法

相位隐藏算法通过改变音频信号的相位来嵌入隐藏信息。由于人类听觉系统对相位变化不敏感，因此这种方法可以实现较高的隐蔽性和嵌入容量，但会对音频质量产生一定的影响。相位隐藏算法可以应用于不同的音频信号域，例如时域、频域或小波域，需要一定的信号处理技术和算法。

6. 傅氏变换域算法

傅氏变换是将一个函数从时域（time domain）转换到频域（frequency domain）的一种方法。傅氏变换域算法是一种在傅氏变换域下进行信息隐藏的方法。首先将音频信号进行傅氏变换，然后将隐藏信息嵌入到傅氏变换后的频谱中，最后通过逆傅氏变换将嵌入了信息的音频信号恢复出来。

7. 小波变换域算法

小波变换域算法通过将音频信号转换到小波域中，在小波系数中嵌入隐藏信息。小波变换可以在多个尺度上对音频信号进行分解，因此可以提供更大的嵌入容量和隐蔽性，但需要一些高级的信号处理技术。首先将音频信号进行小波变换，然后将隐藏信息嵌入到小波变换后的系数中，最后通过逆小波变换将嵌入了信息的音频信号恢复出来。

需要注意的是，无论使用何种方法，都应该考虑到隐藏信息对音频质量的影响，以及隐藏信息的可靠性和安全性。

四、 算法实现

(一) 基于 LSB 的方法

这个算法的信息隐藏原理基于 LSB（最低有效位）隐写术。该技术通过改变图像像素的最低位来隐藏信息，这些变化对肉眼几乎是不可见的，因为它们对像素的颜色影响极小。在隐藏信息时，每个字符的二进制表示（例如"Hello"）被转换成二进制位，然后逐个嵌入到帧的像素中。在提取时，反向读取这些位并重新组合成原始的文本信息。由于这些更改在视觉上不明显，所以这种方法可以在不显著影响视频质量的情况下隐藏信息。

1. 代码实现

Listing 5: hide_message_in_framee 函数-信息嵌入

```python
def hide_message_in_frame(frame, message):
    n_bytes = frame.shape[0] * frame.shape[1] * 3 // 8
    if len(message) > n_bytes:
        raise ValueError("Error: Insufficient bytes, need bigger frame or
            less data.")

    message_binary = message_to_bin(message) + '1111111111111110'  # 添加停止
        标记
    index = 0
    for i in range(frame.shape[0]):
        for j in range(frame.shape[1]):
            for k in range(3):  # 遍历每个像素的RGB通道
                if index < len(message_binary):
                    frame[i, j, k] = int(bin(frame[i, j, k])[2:-1] +
                        message_binary[index], 2)
                    index += 1
                if index == len(message_binary):
                    return frame
    return frame
```

1. 计算可用空间：首先计算图像能够嵌入信息的最大字节数，这是基于图像的像素数和每个像素 RGB 三通道的容量。

2. 信息转换：将文本信息转换为二进制字符串，并在二进制字符串的末尾添加一个特定的停止标记'1111111111111110'，以便之后能够识别信息的终点。

3. 信息嵌入：遍历图像的每个像素的 RGB 通道，将二进制信息的每一位嵌入到像素的最低有效位（LSB）。通过将像素值的二进制表示的最低位替换为信息位来完成。

4. 终止条件：一旦所有的信息位都被嵌入，或者遍历完所有像素后，函数返回修改后的图像。

Listing 6: extract_message_from_frame 函数-信息提取

```python
def extract_message_from_frame(frame):
    binary_str = ''
    for i in range(frame.shape[0]):
        for j in range(frame.shape[1]):
            for k in range(3):  # 遍历每个像素的RGB通道
                binary_str += bin(frame[i, j, k])[-1]  # 提取最低有效位
                if len(binary_str) >= 8:
                    last_byte = binary_str[-8:]  # 每次检查最新的8位
                    if last_byte == '11111110':  # 检查停止标记
                        return bin_to_message(binary_str[:-8])  # 返回结果,
                            排除停止标记
    return bin_to_message(binary_str)  # 如果没有找到停止标记, 返回整个消息
```

1. 初始化二进制字符串：定义一个空字符串 binary_str 用来存储从图像像素中提取的二进制信息。
2. 遍历像素的 RGB 通道：循环遍历图像的每一个像素的 RGB 三个颜色通道。
3. 提取最低有效位：对于每个颜色通道的像素值，使用 bin() 函数将像素值转换为二进制格式，然后使用 [-1] 索引提取最低有效位（LSB），并将这个位添加到 binary_str 字符串中。
4. 检查停止标记：在累加二进制字符串的过程中，每次添加一个新的位后，检查最新的 8 位（last_byte = binary_str[-8:]）是否等于'11111110'，这是预先定义的停止标记。如果检测到停止标记，表明信息已经完全提取。
5. 返回提取的信息：使用 bin_to_message() 函数，将收集到的二进制字符串（除去停止标记）转换成字符形式的信息，并返回这个信息。

Listing 7: 主函数逻辑

```python
# 读取视频
cap = cv2.VideoCapture('input.mp4')
ret, frame = cap.read()  # 读取第一帧
cap.release()

# 隐藏信息
secret_message = "Hello, this is a hidden message!"
frame = hide_message_in_frame(frame, secret_message)

# 展示修改后的帧
cv2.imshow('Modified Frame', frame)
cv2.waitKey(0)
cv2.destroyAllWindows()

# 使用前面修改过的帧
extracted_message = extract_message_from_frame(frame)
print("Extracted Message:", extracted_message)
```

2. 举例

假设我们想在一个视频帧中隐藏信息”OK”:

1. 信息转换: 首先, 将”OK” 转换为二进制。”O” 的 ASCII 值是 79, 对应的二进制是 01001111; ”K” 的 ASCII 值是 75，对应的二进制是 01001011。合起来就是 0100111101001011。
2. 隐藏信息：选择一个视频的第一帧。假设这帧的像素值为随机值，我们从左上角开始，按照 RGB 顺序修改最低有效位。比如, 如果第一个像素的 R 通道值是 156（二进制为 10011100），要嵌入’0’，则将其改为 10011100（保持不变）。如果要嵌入’1’，则改为 10011101。
3. 提取信息：读取修改过的帧，提取每个像素的 RGB 通道的最低有效位，合并得到二进制串。从这串中恢复出原始信息”OK”。
4. 这个过程是通过微小地调整像素值的最低有效位来隐藏信息, 对视频的可见影响非常小, 一般人肉眼难以察觉。

3. 运行效果

图 1: 未修改的第一帧

图 2: 修改后的第一帧

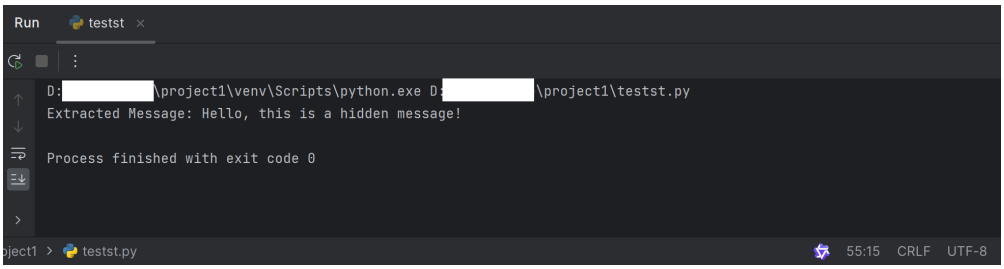


图 3: 提取出的信息

(二) 基于帧间差异的方法

本方法的核心是通过修改视频帧中的像素值来隐藏信息。通过在第二帧中增加一个基于信息的二进制位的差异值（乘以 10 以增加差异的可视性），实现了信息的隐藏。这种方法基于假设观看者不会注意到细微的像素变化。提取信息时，通过计算两帧之间的像素差异，并根据预设的阈值将这些差异转换回二进制信息，最终解码出原始文本信息。

1. 代码实现

Listing 8: embed_message 函数-信息嵌入

```python
def embed_message(frame1, frame2, message):
    binary_message = ''.join(format(ord(char), '08b') for char in message)
    index = 0
    for i in range(frame1.shape[0]):
        for j in range(frame1.shape[1]):
            if index < len(binary_message):
                diff = int(binary_message[index]) * 10  # 使用10作为差异基数
                frame2[i, j, 0] = frame1[i, j, 0] + diff
                index += 1
            else:
                break
    return frame2
```

1. 二进制转换：首先将输入的文本信息转换为二进制字符串。
2. 信息嵌入：通过遍历视频帧（frame1）的每个像素，并将二进制数据嵌入到另一个帧（frame2）中。嵌入的方式是在 frame1 对应像素的基础上加上一个基于二进制位的差值（差值 = 二进制位 × 10），这样修改 frame2 的相应像素值。
3. 循环和索引管理：使用索引变量 index 追踪当前嵌入的二进制位。如果所有的二进制信息都已嵌入，或者已遍历所有像素，则停止处理。

Listing 9: extract_message 函数-信息提取

```python
def extract_message(frame1, frame2):
    binary_message = ''
    for i in range(frame1.shape[0]):
        for j in range(frame1.shape[1]):
```

```
            diff = frame2[i, j, 0] - frame1[i, j, 0]
            bit = '1' if diff > 5 else '0'
            binary_message += bit
            if len(binary_message) >= 8 and binary_message[-8:] == '00000000'
                :
                break
        else:
            continue
        break
    message = ''.join(chr(int(binary_message[i:i+8], 2)) for i in range(0,
        len(binary_message) - 8, 8))
    return message
```

1. 计算像素差异：遍历每个像素，计算 frame1 和 frame2 同一位置像素的差异（diff）。这个差异是在隐藏信息时添加的。
2. 差异转换为二进制位：根据差异的大小确定对应的二进制位是'1' 还是'0'。如果差异大于 5，则认为该位是'1'；否则是'0'。
3. 检测停止标志：连续的 8 个'0' 被用作停止标志，表示消息的结束。检测到停止标志时，停止进一步读取二进制位。
4. 二进制转换为文本：将收集到的二进制字符串（除停止标志）转换回人可读的文本信息。

Listing 10: 主函数逻辑

```
# 读取视频
cap = cv2.VideoCapture('input.mp4')
ret, frame1 = cap.read()
ret, frame2 = cap.read()

# 隐藏信息
secret_message = "Hello, this is a hidden message!"
modified_frame = embed_message(frame1, frame2, secret_message)

# 输出观看
cv2.imshow('Frame 1', frame1)
cv2.imshow('Frame 2', modified_frame)
cv2.waitKey(0)  # 等待按键
cv2.destroyAllWindows()

# 写入修改后的视频
fourcc = cv2.VideoWriter_fourcc(*'mp4v')
out = cv2.VideoWriter('output.mp4', fourcc, 20.0, (frame1.shape[1], frame1.
    shape[0]))

# 添加帧到输出视频
out.write(frame1)
out.write(modified_frame)
```

```
23
24 # 读取更多帧并添加到视频
25 while True:
26     ret, frame = cap.read()
27     if not ret:
28         break
29     out.write(frame)
30
31 # 清理资源
32 cap.release()
33 out.release()
34
35 # 提取信息测试
36 print("Extracted Message:", extract_message(frame1, modified_frame))
```

2. 举例

如果想在一个视频中隐藏信息”Hi”，使用的是上面提到的代码。首先，将”Hi” 转换成二进制形式：’H’ 是 01001000，’i’ 是 01101001，总共就是 0100100001101001。

1. 嵌入信息：对于第一帧和第二帧中的每一个像素，代码会检查信息（”Hi”）的每一个二进制位。对于每个’1’，在第二帧的相应像素中增加 10；对于每个’0’，则不变。这样，就在第二帧的像素中创建了一种模式，对比第一帧有细微差异。

2. 提取信息：在提取过程中，检查第一帧和第二帧之间的像素差异。如果差异大于 5，就记录’1’；如果差异小于或等于 5，就记录’0’。从这些差异中恢复出二进制字符串 0100100001101001，最后将这些二进制转换回字符，得到原始信息”Hi”。

3. 运行效果

图 4: 第一帧

图 5: 修改后的第二帧

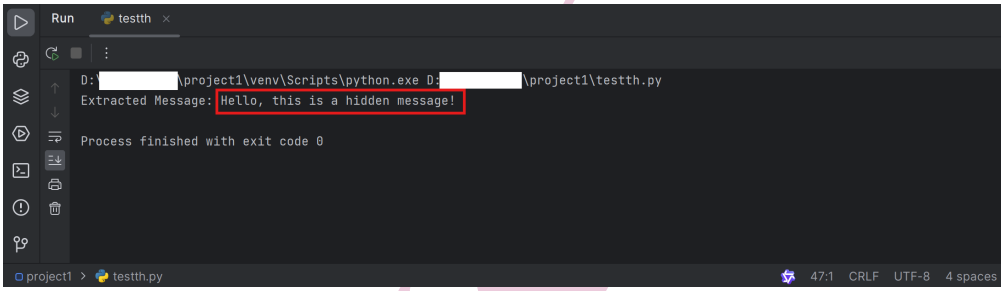


图 6: 提取出的信息