一、总体要求

以智慧城市校园安防为真实工程背景，围绕信息的“采集→处理→传输→计算→存储→应用”全闭环流程，构建基于**国产根技术**+AI的音视频**认证**系统。系统采用发送端与接收端分离架构，通过四层递进任务（基础层→安全层→溯源层→智能层）引导学生逐步完成从单一技能训练到综合系统能力培养的梯度跃升。

1.1 总体架构

要求学生独立设计并实现一套完整的音视频**认证**系统，具体见表 1.1。

表 1.1 总体内容

模块	核心功能	涉及技术
发送端（鲲鹏开发板）	音视频采集、人脸检测、说话人特征提取、水印嵌入、SM4 加密、编码压缩、网络传输	OpenCV、YOLO、FFmpeg、鲁棒水印算法、SM4 国密算法、TCP/UDP Socket
接收端（学生笔记本电脑）	数据接收、解码分离、SM4 解密、水印提取验证、多模态认证、数据库存储、音视频播放	FFmpeg、SM4、人脸识别、说话人识别、openGauss 数据库、音视频同步技术

如图 1.1 所示，本系统采用发送端与接收端分离架构，形成完整的“采集→处理→传输→计算→存储→应用”闭环。

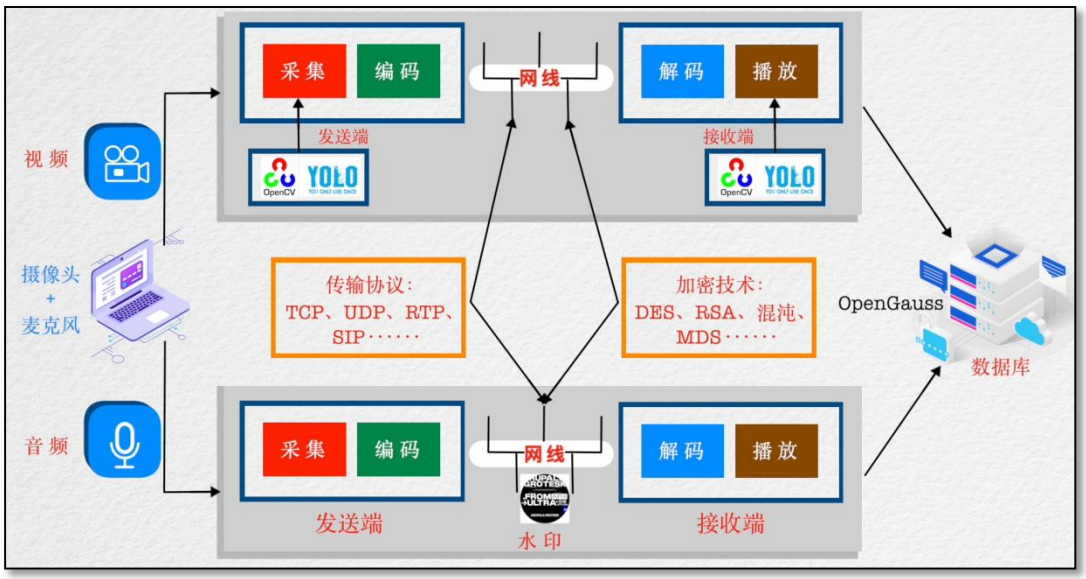

图 1.1 技术路线图

发送端基于鲲鹏开发板运行，首先通过 FFmpeg 完成音视频采集，利用 OpenCV 和 YOLO 模型对视频帧进行人脸检测并生成元数据，在此基础上，通过 DCT 域鲁棒水印算法嵌入溯源信息，并采用 SM4 国密算法对敏感数据（人脸特征、水印信息等）进行加密，随后对音视频流进行 H.264/AAC 编码压缩以减轻

传输压力，最终通过 TCP 传输关键元数据、UDP 传输实时音视频流，实现安全可靠的网络发送。

接收端基于学生笔记本电脑，首先通过 TCP/UDP 双协议接收数据，利用 FFmpeg 解码分离音视频流与加密元数据，随后通过 SM4 解密还原敏感信息，并对视频帧进行水印提取与完整性验证；接着进入多模态认证环节，分别通过人脸识别与说话人识别提取特征，结合环境光照、信噪比等指标动态调整融合权重，实现交叉验证；认证通过的记录与元数据存入 openGauss 国产数据库，同时基于时间戳同步机制完成音视频的实时播放与交互。

系统可采用 C++为主体、Python 为辅的异构混合编程架构。底层性能敏感模块（音视频采集/编解码、人脸检测 NPU 推理、SM4 国密加密、网络传输等）建议采用 C++实现，以充分利用鲲鹏硬件加速能力和实时性要求；算法原型验证和测试分析模块（模型量化校准、水印鲁棒性测试等）建议采用 Python 实现，以提升算法迭代效率。两种语言可通过 ONNX 模型格式（人脸检测）和独立测试脚本（水印测试）实现松耦合集成，互不影响。这一设计既可保证系统运行效率，又可兼顾算法探索的灵活性，是国产根技术平台上的标准工程实践。

1.2 任务分层设置

为体现因材施教与挑战度，设置四层递进式任务，如表 1.2 所示，基础层考查编解码与网络编程能力，安全层强化国密算法应用，溯源层聚焦鲁棒水印嵌入与抗攻击设计，智能层要求实现多模态识别融合与动态身份认证。学生可根据自身基础选择起点，实现从单一技能训练到综合系统能力培养的梯度跃升。

表 1.2 任务分解

层次	任务目标	系统能力培养点	成绩上限
基础层	实现音视频采集、编解码与网络传输	掌握 FFmpeg、Socket 编程，理解端到端通信机制	70
安全层	增加 SM4 国密算法加密传输	强化密码学应用，理解数据安全与系统防护	80
溯源层	嵌入鲁棒水印，实现泄密溯源	掌握数字水印技术，培养系统安全设计意识	90
智能层	融合人脸识别与说话人识别，实现多模态身份认证	综合运用 AI 技术，构建复杂决策系统	100

四层递进任务按层次设定成绩上限，基础层最高 70 分，安全层 80 分，溯源层 90 分，智能层 100 分，鼓励学生挑战高阶任务。从系统视角出发，统筹考虑

各模块的接口设计、性能优化与整体集成，力争最终可以交付一个“跑得通、看得见、可验证”的国产化音视频传输认证系统。

1.3 发送端设计内容

1.3.1 音视频采集（基础层，必选）

（1）任务目标

在鲲鹏开发板上，通过 FFmpeg 库实现摄像头和麦克风的实时数据采集，获取原始音视频帧，为后续处理提供数据源。

（2）硬件准备

- 开发板：香橙派鲲鹏 Pro（预装 openEuler 操作系统）。
- 摄像头：USB 摄像头（支持 UVC 协议）或 CSI 接口摄像头。
- 麦克风：USB 麦克风或 3.5mm 接口麦克风。
- 连接方式：摄像头插入 USB 口，麦克风插入音频口或 USB 口。

（3）环境配置

确保 FFmpeg 库已正确安装并支持设备输入。

```
#安装 FFmpeg 及开发库
sudo apt update
sudo apt install ffmpeg libavcodec-dev libavformat-dev libavdevice-dev libavutil-dev
#验证摄像头是否被识别
ls /dev/video*
#若输出 /dev/video0，表示摄像头已识别
# 测试摄像头采集（可选）
ffplay /dev/video0
#若弹出视频窗口，表示摄像头工作正常
#测试麦克风采集（可选）
arecord -l
#列出音频设备，找到 card 和 device 编号，如 hw:0,0
```

（4）视频采集示例

利用 FFmpeg 库实现视频的采集。

```
#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include <libavutil/avutil.h>
#include <libavdevice/avdevice.h>
int main(int argc, char* argv[]) {
    AVFormatContext *pFormatCtx = NULL;
    AVDictionary *options = NULL;
```

```
        int video_stream_index = -1;
        int audio_stream_index = -1;
        // 1. 注册所有设备（必须）
        avdevice_register_all();
        // 2. 配置视频采集参数
        AVInputFormat *video_fmt = av_find_input_format("v4l2");
        if (!video_fmt) {
            av_log(NULL, AV_LOG_ERROR, "Cannot find v4l2 input format\n");
            return -1;
        }
        // 3. 设置视频采集选项
        av_dict_set(&options, "video_size", "1280x720", 0);          // 分辨率
        av_dict_set(&options, "framerate", "30", 0);                  // 帧率
        av_dict_set(&options, "pixel_format", "yuv420p", 0);          // 像素格式（推荐 yuv420p）
        av_dict_set(&options, "input_format", "mjpeg", 0);            // 输入格式（部分摄像头支持 mjpeg）
        // 4. 打开视频设备
        const char *video_device = "/dev/video0";
        if (avformat_open_input(&pFormatCtx, video_device, video_fmt, &options) != 0) {
            av_log(NULL, AV_LOG_ERROR, "Cannot open video device %s\n", video_device);
            return -1;
        }
        // 5. 查找视频流
        for (unsigned int i = 0; i < pFormatCtx->nb_streams; i++) {
            if (pFormatCtx->streams[i]->codecpar->codec_type == AVMEDIA_TYPE_VIDEO) {
                video_stream_index = i;
                break;
            }
        }
        if (video_stream_index == -1) {
            av_log(NULL, AV_LOG_ERROR, "Cannot find video stream\n");
            return -1;
        }
        // 6. 打印视频流信息
        av_dump_format(pFormatCtx, 0, video_device, 0);
        // 7. 获取视频解码器
        AVCodec *pCodec = avcodec_find_decoder(pFormatCtx->streams[video_stream_index]->codecpar->codec_id);
        AVCodecContext *pCodecCtx = avcodec_alloc_context3(pCodec);
        avcodec_parameters_to_context(pCodecCtx, pFormatCtx->streams[video_stream_index]->codecpar);
```

```
    avcodec_open2(pCodecCtx, pCodec, NULL);
    // 8. 循环读取视频帧
    AVPacket *packet = av_packet_alloc();
    AVFrame *frame = av_frame_alloc();
    int frame_count = 0;
    while (frame_count < 300) {   // 采集 300 帧（约 10 秒）
        if (av_read_frame(pFormatCtx, packet) < 0) {
            break;
        }
        if (packet->stream_index == video_stream_index) {
            // 解码视频帧
            avcodec_send_packet(pCodecCtx, packet);
            while (avcodec_receive_frame(pCodecCtx, frame) == 0) {
                frame_count++;
                av_log(NULL, AV_LOG_INFO, "Captured frame %d: %dx%d\n",
                       frame_count, frame->width, frame->height);
                // TODO: 将 frame 送入下一模块（人脸检测、编码压缩等）
            }
        }
        av_packet_unref(packet);
    }
    // 9. 释放资源
    av_packet_free(&packet);
    av_frame_free(&frame);
    avcodec_free_context(&pCodecCtx);
    avformat_close_input(&pFormatCtx);
    av_dict_free(&options);
    return 0;
}
```

（5）采集音频示例

若同时采集音频，可创建第二个 AVFormatContext，使用 alsa 输入格式。

```
//音频采集（使用 ALSA）
AVFormatContext *pAudioCtx = NULL;
AVInputFormat *pAudioFmt = av_find_input_format("alsa");
//设置音频参数
AVDictionary *audioOpts = NULL;
av_dict_set(&audioOpts, "sample_rate", "44100", 0);
av_dict_set(&audioOpts, "channels", "2", 0);
av_dict_set(&audioOpts, "sample_fmt", "s16", 0);
//打开麦克风设备（hw:0,0 是第一个声卡，需根据实际情况调整）
if (avformat_open_input(&pAudioCtx, "hw:0,0", pAudioFmt, &audioOpts) != 0) {
    printf("无法打开麦克风\n");
}
```

```
AVPacket pkt;
while (av_read_frame(pAudioCtx, &pkt) >= 0) {
    // 此时 pkt.data 就是 PCM 数据（16 位整型，交错格式）
    uint8_t* pcm_data = pkt.data;
    int data_size = pkt.size;
    // 处理 PCM 数据（例如提取说话人特征，嵌入水印）
    av_packet_unref(&pkt);  // 释放 packet 内部引用
}
```

这里采集到的就是原始的 PCM 格式音频数据（16 位有符号整型，小端字节序，立体声）。可以直接对这个 pkt->data 中的 PCM 数据进行后续处理（如说话人特征提取，音频水印嵌入等）。

（6）常见问题与优化建议

香橙派鲲鹏 Pro 的 NPU 可用于图像预处理，参考官方 RGA（图形加速引擎）替代 OpenCV 进行格式转换，对于高分辨率场景，可考虑使用 MIPI CSI 摄像头替代 USB 摄像头以获得更高带宽，其他问题见表 1.3。

表 1.3 音视频采集常见问题与解决思路

问题现象	可能原因	解决思路
无法打开 /dev/video0	权限不足	sudo chmod 666 /dev/video0 或将用户加入 video 组
采集帧率低于预期	分辨率过高或 USB 带宽限制	降低分辨率（如 640x480）或使用 MIPI 摄像头
画面卡顿或丢帧	CPU 负载过高	使用硬件加速（鲲鹏 NPU）或降低采集帧率
音频不同步	音视频时钟未对齐	在采集时记录 PTS 时间戳，后续同步处理

通过以上详细步骤，学生可以基于香橙派鲲鹏开发板完成音视频采集模块的开发，为后续的人脸检测、编码压缩等环节奠定基础。实践中鼓励学生多尝试、多调试，遇到问题可查阅 FFmpeg 官方文档或课程群交流。

1.3.2 编码压缩（基础层，必选）

（1）环境准备

香橙派鲲鹏开发板搭载的鲲鹏处理器内置了专用的视频处理单元（VPU），支持 H.264/H.265 硬件编码加速。需要先安装必要的多媒体驱动和库：

```
#添加 Rockchip 多媒体 PPA（适用于基于 Rockchip 的鲲鹏 Pro）
sudo add-apt-repository ppa:liujianfeng1994/rockchip-multimedia
sudo apt update
#安装多媒体配置和 VPU 驱动
```

```
sudo apt install rockchip-multimedia-config
#安装 FFmpeg 及硬件加速支持
sudo apt install ffmpeg libavcodec-extra
```

验证硬件编码器可用性，预期输出应包含 h264_rkmpp 或 h264_omx 等硬件编码器选项。

```
# 查看 FFmpeg 支持的编码器，确认是否存在 h264_rkmpp（Rockchip 硬件编码器）
ffmpeg -encoders | grep -E "h264|hevc|mpp"
```

（2）编码参数设置

视频可采用 H.264 格式，码率控制为 CBR 模式 1.5Mbps；音频可采用 AAC 格式，比特率 128kbps。其他关键参数设置可参考表 1.4。

表 1.4 视频元数据格式

参数类别	参数	值	说明
视频编码器	codec	libx264 或 h264_rkmpp	软件编码/硬件加速编码
视频码率	b:v	1500k	目标码率 1.5Mbps
CBR 模式	minrate/ maxrate	1500k/1500k	固定码率，与 b:v 保持一致
码率缓冲区	bufsize	1500k	与码率匹配，避免波动
GOP 大小	g	30	每 30 帧一个 I 帧（约 1 秒）
编码速度预设	preset	fast	平衡编码速度与压缩率
音频编码器	c:a	aac	AAC 编码
音频码率	b:a	128k	128kbps
音频采样率	ar	44100	44.1kHz
音频声道	ac	2	双声道立体声

需要注意的是，要实现真正的恒定码率（CBR），必须同时设置-b:v、-minrate、-maxrate 为相同值，并配合-bufsize，否则 FFmpeg 默认使用可变码率（VBR）模式。

（3）编码示例

参考如下思路将从摄像头采集的原始帧（YUV420P 格式）进行编码压缩。

```
#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include <libavutil/opt.h>
// 视频编码器初始化
AVCodecContext* init_video_encoder(int width, int height, int fps, int bitrate) {
    // 1. 查找编码器（优先使用硬件编码器）
    const AVCodec *codec = avcodec_find_encoder_by_name("h264_rkmpp");
    if (!codec) {
        // 回退到软件编码器
        codec = avcodec_find_encoder(AV_CODEC_ID_H264);
```

```
        if (!codec) {
            fprintf(stderr, "H.264 encoder not found\n");
            return NULL;
        }
        printf("Using software encoder: libx264\n");
    } else {
        printf("Using hardware encoder: h264_rkmpp\n");
    }
    // 2. 分配编码器上下文
    AVCodecContext *ctx = avcodec_alloc_context3(codec);
    if (!ctx) return NULL;
    // 3. 设置编码参数
    ctx->width = width;                 // 1280
    ctx->height = height;               // 720
    ctx->time_base = (AVRational){1, fps};   // 帧率 30fps
    ctx->framerate = (AVRational){fps, 1};
    ctx->pix_fmt = AV_PIX_FMT_YUV420P;
    // 码率控制：CBR 模式
    ctx->bit_rate = bitrate;                // 1500000 bps
    ctx->rc_min_rate = bitrate;             // 最小码率
    ctx->rc_max_rate = bitrate;             // 最大码率
    ctx->rc_buffer_size = bitrate;          // 缓冲区大小
    ctx->rc_initial_buffer_occupancy = bitrate * 0.5;   // 初始缓冲占用
    // GOP 设置：每 30 帧一个 I 帧
    ctx->gop_size = 30;
    ctx->keyint_min = 30;                   // 最小关键帧间隔
    // 编码速度预设（仅软件编码器支持）
    if (strcmp(codec->name, "libx264") == 0) {
        av_opt_set(ctx->priv_data, "preset", "fast", 0);      }

    // 4. 打开编码器
    if (avcodec_open2(ctx, codec, NULL) < 0) {
        avcodec_free_context(&ctx);
        return NULL;
    }

    return ctx;
}

// 音频编码器初始化
AVCodecContext* init_audio_encoder(int sample_rate, int channels, int bitrate) {
    // 1. 查找 AAC 编码器
    const AVCodec *codec = avcodec_find_encoder(AV_CODEC_ID_AAC);
    if (!codec) {
```

```
            fprintf(stderr, "AAC encoder not found\n");
            return NULL;
        }
        // 2. 分配编码器上下文
        AVCodecContext *ctx = avcodec_alloc_context3(codec);
        if (!ctx) return NULL;
        // 3. 设置编码参数
        ctx->sample_fmt = AV_SAMPLE_FMT_FLTP;  // AAC 需要平面浮点格式
        ctx->sample_rate = sample_rate;              // 44100 Hz
        ctx->channels = channels;                    // 2 立体声
        ctx->channel_layout = av_get_default_channel_layout(channels);
        ctx->bit_rate = bitrate;                     // 128000 bps
        // 4. 打开编码器
        if (avcodec_open2(ctx, codec, NULL) < 0) {
            avcodec_free_context(&ctx);
            return NULL;
        }

        return ctx;
    }
```

编码后的音视频流可以采用裸流格式（用于实时传输），输出为纯 H.264 视频流（.h264）和纯 AAC 音频流（.aac），便于通过 UDP/RTP 实时传输，文件体积小，无需封装开销。

（4）常见问题排查

具体见表 1.5。

表 1.5 编码常见问题与解决思路

问题现象	可能原因	解决思路
硬件编码器不可用	VPU 驱动未正确安装	重新安装 rockchip-multimedia-config
编码后视频无法播放	缺少 I 帧或 PTS 不正确	确保 GOP 设置正确，添加时间戳
码率波动过大	CBR 参数设置不完整	同时设置-b:v、-minrate、-maxrate
编码延迟过高	编码速度预设过慢	使用 preset=ultrafast 或硬件编码器

通过以上步骤，学生可以在香橙派鲲鹏开发板上实现完整的音视频编码压缩功能。编码后的数据量大幅降低（原始 YUV420P 约 1.4Mbps 的 30fps 视频约 132MB/s，压缩后仅 1.5Mbps 约 0.19MB/s），有效缓解网络传输压力。

1.3.3 网络传输（基础层，必选）

在实时音视频传输系统中，传输层协议的选择直接决定了系统的实时性和可靠性。采用 UDP 传输音视频数据+TCP 传输控制信令与元数据的双通道策略，既保证了实时媒体的低延迟传输，又确保了关键控制信息的可靠送达。

在 UDP 传输中，采用裸流传输方案，不对音视频数据做额外的应用层封装（如 RTP），直接将 H.264 视频流和 AAC 音频流以 UDP 数据报的形式发送。需要注意的是，裸流传输不具备 RTP 的序列号和负载类型标识功能，在丢包发生时，接收端只能通过解析 NAL 单元边界来判断数据完整性。但在开发板与学生笔记本电脑直连环境下，丢包率较低，裸流方案完全满足需求。若需增强弱网对抗能力，可将发送端升级为 RTP 封装（采用开源库如 jrtplib）。

发送端位于鲲鹏开发板上，采用多线程架构分别处理视频流、音频流、元数据的发送任务，三者可并行执行，互不阻塞，如表 1.6 所示。

表 1.6 数据通道建议分工表

数据类型	传输协议	端口建议	说明
视频流（H.264）	UDP	5004	实时传输，允许偶发丢包
音频流（AAC）	UDP	5006	实时传输，允许偶发丢包
加密元数据	TCP	5008	可靠传输，确保特征信息完整送达
控制信令	TCP	5009	会话协商、参数同步等

通过以上详细步骤，学生可以基于香橙派鲲鹏开发板完成音

（1）H.264 视频流的 UDP 封装

H.264 编码后的数据以 NAL 单元（Network Abstraction Layer Unit）为基本单位。每个 NAL 单元由起始码（0x00 0x00 0x00 0x01 或 0x00 0x00 0x01）和 NAL 载荷组成。在 UDP 传输中，推荐每个 UDP 包承载一个完整的 NAL 单元，原因如下：确保接收端能够直接解析，无需处理跨包组帧；避免因一个 UDP 包丢失导致多个 NAL 单元无法解码。

需要注意的是，对于 I 帧等较大的 NAL 单元（可能超过 1500 字节的 MTU 限制），在以太网环境下超过 MTU 时，IP 层会自动分片，应用层无需特殊处理。若需避免 IP 分片带来的额外开销，可在编码器侧设置合适的切片大小。

（2）AAC 音频流的 UDP 封装

AAC 编码器每次输出一个完整的 ADTS 帧（包含 ADTS 头+AAC 原始数据块）。每个 ADTS 帧的典型长度为数百字节（如 128kbps 码率下，每帧约 384 字

节），远小于 MTU 限制。因此，每个 UDP 包直接承载一个完整的 ADTS 帧即可。

（3）视频 UDP 发送线程示例

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <cstring>
#include <thread>
#include <queue>
#include <mutex>
#include <vector>
class VideoUdpSender {
private:
    int sockfd;
    struct sockaddr_in dest_addr;
    std::queue<std::vector<uint8_t>> packet_queue;  // 存储待发送的 NAL 单元
    std::mutex queue_mutex;
    bool running = false;
    std::thread send_thread;
public:
    VideoUdpSender(const std::string& dest_ip, int dest_port) {
        // 1. 创建 UDP socket
        sockfd = socket(AF_INET, SOCK_DGRAM, 0);
        if (sockfd < 0) {
            perror("socket creation failed");
            exit(1);
        }
        // 2. 配置目标地址
        memset(&dest_addr, 0, sizeof(dest_addr));
        dest_addr.sin_family = AF_INET;
        dest_addr.sin_port = htons(dest_port);
        inet_pton(AF_INET, dest_ip.c_str(), &dest_addr.sin_addr);
    }
    ~VideoUdpSender() {
        running = false;
        if (send_thread.joinable()) send_thread.join();
        close(sockfd);
    }
    // 将 NAL 单元加入发送队列
    void enqueuePacket(const uint8_t* data, size_t len) {
        std::lock_guard<std::mutex> lock(queue_mutex);
        packet_queue.push(std::vector<uint8_t>(data, data + len));
```

```
        }
        // 发送线程主循环
        void start() {
            running = true;
            send_thread = std::thread([this]() {
                while (running) {
                    std::vector<uint8_t> packet;
                    {
                        std::lock_guard<std::mutex> lock(queue_mutex);
                        if (!packet_queue.empty()) {
                            packet = std::move(packet_queue.front());
                            packet_queue.pop();
                        }
                    }
                    if (!packet.empty()) {
                        ssize_t sent = sendto(sockfd, packet.data(), packet.size(), 0,
                                              (struct sockaddr*)&dest_addr,
sizeof(dest_addr));
                        if (sent < 0) {
                            perror("sendto failed");
                        }
                    }
                    // 控制发送速率，避免过载
                    std::this_thread::sleep_for(std::chrono::milliseconds(1));
                }
            });
        }
    };
```

在鲲鹏开发板上，建议为视频发送线程设置较高优先级，确保视频流能够及时发送，避免因 CPU 调度延迟导致卡顿。可使用 pthread_setschedparam 设置实时调度策。编码后的数据需要按帧率发送，而不是尽可能快地发送。发送循环中的 sleep(1ms)已提供基本速率控制。更精细的控制可基于帧的 PTS 时间戳计算发送间隔。此外，鲲鹏开发板的内存资源有限，Socket 缓冲区需要合理设置。

（4）音频 UDP 发送线程示例

音频 UDP 发送的实现思路与视频类似，区别在于：目标端口使用独立的端口（如 5006），便于接收端区分视频流和音频流；每个 UDP 包承载一个完整的 ADTS 帧（AAC 编码输出）；音频数据量远小于视频，发送队列长度通常很稳定。

此外，音频编码器 avcodec_receive_packet()每次输出一个完整的 AVPacket，其中包含一个 ADTS 帧。将该包的 data 直接通过 UDP 发送即可，无需额外处理。

（5）元数据 TCP 发送（独立线程）示例

元数据（加密后的人脸特征、声纹特征、水印信息等）采用 TCP 传输，确保可靠送达。发送端实现一个 TCP 客户端，接收端作为 TCP 服务器监听。

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <cstring>
#include <thread>
class MetaDataTcpSender {
private:
    int sockfd;
    bool connected = false;
public:
    bool connectToServer(const std::string& server_ip, int server_port) {
        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        if (sockfd < 0) return false;
        struct sockaddr_in server_addr;
        memset(&server_addr, 0, sizeof(server_addr));
        server_addr.sin_family = AF_INET;
        server_addr.sin_port = htons(server_port);
        inet_pton(AF_INET, server_ip.c_str(), &server_addr.sin_addr);
        if (::connect(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
            perror("TCP connect failed");
            return false;
        }
        // 关键优化：禁用 Nagle 算法，降低小包延迟
        int flag = 1;
        if (setsockopt(sockfd, IPPROTO_TCP, TCP_NODELAY, &flag, sizeof(int)) < 0)
{
            perror("setsockopt TCP_NODELAY failed");
        }
        connected = true;
        return true;
    }
    bool sendMetaData(const uint8_t* data, size_t len) {
        if (!connected) return false;
        // 先发送 4 字节长度头，便于接收端分帧
        uint32_t net_len = htonl(len);
        ssize_t sent = send(sockfd, &net_len, 4, 0);
```

```
            if (sent != 4) return false;
            sent = send(sockfd, data, len, 0);
            return sent == (ssize_t)len;
        }
        ~MetaDataTcpSender() {
            if (connected) close(sockfd);
        }
};
```

这里考虑禁用 Nagle 算法，这是因为 Nagle 算法旨在减少小包数量，会将多个小数据包合并后再发送，这会导致数据发送被延迟等待。对于实时性要求较高的控制信令和元数据传输，禁用 Nagle 算法可以降低延迟，避免因等待 ACK 而引入的不必要等待。使用 setsockopt 设置 TCP_NODELAY 即可实现。

（6）常见问题与解决思路

具体见表 1.7。

表 1.7 传输常见问题与解决思路

问题现象	可能原因	解决思路
接收端收不到 UDP 数据	IP 地址或端口配置错误	检查发送端目标地址和端口；用 tcpdump -i eth0 udp 抓包确认数据是否发出
视频播放卡顿/马赛克	UDP 丢包或乱序	降低视频码率；增大接收端缓冲区；使用有线网络替代 WiFi
TCP 连接建立失败	接收端服务器未启动或端口被占用	确保接收端先启动监听；检查端口可用性；验证防火墙设置
延迟逐渐增大	发送队列积压	检查编码器是否实时；考虑使用无锁队列或增大发送缓冲区
画面与声音不同步	音视频独立传输，未做同步	接收端增加同步逻辑（参考 4.4.2 节音视频同步）

通过以上详细步骤，学生可以基于香橙派鲲鹏开发板完成音视频的传输，为后续的安全层加密、溯源层水印嵌入以及智能层多模态认证奠定坚实的数据通路基础。

1.3.4 SM4 国密加密（安全层，可选）

考虑到实时性要求，不对体积庞大的编码后音视频流进行全量加密，而是对包含人脸特征、说话人特征、水印信息、时间戳等敏感元数据进行 SM4-CBC 模式加密。这既保证了核心身份信息的安全，又最大限度地降低了加密带来的性能开销。加密后的元数据与音视频流并行传输。

（1）环境准备：在鲲鹏开发板上安装 GmSSL

SM4 是我国国家密码管理局发布的分组密码算法标准（GB/T 32907-2016），分组长度和密钥长度均为 128 位（16 字节）。相较于国际通用的 AES 算法，SM4 在算法设计上采用非平衡 Feistel 网络结构，轮函数中包含线性变换和非线性变换（S 盒），具有更高的安全冗余度。选择 SM4 主要基于以下考虑：一是符合国家“核心技术自主可控”战略要求，二是配合鲲鹏开发板的硬件加速能力（鲲鹏 KAE 引擎原生支持 SM4 硬件加速），三是培养学生对国产密码技术的应用能力。

本模块采用 CBC（密码分组链接）模式进行加密，该模式通过引入随机初始化向量（IV），使得即使明文相同，加密后的密文也完全不同，有效抵抗模式泄露攻击。

GmSSL 是由北京大学自主开发的开源密码库，全面支持 SM2/SM3/SM4 等国密算法，适用于包括嵌入式设备在内的各种主流平台。以下是在香橙派鲲鹏开发板（Ubuntu/openEuler 系统）上安装 GmSSL 的大致步骤。

第一步：安装基础依赖。

```
# 更新系统
sudo apt update && sudo apt upgrade -y
# 安装编译工具和依赖库
sudo apt install -y git cmake gcc g++ make
```

第二步：获取 GmSSL 源码。

```
# 克隆官方仓库（推荐 GmSSL 3.0+版本，对嵌入式优化更好）
git clone https://github.com/guanzhi/GmSSL.git
cd GmSSL
```

第三步：编译构建。GmSSL 3 采用 CMake 构建系统，执行以下命令：

```
# 创建构建目录
mkdir build
cd build
# 配置编译选项（prefix 指定安装路径，便于管理）
cmake .. -DCMAKE_INSTALL_PREFIX=/usr/local/gmssl
# 编译
make -j$(nproc)
# 运行测试验证（可选）
make test
# 安装
sudo make install
```

第四步：配置库路径。

```
# 克隆官方仓库（推荐 GmSSL 3.0+版本，对嵌入式优化更好）
git clone https://github.com/guanzhi/GmSSL.git
cd GmSSL
```

安装完成后，GmSSL 会在/usr/local/gmssl 目录下安装 gmssl 命令行工具、头文件（/usr/local/gmssl/include/gmssl）和库文件（libgmssl.a 静态库、libgmssl.so 动态库）。为了让系统能找到 GmSSL 库，需要配置环境变量：

```
# 添加库路径
echo '/usr/local/gmssl/lib' | sudo tee /etc/ld.so.conf.d/gmssl.conf
sudo ldconfig
# 添加头文件路径到环境变量
export C_INCLUDE_PATH=/usr/local/gmssl/include:$C_INCLUDE_PATH
export CPLUS_INCLUDE_PATH=/usr/local/gmssl/include:$CPLUS_INCLUDE_PATH
```

为避免与系统 OpenSSL 库冲突，建议将 GmSSL 安装在独立目录（如/usr/local/gmssl），并与 OpenSSL 保持相互独立。

第五步：验证安装。

```
# 测试 SM4 加密功能
gmssl sm4 -e -in test.txt -out test.enc -key 0123456789ABCDEFFEDCBA9876543210
# 验证库是否可用
ldconfig -p | grep gmssl
```

（2）密钥派生：PBKDF2 从用户密码生成加密密钥

在实际工程应用中，不应直接使用用户输入的密码作为 SM4 密钥，因为用户密码通常长度不足、熵值不够。需要使用密钥派生函数（KDF）将用户密码转换为符合 SM4 要求的 16 字节安全密钥。PBKDF2（Password-Based Key Derivation Function 2）是 RFC 2898 标准中定义的密钥派生算法，通过增加迭代次数来减慢哈希过程，显著提高密码破解的难度。

本模块采用 PBKDF2-HMAC-SHA256 算法，从用户密码派生 128 位（16 字节）SM4 密钥，同时生成随机盐值（salt）用于增强安全性。

```
#include <iostream>
#include <vector>
#include <cstring>
#include <random>
#include <gmssl/sm4.h>
// PBKDF2 参数
const int ITERATION_COUNT = 10000;          // 迭代次数（安全性与性能平衡）
const int KEY_LEN = 16;                     // SM4 密钥长度：16 字节
```

```
const int SALT_LEN = 16;                    // 盐值长度：16 字节
/**
 * 生成随机盐值（用于 PBKDF2）
 * @param salt 输出缓冲区，长度至少为 SALT_LEN
 */
void generateRandomSalt(uint8_t* salt) {
    // 使用硬件随机数生成器（鲲鹏开发板支持 RND 指令）
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, 255);
    for (int i = 0; i < SALT_LEN; i++) {
        salt[i] = static_cast<uint8_t>(dis(gen));
    }
}
/**
 * 使用 PBKDF2 从用户密码派生 SM4 密钥
 * @param password 用户输入的密码字符串
 * @param salt 随机盐值（16 字节）
 * @param key 输出密钥缓冲区（16 字节）
 * @return 成功返回 true，失败返回 false
 */
bool deriveKeyFromPassword(const std::string& password,
                           const uint8_t* salt,
                           uint8_t* key) {
    // 方法一：使用 OpenSSL 的 PKCS5_PBKDF2_HMAC（如果系统安装了 OpenSSL）
    // 方法二：使用 GmSSL 提供的 KDF 接口（推荐，避免外部依赖）

    // 此处使用 GmSSL 的 KDF 实现（需引入 gmssl/sm4.h）
    // PBKDF2-HMAC-SHA256: password, salt, iteration, key_len
    int ret = PKCS5_PBKDF2_HMAC(password.c_str(), password.length(),
                                salt, SALT_LEN,
                                ITERATION_COUNT,
                                EVP_sha256(),
                                KEY_LEN, key);
    return ret == 1;
}
/**
 * 从密码派生密钥的完整流程（加密端使用）
 * @param password 用户密码
 * @param key 输出密钥（16 字节）
 * @param salt 输出盐值（16 字节，需随密文一起传输给接收端）
 * @return 成功返回 true
 */
```

```
bool setupEncryptionKey(const std::string& password,
                        uint8_t* key,
                        uint8_t* salt) {
    // 1. 生成随机盐值
    generateRandomSalt(salt);
    // 2. 使用 PBKDF2 派生密钥
    return deriveKeyFromPassword(password, salt, key);
}
```

在实际系统集成时，发送端和接收端需使用相同的密码、相同的盐值、相同的迭代次数，才能派生出相同的密钥进行加解密。盐值作为明文随密文一起传输（通常附加在密文前面），接收端从中提取盐值后执行相同的 PBKDF2 派生过程。

（3）SM4-CBC 加密模块实现

本模块完成对敏感元数据的 SM4-CBC 加密。加密对象包括：视频元数据（人脸特征向量、水印信息、时间戳）、说话人特征向量等 JSON 格式数据。

```
#include <iostream>
#include <vector>
#include <cstring>
#include <gmssl/sm4.h>
// SM4 块大小：16 字节
#define SM4_BLOCK_SIZE 16
// PKCS#7 填充函数（将明文填充到块大小的整数倍）
void pkcs7_padding(const uint8_t* input, size_t input_len,
                   std::vector<uint8_t>& output) {
    size_t padding_len = SM4_BLOCK_SIZE - (input_len % SM4_BLOCK_SIZE);
    if (padding_len == 0) padding_len = SM4_BLOCK_SIZE;  // 恰好是块倍数时填充一个完整块

    output.assign(input, input + input_len);
    output.resize(input_len + padding_len, static_cast<uint8_t>(padding_len));
}
// PKCS#7 去填充（解密后去除填充字节）
bool pkcs7_unpadding(const uint8_t* input, size_t input_len,
                     std::vector<uint8_t>& output) {
    if (input_len == 0 || input_len % SM4_BLOCK_SIZE != 0) return false;
    uint8_t padding_len = input[input_len - 1];
    if (padding_len == 0 || padding_len > SM4_BLOCK_SIZE) return false;
    // 验证填充的正确性（可选：检查所有填充字节是否一致）
    for (size_t i = input_len - padding_len; i < input_len; i++) {
        if (input[i] != padding_len) return false;
```

```
        }
        output.assign(input, input + input_len - padding_len);
        return true;
    }
    /**
     * SM4-CBC 加密函数
     * @param plaintext 明文数据
     * @param plaintext_len 明文长度
     * @param key SM4 密钥（16 字节）
     * @param iv 初始化向量（16 字节，随机生成）
     * @param ciphertext 输出密文缓冲区
     * @return 成功返回密文长度，失败返回-1
     */
    int sm4_cbc_encrypt(const uint8_t* plaintext, size_t plaintext_len,
                        const uint8_t* key, uint8_t* iv,
                        std::vector<uint8_t>& ciphertext) {
        // 1. PKCS#7 填充
        std::vector<uint8_t> padded_data;
        pkcs7_padding(plaintext, plaintext_len, padded_data);
        // 2. 初始化 SM4 密钥结构
        SM4_KEY sm4_key;
        sm4_set_encrypt_key(&sm4_key, key);
        // 3. 复制 IV（CBC 模式会修改 IV，需要保护原始 IV）
        uint8_t iv_copy[SM4_BLOCK_SIZE];
        memcpy(iv_copy, iv, SM4_BLOCK_SIZE);
        // 4. CBC 模式加密（注意：此处的 sm4_cbc_encrypt 是 GmSSL 提供的接口）
        size_t ciphertext_len = padded_data.size();
        ciphertext.resize(ciphertext_len);
        sm4_cbc_encrypt(padded_data.data(), ciphertext.data(),
                        ciphertext_len, &sm4_key, iv_copy, SM4_ENCRYPT);
        return ciphertext_len;
    }
    /**
     * SM4-CBC 解密函数
     * @param ciphertext 密文数据
     * @param ciphertext_len 密文长度
     * @param key SM4 密钥（16 字节）
     * @param iv 初始化向量（16 字节，与加密时相同）
     * @param plaintext 输出明文缓冲区
     * @return 成功返回明文长度，失败返回-1
     */
    int sm4_cbc_decrypt(const uint8_t* ciphertext, size_t ciphertext_len,
                        const uint8_t* key, const uint8_t* iv,
                        std::vector<uint8_t>& plaintext) {
```

```
        // 1. 初始化 SM4 密钥结构
        SM4_KEY sm4_key;
        sm4_set_decrypt_key(&sm4_key, key);
        // 2. 复制 IV（解密会修改 IV 副本）
        uint8_t iv_copy[SM4_BLOCK_SIZE];
        memcpy(iv_copy, iv, SM4_BLOCK_SIZE);
        // 3. CBC 模式解密
        std::vector<uint8_t> decrypted_data(ciphertext_len);
        sm4_cbc_encrypt(ciphertext, decrypted_data.data(),
                                ciphertext_len, &sm4_key, iv_copy, SM4_DECRYPT);
        // 4. PKCS#7 去填充
        return pkcs7_unpadding(decrypted_data.data(), decrypted_data.size(), plaintext)
                ? plaintext.size() : -1;
    }
```

关于 IV（初始化向量）的重要说明：IV 不需要保密，但每次加密必须使用唯一且不可预测的随机 IV；IV 需要随密文一起传输给接收端（通常附加在密文前面或后面），接收端使用相同的 IV 进行解密；不要使用固定 IV 或全零 IV，否则会丧失 CBC 模式的安全优势；加密后的数据包格式为：[16 字节盐值]+[16 字节 IV]+[密文]。

（4）元数据加密与传输格式定义

加密的敏感元数据以 JSON 格式组织，加密后与音视频流并行传输。

元数据 JSON 结构示例：

```
{
    "frame_id": 12345,
    "timestamp": 1743321600000000,
    "faces": [
        {
            "rect": [320, 240, 100, 120],
            "feature": [0.12, 0.45, -0.33, ...]
        }
    ],
    "watermark": {
        "student_id": "2024001",
        "device_id": "DEV_001",
        "embed_time": 1743321600
    },
    "voice_feature": [0.21, -0.15, 0.08, ...]
}
```

加密传输格式可以参考如下设计：

```
/**
 * 构造加密数据包（用于网络传输）
 * 数据包格式：
 * +---------------+---------------+----------------------+
 *|  盐值 (16B)     |    IV (16B)      |    密文 (变长)             |
 * +---------------+---------------+----------------------+
 */
struct EncryptedPacket {
    uint8_t salt[16];          // PBKDF2 盐值
    uint8_t iv[16];             // SM4-CBC 初始化向量
    std::vector<uint8_t> ciphertext;   // 加密后的元数据
    // 序列化为字节流（用于网络发送）
    std::vector<uint8_t> serialize() const {
        std::vector<uint8_t> result;
        result.insert(result.end(), salt, salt + 16);
        result.insert(result.end(), iv, iv + 16);
        result.insert(result.end(), ciphertext.begin(), ciphertext.end());
        return result;
    }
    // 从字节流反序列化
    static EncryptedPacket deserialize(const uint8_t* data, size_t len) {
        EncryptedPacket packet;
        if (len >= 32) {
            memcpy(packet.salt, data, 16);
            memcpy(packet.iv, data + 16, 16);
            packet.ciphertext.assign(data + 32, data + len);
        }
        return packet;
    }
};
```

（5）完整加密流程示例

```
#include <iostream>
#include <vector>
#include <string>
#include <cstring>
#include <gmssl/sm4.h>
// 加密端完整流程
class SM4Encryptor {
private:
    uint8_t key[16];          // 派生后的 SM4 密钥
    uint8_t salt[16];        // PBKDF2 盐值
    std::string password; // 用户密码
```

```
    public:
        SM4Encryptor(const std::string& pwd) : password(pwd) {
            // 从密码派生密钥并生成盐值
            setupEncryptionKey(password, key, salt);
        }
        // 加密元数据（JSON 格式字符串）
        EncryptedPacket encryptMetadata(const std::string& metadata_json) {
            EncryptedPacket packet;
            // 1. 复制盐值到数据包
            memcpy(packet.salt, salt, 16);
            // 2. 生成随机 IV（每次加密都重新生成）
            generateRandomIV(packet.iv);
            // 3. 执行 SM4-CBC 加密
            std::vector<uint8_t> ciphertext;
            const uint8_t* plaintext = reinterpret_cast<const
uint8_t*>(metadata_json.c_str());
            size_t plaintext_len = metadata_json.length();

            sm4_cbc_encrypt(plaintext, plaintext_len, key, packet.iv, ciphertext);
            packet.ciphertext = std::move(ciphertext);
            return packet;
        }
        // 获取盐值（用于接收端同步）
        const uint8_t* getSalt() const { return salt; }
    };
    // 解密端完整流程
    class SM4Decryptor {
    private:
        uint8_t key[16];

    public:
        SM4Decryptor(const std::string& password, const uint8_t* salt) {
            // 使用相同的密码和盐值派生密钥
            deriveKeyFromPassword(password, salt, key);
        }

        // 解密数据包
        std::string decryptMetadata(const EncryptedPacket& packet) {
            std::vector<uint8_t> plaintext;
            int ret = sm4_cbc_decrypt(packet.ciphertext.data(),
                                      packet.ciphertext.size(),
                                      key, packet.iv, plaintext);
            if (ret < 0) {
                return "";  // 解密失败
```

```
            }
            return std::string(plaintext.begin(), plaintext.end());
        }
    };
```

（6）鲲鹏开发板硬件加速优化

鲲鹏 920 处理器内置了加解密加速引擎（KAE），支持 SM4 硬件加速。鲲鹏 KAE 引擎通过外部引擎的方式嵌入到 OpenSSL/GmSSL 软件中提供给上层应用使用，明文数据仅通过片内总线传输，不占用计算资源，安全性高。启用硬件加速后，SM4 加解密性能可提升 30 倍左右。

在 openEuler 系统上，通过安装以下 RPM 包启用鲲鹏加速引擎，并进行验证：

```
# 安装 KAE 软件包（需要以 root 权限执行）
rpm -ivh uacce*.rpm hisi*.rpm libwd-*.rpm libkae*.rpm
# 查看 KAE 引擎状态
cat /proc/crypto | grep -A 5 sm4
# 输出中应包含"driver: hisi_sec"表示硬件加速已启用
```

GmSSL 默认会自动检测并使用 KAE 硬件加速，无需额外代码修改。鲲鹏 KAE 支持的国密算法硬件加速包括 SM4-CBC、SM4-ECB、SM4-CTR、SM4-XTS 等模式。

（7）常见问题与解决思路

用户密码不应硬编码在代码中，建议通过配置文件或交互式输入获取。每次加密必须使用新的随机 IV，建议使用硬件随机数生成器（鲲鹏开发板支持 RND 指令）。PBKDF2 迭代次数建议≥10000 次，在安全性与性能之间取得平衡。建议用户设置足够强度的密码（长度≥8 位，包含大小写字母、数字、特殊字符）。CBC 模式存在填充预言攻击风险，实际生产环境建议使用 GCM 等认证加密模式。采用 CBC 模式可帮助学生更好理解分组密码工作原理；在工程应用中，建议升级到 SM4-GCM 模式以同时提供加密和完整性校验。其他问题见表 1.8。

表 1.8 加密常见问题与解决思路

问题现象	可能原因	解决思路
编译时找不到 gmssl 头文件	头文件路径未正确配置	检查/usr/local/gmssl/include/gmssl 是否存在，添加-I/usr/local/gmssl/include 编译选项
运行时找不到 libgmssl.so	动态库路径未配置	执行 sudo ldconfig 或设置 LD_LIBRARY_PATH=/usr/local/gmssl/lib

解密后数据乱码	IV 不一致或填充错误	确保加密端和解密端使用相同的 IV，检查 PKCS#7 填充逻辑
PBKDF2 派生密钥不一致	盐值或迭代次数不一致	确保盐值随密文一起传输，两端使用相同的迭代次数
硬件加速未生效	KAE 驱动未正确安装	检查/proc/crypto 中是否有 hisi_sec 驱动，重新安装 KAE 软件包
加密后数据膨胀	PKCS#7 填充增加了额外字节	填充最多增加 15 字节（块大小-1），属于正常现象

通过以上步骤，学生可以在鲲鹏开发板上独立完成 SM4 国密加密模块的开发，掌握从密钥派生、CBC 模式加密到数据包封装的完整流程，为后续网络传输模块提供安全的数据保障。

1.3.5 视频水印嵌入（溯源层，可选）

在音视频流中嵌入不可见的鲁棒水印，实现版权保护与泄密溯源。水印信息可以包含学生 ID、时间戳、设备编号等，嵌入后即使音视频经过压缩、裁剪、滤波等攻击，仍能提取出完整水印。本模块在编码压缩之前对原始视频帧进行水印嵌入，以确保水印能够抵抗后续的 H.264/AAC 有损压缩。

（1）DCT 变换域水印

将视频帧划分为 8×8 像素块，对每块进行离散余弦变换（DCT），将空间域信息转换到频率域。人眼对高频信息不敏感，对低频信息敏感；选择中频系数嵌入水印，可在不可见性与鲁棒性之间取得平衡：

$$C'=C+\alpha\times w$$

其中，C 为原始 DCT 系数，α 为嵌入强度，w 为水印比特。

纹理复杂区域（方差大）对噪声容忍度高，可增大 α；平滑区域（方差小）则减小 α。可使用块方差计算纹理复杂度：

$$\alpha=\text{base\_alpha}\times(1+k\times\text{variance}/\text{max\_variance})$$

其中，k 为调节因子。

每帧选择 N 个中频系数嵌入，每个系数嵌入 1 比特，实现每帧嵌入 32 比特（4 字节）水印信息。支持逐帧溯源，即使单帧受损，后续帧仍可提取。此外，中频系数抵抗压缩能力强（H.264/5 压缩主要量化高频系数）。采用纠错编码（如重复码）可以提高鲁棒性。还可以在嵌入前对水印进行扩频或 BCH 编码，增加容错能力。

（2）水印嵌入示例

首先安装依赖库。

```
# 安装 OpenCV（包含 DCT 函数）
sudo apt install libopencv-dev
# 安装线性代数库（用于矩阵计算）
sudo apt install libeigen3-dev
```

然后将水印字符串（如“SID:2024001|TS:1743321600|DEV:001”）转换为二进制比特流，每帧嵌入 32 比特。

```
#include <iostream>
#include <vector>
#include <string>
#include <bitset>
#include <cstdint>
// 将水印字符串转换为二进制向量（每帧 32 比特）
std::vector<uint8_t> encodeWatermark(const std::string& watermark_str) {
    std::vector<uint8_t> bits;
    for (char c : watermark_str) {
        std::bitset<8> b(c);
        for (int i = 0; i < 8; i++) {
            bits.push_back(b[i]); // 0 或 1
        }
    }
    // 不足 32 比特补零
    while (bits.size() % 32 != 0) bits.push_back(0);
    return bits;
}
```

DCT 变换与嵌入参考如下：

```
#include <opencv2/opencv.hpp>
#include <vector>
void embedWatermark(cv::Mat& frame, const std::vector<uint8_t>& watermark_bits, int
frame_idx) {
    // 确保帧是灰度图（Y 通道），如果是彩色则转为 YUV，处理 Y 通道
    cv::Mat yuv;
    cv::cvtColor(frame, yuv, cv::COLOR_BGR2YUV);
    cv::Mat y_channel(yuv.size(), CV_8UC1);
    cv::extractChannel(yuv, y_channel, 0); // Y 分量
    // 1. 获取当前帧对应的 32 比特水印
    int start_bit = (frame_idx * 32) % watermark_bits.size();
    std::vector<uint8_t> curr_bits(watermark_bits.begin() + start_bit,
                                   watermark_bits.begin() + start_bit + 32);
    // 2. 计算块数量（8x8 分块）
    int rows = y_channel.rows / 8;
```

```
int cols = y_channel.cols / 8;
if (rows == 0 || cols == 0) return;
// 3. 计算每块方差（纹理复杂度）
cv::Mat variance(rows, cols, CV_32F);
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        cv::Rect roi(j*8, i*8, 8, 8);
        cv::Mat block = y_channel(roi);
        cv::Scalar mean, stddev;
        cv::meanStdDev(block, mean, stddev);
        variance.at<float>(i,j) = stddev[0] * stddev[0]; // 方差
    }
}
// 4. 归一化方差到[0,1]，用于强度调节
double max_var = 0;
cv::minMaxIdx(variance, nullptr, &max_var);
if (max_var < 1e-6) max_var = 1.0;
// 5. 嵌入参数
const float base_alpha = 12.0;          // 基础嵌入强度
const float k = 0.5;                    // 强度调节因子
const int embed_count = 32;             // 嵌入比特数
// 选择中频系数位置：DCT 系数矩阵的(4,4)~(5,5)区域（索引从 0 开始）
const std::vector<cv::Point> mid_freq = {
    {4,4}, {4,5}, {5,4}, {5,5},
    {4,6}, {5,6}, {6,4}, {6,5}
};
// 每个块嵌入 1 比特，需要至少 32 个块
if (rows*cols < embed_count) return;
// 6. 遍历块，嵌入水印
for (int idx = 0; idx < embed_count; idx++) {
    int i = idx / cols;
    int j = idx % cols;
    cv::Rect roi(j*8, i*8, 8, 8);
    cv::Mat block = y_channel(roi);
    block.convertTo(block, CV_32F, 1.0/255.0); // 归一化到[0,1]
    // DCT 变换
    cv::Mat dct_block;
    cv::dct(block, dct_block);
    // 选择中频系数位置（取第一个中频点即可，但可以组合多个增加鲁棒性）
    cv::Point pos = mid_freq[idx % mid_freq.size()];
    float coeff = dct_block.at<float>(pos.y, pos.x);
    // 计算嵌入强度
    float var = variance.at<float>(i,j);
    float alpha = base_alpha * (1 + k * (var / max_var));
```

```
            // 嵌入比特：0 -> 减少系数，1 -> 增加系数
            uint8_t bit = curr_bits[idx];
            if (bit == 1) {
                coeff += alpha;
            } else {
                coeff -= alpha;
            }
            dct_block.at<float>(pos.y, pos.x) = coeff;
            // IDCT
            cv::Mat new_block;
            cv::dct(dct_block, new_block, cv::DCT_INVERSE);
            new_block.convertTo(new_block, CV_8U, 255.0);
            new_block.copyTo(y_channel(roi));
        }
        // 7. 合并回 YUV 并转回 BGR
        cv::Mat new_yuv;
        std::vector<cv::Mat> channels;
        cv::split(yuv, channels);
        channels[0] = y_channel;
        cv::merge(channels, new_yuv);
        cv::cvtColor(new_yuv, frame, cv::COLOR_YUV2BGR);
    }
    int main() {
        // 1. 初始化摄像头和编码器（略）
        // 2. 准备水印信息
        std::string watermark_str = "SID:2024001|TS:1743321600|DEV:001";
        auto watermark_bits = encodeWatermark(watermark_str);
        // 3. 循环处理每一帧
        cv::Mat frame;
        int frame_count = 0;
        while (true) {
            cap >> frame;
            if (frame.empty()) break;
            // 嵌入水印
            embedWatermark(frame, watermark_bits, frame_count);
            // 继续后续处理（编码、加密、传输）
            // encode_and_send(frame);
            frame_count++;
        }
        return 0;
    }
```

（3）抗攻击能力测试

在开发板上进行鲁棒性测试，验证水印在各种攻击下的存活率。学生可以把嵌入水印后的视频流保存为本地文件，然后对视频进行攻击：重编码（H.264，码率减半）、裁剪（保留 50%区域）、高斯滤波（3×3 核）、旋转（±10°）等。从攻击后的视频中提取水印，与原始水印比对，计算比特正确率。

```
#（Python 测试脚本，用于离线鲁棒性分析，不计入最终系统交付代码）
import cv2
import numpy as np
def test_robustness(video_path, attack_type):
    # 逐帧提取水印并统计正确率
    total_bits = 0
    correct_bits = 0
    cap = cv2.VideoCapture(video_path)
    frame_count = 0
    while True:
        ret, frame = cap.read()
        if not ret:
            break
        # 提取水印（需调用 C++提取函数，此处省略）
        # bits = extractWatermark(frame)
        # compare with original bits
        frame_count += 1
    cap.release()
    return correct_bits / total_bits if total_bits else 0
```

（4）常见问题与解决思路

使用 OpenCV 内置 DCT 已优化，但可进一步利用鲲鹏 NEON 指令集手动加速。可以将 8×8 块 DCT 操作放在独立线程中，利用多核 CPU。也可以考虑只对 I 帧或关键帧嵌入水印，减少计算量（但可能降低溯源粒度）。此外，对于 8×8 DCT，可以预先生成变换矩阵，用矩阵乘法代替函数调用。其他问题见表 1.9。

表 1.9 视频水印嵌入常见问题与解决思路

问题现象	可能原因	解决思路
水印不可见但无法提取	嵌入强度不足	增加 alpha 值，或选择更鲁棒的中频系数
视频质量下降明显	嵌入强度过大	减小 alpha，或仅对 Y 分量处理，并调整动态范围
提取时比特错误率高	攻击导致系数偏移	使用重复码（每个比特嵌入多个位置，投票决定）或 BCH 纠错码
运行速度慢	每帧都做 DCT/IDCT	只对关键帧嵌入，或使用硬件加速库（如 OpenCV with Intel IPP）

通过以上实现，学生可以在鲲鹏开发板上独立完成水印嵌入与提取，并测试其鲁棒性。这一模块为溯源层核心，实现了视频内容的版权保护和泄密溯源功能，是系统安全性的重要组成部分。

1.3.6 音频水印嵌入（溯源层，可选）

视频水印易受画面裁剪、旋转等几何攻击影响，而音频水印可提供独立于画面的辅助溯源信息。音视频双水印可交叉验证，增强系统的整体可靠性。

为了进一步增强溯源能力，可以考虑同时在音频流中嵌入水印信息，与视频水印形成双重校验。音频水印可以在声学上保持不可感知，同时抵抗常见的音频处理攻击（如压缩、重采样、滤波等）。

需要注意的是，人耳对音频噪声非常敏感，水印必须保持不可听性；音频编码（如 AAC）会引入压缩失真，水印需具备抗压缩能力；音频流具有时间连续性，水印需在时域和频域上均匀分布。

以下是基于离散小波变换（DWT）和量化索引调制（QIM）的音频水印实现思路，供学有余力的同学拓展。

（1）DWT 域量化索引调制

离散小波变换（DWT）可将音频信号分解为近似系数（低频）和细节系数（高频）。低频系数对压缩敏感，高频系数易被人耳感知，因此选择中频子带嵌入水印。量化索引调制（QIM）通过修改系数的量化值来嵌入比特，具有较好的鲁棒性和盲提取能力。

具体来说，选择某一中频子带的系数序列 x，量化步长 Δ。对于要嵌入的比特 b∈{0,1}，将系数 x 量化为最近的奇/偶倍数的量化值：

$$x'=round((x+d)/\Delta)*\Delta-d$$

其中，d=Δ/2*b。提取时，计算 b=round(x'/Δ)%2。

（2）PCM 数据缓存

由于水印嵌入可能需要一定长度的 PCM 数据（例如 2048 个样本），需要将 PCM 数据缓存起来，达到足够长度后再进行水印嵌入。

```
// PCM 缓冲区结构
struct PCMBuffer {
    std::vector<float> data;        // 存储 PCM 样本（浮点格式）
    int sample_rate;                // 44100
    int channels;                   // 2
```

```
    int target_size;                    // 目标缓冲区大小（如 2048 帧）
};
// 向缓冲区添加 PCM 数据
void add_to_buffer(PCMBuffer& buf, AVFrame* pcm_frame) {
    // pcm_frame->data[0] 包含交错或平面格式的 PCM 数据
    // 转换为浮点格式并追加到 buf.data
    int nb_samples = pcm_frame->nb_samples;
    int channels = pcm_frame->channels;
    // 将 PCM 转换为浮点格式（假设已经是 AV_SAMPLE_FMT_FLTP 格式）
    float* samples = (float*)pcm_frame->data[0];
    for (int i = 0; i < nb_samples; i++) {
        buf.data.push_back(samples[i]);
    }
}
// 检查缓冲区是否达到嵌入要求
bool is_buffer_ready(PCMBuffer& buf) {
    return buf.data.size() >= buf.target_size;
}
```

在 PCM 上操作水印嵌入：

```
// 在 PCM 缓冲区上嵌入水印
void embed_watermark_in_pcm(PCMBuffer& buf, const std::vector<uint8_t>& watermark_bits, int frame_idx) {
    // 1. 取出一段 PCM 数据进行小波变换（示例：DWT）
    std::vector<float> segment(buf.data.begin(), buf.data.begin() + buf.target_size);
    // 2. 对 segment 进行 DWT 变换，获得系数序列
    // 这里调用之前实现的 DWT 水印嵌入函数（详见水印嵌入示例部分）
    embed_audio_watermark(segment, watermark_bits, frame_idx);
    // 3. 将嵌入水印后的数据写回缓冲区
    for (int i = 0; i < buf.target_size; i++) {
        buf.data[i] = segment[i];
    }
    // 4. 移除已处理的部分，保留剩余数据
    buf.data.erase(buf.data.begin(), buf.data.begin() + buf.target_size);
}
```

最后，将嵌入水印后的 PCM 数据编码为 AAC 格式，再按原流程发送。

```
// 将 PCM 数据编码为 AAC 包
AVPacket* encode_pcm_to_aac(AVCodecContext *enc_ctx, const std::vector<float>& pcm_data, int sample_rate, int channels) {
    // 1. 准备 AVFrame，填充 PCM 数据
    AVFrame *frame = av_frame_alloc();
    frame->format = AV_SAMPLE_FMT_FLTP;
    frame->sample_rate = sample_rate;
```

```
        frame->channels = channels;
        frame->nb_samples = pcm_data.size() / channels;
        av_frame_get_buffer(frame, 0);
        // 将浮点 PCM 数据拷贝到 frame 中（假设平面格式）
        float* dst = (float*)frame->data[0];
        memcpy(dst, pcm_data.data(), pcm_data.size() * sizeof(float));
        // 2. 发送帧到 AAC 编码器
        int ret = avcodec_send_frame(enc_ctx, frame);
        if (ret < 0) {
            av_frame_free(&frame);
            return NULL;
        }
        // 3. 接收编码后的 AAC 包
        AVPacket *pkt = av_packet_alloc();
        ret = avcodec_receive_packet(enc_ctx, pkt);
        if (ret < 0) {
            av_packet_free(&pkt);
            av_frame_free(&frame);
            return NULL;
        }
        av_frame_free(&frame);
        return pkt;  // 返回 AAC 数据包，可继续原流程发送
    }
```

（4）水印嵌入示例

首先可以将音频流（PCM 格式，44.1kHz，16bit）分割成固定长度的帧（例如 2048 个样本），每帧嵌入一个比特。为保证水印分布均匀，对每帧进行三级 DWT 分解。

```
    #include <vector>
    #include <cmath>
    #include <fftw3.h>  // 可选用于小波变换，或使用开源库如 libwavelet
    // 伪代码：分帧
    std::vector<std::vector<float>> split_audio(const std::vector<float>& audio, int frame_size) {
        std::vector<std::vector<float>> frames;
        for (size_t i = 0; i + frame_size <= audio.size(); i += frame_size) {
            frames.push_back(std::vector<float>(audio.begin() + i, audio.begin() + i + frame_size));
        }
        return frames;
    }
```

对每帧进行三级 DWT 分解（例如使用 Daubechies-4 小波），获得近似系数（cA3）和细节系数（cD3, cD2, cD1）。选择 cD3 子带（中频）嵌入水印，因为它兼顾了鲁棒性与不可听性。

```
#include <vector>
#include <cmath>
#include <algorithm>
#include <cassert>
// Daubechies-4 小波滤波器系数 (低通)
const std::vector<double> db4_low = {
    0.4829629131445341, 0.8365163037378079,
    0.2241438680420134, -0.1294095225512604
};
// 高通滤波器系数（由低通系数通过翻转和交替符号得到）
std::vector<double> db4_high(const std::vector<double>& low) {
    std::vector<double> high(low.size());
    for (size_t i = 0; i < low.size(); ++i) {
        high[i] = pow(-1.0, i) * low[low.size() - 1 - i];
    }
    return high;
}
// 单层 DWT 分解
void dwt_step(const std::vector<float>& signal, std::vector<float>& approx, std::vector<float>& detail) {
    const auto& low = db4_low;
    auto high = db4_high(low);
    size_t n = signal.size();
    size_t len = n / 2;
    approx.resize(len);
    detail.resize(len);
    for (size_t i = 0; i < len; ++i) {
        double sum_low = 0.0, sum_high = 0.0;
        for (size_t k = 0; k < low.size(); ++k) {
            size_t idx = (2*i + k) % n;   // 周期延拓
            sum_low += low[k] * signal[idx];
            sum_high += high[k] * signal[idx];
        }
        approx[i] = static_cast<float>(sum_low);
        detail[i] = static_cast<float>(sum_high);
    }
}
// 多层 DWT，返回系数列表（类似于 pywt.wavedec）
std::vector<std::vector<float>> wavedec(const std::vector<float>& signal, int level) {
    std::vector<std::vector<float>> coeffs;
```

```
        std::vector<float> current = signal;
        for (int l = 0; l < level; ++l) {
            std::vector<float> approx, detail;
            dwt_step(current, approx, detail);
            coeffs.push_back(approx);    // 注意：此处我们存储细节，最后再加近似
            coeffs.push_back(detail);
            // 将 approx 作为下一层的输入
            current = std::move(approx);
        }
        // 现在 coeffs 包含 [cD1, cA1, cD2, cA2, ...] 顺序混乱，需要调整
        // 更好的方式是返回 [cA_level, cD_level, cD_{level-1}, ..., cD1]
        // 重新组织
        std::vector<std::vector<float>> result;
        result.push_back(current);  // 最终近似系数
        // 从最深层细节开始依次添加
        for (int l = level-1; l >= 0; --l) {
            // 在 coeffs 中，第 l 层的细节位于索引 2*l+1
            result.push_back(coeffs[2*l+1]);
        }
        return result;
    }
    // 对信号进行 level 层离散小波变换，返回扁平化的系数向量
    // 排列顺序: [cA_level, cD_level, cD_{level-1}, ..., cD1]
    std::vector<float> dwt(const std::vector<float>& signal, int level) {
        // 确保信号长度为 2 的幂次，如果不是，用零填充
        size_t n = signal.size();
        size_t new_n = 1;
        while (new_n < n) new_n <<= 1;
        std::vector<float> padded_signal = signal;
        if (new_n > n) {
            padded_signal.resize(new_n, 0.0f);
        }
        auto coeffs = wavedec(padded_signal, level);
        // 将嵌套向量扁平化
        std::vector<float> flat;
        for (const auto& vec : coeffs) {
            flat.insert(flat.end(), vec.begin(), vec.end());
        }
        return flat;
    }
```

在选定的子带系数序列中，每隔一定间隔选择一个系数，使用 QIM 嵌入比特。

```
    float embed_bit(float coeff, int bit, float delta) {
```

```
        float q = coeff / delta;
        int q_int = round(q);
        if (bit == 1) {
            if (q_int % 2 == 0) q_int += 1;
        } else {
            if (q_int % 2 == 1) q_int += 1;
        }
        return q_int * delta;
    }
```

将修改后的系数与未修改的系数重组，通过逆小波变换重构音频帧，再拼接成完整音频。

（5）与视频水印的协同

音频水印与视频水印应嵌入相同的水印内容（学生 ID、时间戳、设备编号等），便于交叉验证。在发送端，为音视频流打上统一的时间戳，接收端按时间戳将音视频帧对齐，分别提取水印后比对。若视频水印因几何攻击无法提取，而音频水印仍可提取，则仍可完成溯源；若两者均提取成功且信息一致，则可信度更高。

（6）抗攻击能力与测试

音频水印应能抵抗以下常见攻击：

- MP3/AAC 压缩：选择中频子带，抵抗有损压缩的量化误差。
- 重采样：设计水印嵌入在变换域，对重采样有一定鲁棒性。
- 噪声添加：通过调节量化步长 Δ，平衡鲁棒性与音质。
- 时间裁剪：若水印分散嵌入所有帧，部分裁剪不影响剩余帧的水印提取。

测试方法：

- 嵌入水印后，使用 FFmpeg 将音频转码为 AAC 128kbps，再解码回 PCM，提取水印，计算比特正确率。
- 对音频添加高斯白噪声（SNR=30dB），再提取。
- 截取音频中间部分，提取水印。

（7）优化思路

音频水印嵌入可在编码前进行，若采用 DWT，计算量适中；可选择只对关键帧或每隔几帧嵌入，降低计算开销。音频流需与视频流保持时间同步，建议在封装时使用相同的 PTS 时间戳。Δ 过大影响音质，过小鲁棒性差；建议通过主观

听觉测试和客观信噪比（SNR）确定。可使用 Python 快速原型验证算法逻辑，验证通过后使用 C++调用 FFmpeg 的 libavcodec 库进行最终集成，利用 FFmpeg 的 C 语言接口实现音频解码与水印嵌入的工程化部署。

通过增加音频水印模块，学生可以进一步理解多模态溯源的设计思路，并掌握在变换域中嵌入鲁棒信息的通用方法。音频水印与视频水印相结合，使系统的溯源能力更加全面可靠。

1.3.7 人脸检测（智能层，可选）

香橙派鲲鹏开发板搭载鲲鹏处理器，内置神经网络加速单元（NPU），适合边缘端部署轻量级 AI 模型。

（1）验证 NPU 可用性

```
#查看 NPU 状态
npu-smi info
#输出应显示 NPU 芯片信息和健康状态
```

（2）模型获取与量化

首先下载 YOLOv5s 模型。YOLOv5s 是 YOLOv5 系列中的轻量级版本，适合嵌入式部署。从官方仓库下载预训练模型（PyTorch 格式）

```
git clone https://github.com/ultralytics/yolov5.git
cd yolov5
pip install -r requirements.txt
wget https://github.com/ultralytics/yolov5/releases/download/v7.0/yolov5s.pt
```

为了在 NPU 上高效运行，需先将 PyTorch 模型转换为 ONNX：

```
#export.py
import torch
model = torch.hub.load('ultralytics/yolov5', 'yolov5s', pretrained=True)
model.eval()
dummy_input = torch.randn(1, 3, 640, 640)
torch.onnx.export(model, dummy_input, "yolov5s.onnx",
                  input_names=['images'], output_names=['output'],
                  opset_version=11)
```

NPU 对 int8 量化模型推理速度最快。使用 CANN 提供的 ATC 工具进行量化，转换后得到 yolov5s_int8.om 文件，可直接在 NPU 上加载运行。

```
#生成量化校准集（从视频帧中采集 100 张图像）
#使用 ATC 转换
atc --model=yolov5s.onnx --framework=5 --output=yolov5s_int8 \
```

```
    --input_format=NCHW --input_shape="images:1,3,640,640" \
    --soc_version=Ascend310B4 --precision_mode=allow_mix_precision
```

量化后模型精度会略有下降，建议在测试集上验证：

```
#使用 CANN 提供的精度对比工具
omg --model=yolov5s.om --precision
```

以上模型导出和量化操作可在任意 PC（x86 或 ARM 架构均可）上完成，将生成的 yolov5s_int8.om 模型文件拷贝至鲲鹏开发板即可加载运行。鲲鹏 NPU 推理阶段完全采用 C++实现，无需在开发板上安装 PyTorch 环境。

（3）人脸检测示例

建议采用 C++实现，便于在鲲鹏开发板上获得更好的性能。

```
#include <opencv2/opencv.hpp>
#include <opencv2/dnn.hpp>
#include <iostream>
#include <vector>
// 引入 CANN 头文件
#include <acl/acl.h>
class FaceDetector {
private:
    cv::VideoCapture cap;
    cv::dnn::Net net;                  // 用于 CPU 回退
    aclrtContext context;             // NPU 上下文
    uint32_t modelId;                   // 模型 ID
    int frame_width, frame_height;
public:
    FaceDetector(int camera_id = 0, int width = 1280, int height = 720) {
        // 打开摄像头
        cap.open(camera_id);
        if (!cap.isOpened()) {
            std::cerr << "Error: Cannot open camera" << std::endl;
            exit(1);
        }
        cap.set(cv::CAP_PROP_FRAME_WIDTH, width);
        cap.set(cv::CAP_PROP_FRAME_HEIGHT, height);
        cap.set(cv::CAP_PROP_FPS, 30);
        frame_width = cap.get(cv::CAP_PROP_FRAME_WIDTH);
        frame_height = cap.get(cv::CAP_PROP_FRAME_HEIGHT);
        // 初始化 NPU
        aclInit(nullptr);
        aclrtSetDevice(0);
        aclrtCreateContext(&context, 0);
```

```
        // 加载量化模型
        loadModel("yolov5s_int8.om");
    }
    ~FaceDetector() {
        cap.release();
        aclrtDestroyContext(context);
        aclrtResetDevice(0);
        aclFinalize();
    }
    void loadModel(const std::string& model_path) {
        // 使用 CANN 加载离线模型
        aclmdlDesc *modelDesc = aclmdlCreateDesc();
        aclmdlLoadFromFile(model_path.c_str(), &modelId);
        std::cout << "Model loaded successfully, modelId=" << modelId << std::endl;
    }
    // 人脸检测主函数
    void detect() {
        cv::Mat frame;
        int frame_count = 0;
        std::vector<cv::Rect> faces;
        cv::Ptr<cv::Tracker> tracker;
        bool tracking = false;
        while (true) {
            cap >> frame;
            if (frame.empty()) break;
            frame_count++;
            // 每 5 帧进行一次 YOLO 检测，其余帧使用 KCF 跟踪
            if (frame_count % 5 == 0) {
                // 调用 NPU 推理
                faces = detectWithNPU(frame);
                if (!faces.empty()) {
                    // 初始化跟踪器
                    tracker = cv::TrackerKCF::create();
                    tracker->init(frame, faces[0]);
                    tracking = true;
                } else {
                    tracking = false;
                }
            } else if (tracking) {
                // 使用 KCF 更新跟踪位置
                cv::Rect2d rect;
                if (tracker->update(frame, rect)) {
                    faces.clear();
                    faces.push_back(rect);
```

```
                    } else {
                        tracking = false;
                    }
                }
                // 绘制检测结果
                for (const auto& face : faces) {
                    cv::rectangle(frame, face, cv::Scalar(0, 255, 0), 2);
                    // 生成时间戳和特征向量（此处简化为占位）
                    std::string timestamp = std::to_string(cv::getTickCount());
                    cv::putText(frame, "Face", cv::Point(face.x, face.y-5),
                                cv::FONT_HERSHEY_SIMPLEX, 0.5,
cv::Scalar(0,255,0), 1);
                }
                // 显示视频（可选）
                cv::imshow("Face Detection", frame);
                if (cv::waitKey(1) == 'q') break;
            }
        }
        // NPU 推理实现
        std::vector<cv::Rect> detectWithNPU(cv::Mat& frame) {
            // 预处理：缩放至 640x640，归一化
            cv::Mat input_blob;
            cv::resize(frame, input_blob, cv::Size(640, 640));
            input_blob.convertTo(input_blob, CV_32FC3, 1.0/255.0);
            // 转换 NHWC→NCHW（NPU 要求）
            cv::Mat nchw_blob;
            cv::dnn::blobFromImage(input_blob, nchw_blob);
            // 准备输入内存（NPU 需要连续内存）
            size_t input_size = 1 * 3 * 640 * 640 * sizeof(float);
            void *input_mem = nullptr;
            aclrtMalloc(&input_mem, input_size, ACL_MEM_MALLOC_HUGE_FIRST);
            aclrtMemcpy(input_mem, input_size, nchw_blob.data, input_size,
ACL_MEMCPY_HOST_TO_DEVICE);
            // 执行推理
            aclmdlDataset *input_dataset = aclmdlCreateDataset();
            aclmdlDataset *output_dataset = aclmdlCreateDataset();
            aclDataBuffer *input_buffer = aclCreateDataBuffer(input_mem, input_size);
            aclmdlAddDatasetBuffer(input_dataset, input_buffer);
            // 获取输出大小
            size_t output_size = 0;
            aclmdlGetOutputSizeByIndex(modelId, 0, &output_size);
            void *output_mem = nullptr;
            aclrtMalloc(&output_mem, output_size,
ACL_MEM_MALLOC_HUGE_FIRST);
```

```
        aclDataBuffer *output_buffer = aclCreateDataBuffer(output_mem, output_size);
        aclmdlAddDatasetBuffer(output_dataset, output_buffer);
        // 执行推理
        aclmdlExecute(modelId, input_dataset, output_dataset);
        // 解析输出（YOLOv5 输出格式：[batch, 25200, 85]）
        float *output_data = (float*)output_mem;
        std::vector<cv::Rect> faces;
        // 后处理：解析检测框、置信度筛选、NMS
        float confidence_threshold = 0.5;
        float nms_threshold = 0.4;
        std::vector<cv::Rect> boxes;
        std::vector<float> confidences;
        for (int i = 0; i < 25200; i++) {
            float confidence = output_data[i*85 + 4];
            if (confidence > confidence_threshold) {
                // 解码坐标（相对于 640x640）
                float x_center = output_data[i*85 + 0];
                float y_center = output_data[i*85 + 1];
                float w = output_data[i*85 + 2];
                float h = output_data[i*85 + 3];
                int x = int((x_center - w/2) * frame_width / 640);
                int y = int((y_center - h/2) * frame_height / 640);
                int width = int(w * frame_width / 640);
                int height = int(h * frame_height / 640);
                boxes.push_back(cv::Rect(x, y, width, height));
                confidences.push_back(confidence);
            }
        }
        // 非极大值抑制
        std::vector<int> indices;
        cv::dnn::NMSBoxes(boxes, confidences, confidence_threshold, nms_threshold,
indices);
        for (int idx : indices) {
            faces.push_back(boxes[idx]);
        }
        // 释放内存
        aclDestroyDataBuffer(input_buffer);
        aclDestroyDataBuffer(output_buffer);
        aclmdlDestroyDataset(input_dataset);
        aclmdlDestroyDataset(output_dataset);
        aclrtFree(input_mem);
        aclrtFree(output_mem);
        return faces;
    }
```

```
};

int main() {
    FaceDetector detector(0);
    detector.detect();
    return 0;
}
```

（4）调试与优化建议

通过 npu-smi set -t 1000 -i 0 将 NPU 频率调至最高，提升推理速度，将采集、推理、显示分别放在不同线程，利用开发板多核能力，输入输出内存预先分配并复用，避免频繁申请释放，其他问题见表 1.10。

表 1.10 人脸检测常见问题与解决思路

问题现象	可能原因	解决思路
NPU 推理失败	模型版本不匹配	确认 ATC 转换时 soc_version 参数正确
检测框偏移	坐标映射错误	检查缩放比例计算，确保从 640 映射回原始分辨率
跟踪效果差	KCF 参数不当	调整跟踪区域大小，或改用 MedianFlow 跟踪器

（5）输出数据格式

检测完成后，生成以下视频元数据，如表 1.11 所示，供后续模块使用。

表 1.11 视频元数据格式

字段	类型	说明
face_rect	[x, y, w, h]	人脸位置坐标（整数）
face_feature	float[128]	人脸特征向量（后续认证使用）
timestamp	uint64	采集时间戳（微秒级）
frame_id	uint32	帧序号

元数据以 JSON 格式附加在视频流之前，可以通过 SM4 加密后与音视频流并行传输。

```
{
  "frame_id": 123,
  "timestamp": 1743321600000000,
  "faces": [
    {
      "rect": [320, 240, 100, 120],
      "feature": [0.12, 0.45, ...]
    }
  ]
}
```

通过以上步骤，学生可以在香橙派鲲鹏开发板上完整实现人脸检测模块，既掌握了 NPU 推理部署的全流程，也学会了“检测+跟踪”这一实用的嵌入式优化策略，为后续水印嵌入、加密传输、多模态认证等高级任务打下坚实基础。

1.3.8 说话人特征提取（智能层，可选）

对采集的音频流进行说话人识别特征提取，生成说话人特征向量（如 MFCC + GMM 超向量或深度嵌入向量），并与视频元数据一并加密传输，供接收端进行多模态融合认证。

（1）环境准备

说话人识别（Speaker Recognition）的目标是根据语音信号识别说话人身份。常用的声学特征为梅尔频率倒谱系数（MFCC），它模拟了人耳对不同频率的感知特性，具有良好的鲁棒性。在提取 MFCC 后，可采用高斯混合模型（GMM）对说话人进行建模，并将模型参数（如均值超向量）作为说话人特征向量。

发送端需对当前采集的音频进行说话人特征提取，生成固定维度的特征向量（例如 512 维），供接收端与预先注册的说话人模型进行比对。为了简化部署并适应鲲鹏开发板资源，可以采用轻量级 GMM-UBM（通用背景模型）方法：事先在服务器上训练一个 UBM（包含 256 个高斯分量），发送端仅提取当前音频的 MFCC，然后计算相对于 UBM 的均值超向量（即每个高斯分量的均值偏移量拼接而成的向量），作为说话人特征。该特征维度为 256×39=9984 维（若采用 39 维 MFCC），或使用降维技术压缩至 512 维。

在香橙派鲲鹏开发板上安装音频处理与机器学习库：

```
#安装音频处理库
sudo apt install libsndfile1-dev libfftw3-dev
#安装 Python 科学计算环境（可选，用于原型验证，生产使用 C++）
pip3 install numpy scipy scikit-learn python_speech_features
```

若使用 C++实现，推荐以下库：

- Essentia：开源音频分析库，支持 MFCC 提取。
- OpenCV：提供 DNN 模块，可加载预训练的说话人识别模型。
- Kaldi（轻量级子集）：说话人识别领域标准工具，但编译复杂，建议交叉编译或使用预编译库。

为简化教学，提供 C++实现 MFCC 提取+预训练 GMM 均值超向量计算的完整代码，可以依赖开源库 librosa 的 C++移植版本或自行实现。

（2）音频采集与预处理

发送端已经通过 FFmpeg 或 ALSA 采集了 PCM 音频（见 4.3.节步骤 5 中的音频流）。说话人特征提取需要一段固定长度（如 2 秒）的音频片段。为了实时性，可采用滑动窗口方式，每隔一定时间（如 1 秒）提取一次特征。

音频预处理步骤大致如下：分帧，将音频划分为 20-30ms 的帧，帧移 10ms；加窗，对每帧应用汉明窗，减少频谱泄漏；预加重，提升高频分量，补偿声门激励和唇辐射，预加重系数通常为 0.97。

```
#include <vector>
#include <cmath>
#include <cstring>
// 预加重滤波器
std::vector<double> preEmphasis(const std::vector<double>& signal, double coeff = 0.97)
{
    std::vector<double> out(signal.size());
    out[0] = signal[0];
    for (size_t i = 1; i < signal.size(); ++i) {
        out[i] = signal[i] - coeff * signal[i-1];
    }
    return out;
}
// 汉明窗
std::vector<double> hammingWindow(int length) {
    std::vector<double> window(length);
    for (int i = 0; i < length; ++i) {
        window[i] = 0.54 - 0.46 * cos(2 * M_PI * i / (length - 1));
    }
    return window;
}
// 分帧（帧长 frame_len，帧移 frame_step，单位：采样点数）
std::vector<std::vector<double>> framing(const std::vector<double>& signal,
                                         int frame_len, int frame_step) {
    std::vector<std::vector<double>> frames;
    for (size_t start = 0; start + frame_len <= signal.size(); start += frame_step) {
        std::vector<double> frame(signal.begin() + start, signal.begin() + start +
frame_len);
        frames.push_back(frame);
    }
    return frames;
```

```
}
```

（3）MFCC 特征提取示例

MFCC 提取流程：FFT→梅尔滤波器组→对数能量→DCT。可以基于 FFTW3 库实现，需要预先安装：sudo apt install libfftw3-dev。

```
#include <fftw3.h>
#include <vector>
#include <cmath>
class MfccExtractor {
private:
    int n_mfcc;              // MFCC 系数个数（通常 13 或 20）
    int n_mels;              // 梅尔滤波器个数（通常 40）
    int fft_size;            // FFT 点数（通常 512 或 1024）
    int sample_rate;
    std::vector<double> mel_filterbank;
    std::vector<double> dct_matrix;
public:
    MfccExtractor(int sr, int n_mfcc = 13, int n_mels = 40, int fft_size = 512)
        : sample_rate(sr), n_mfcc(n_mfcc), n_mels(n_mels), fft_size(fft_size) {
        buildMelFilterbank();
        buildDctMatrix();
    }
    // 构建梅尔滤波器组
    void buildMelFilterbank() {
        double mel_min = 2595 * log10(1 + 0 / 700.0);
        double mel_max = 2595 * log10(1 + sample_rate/2 / 700.0);
        double mel_step = (mel_max - mel_min) / (n_mels + 1);

        std::vector<double> mel_points(n_mels + 2);
        for (int i = 0; i < n_mels + 2; ++i) {
            double mel = mel_min + i * mel_step;
            mel_points[i] = 700 * (pow(10, mel/2595) - 1);
        }
        // 计算每个 FFT 频点对应的频率索引
        std::vector<int> fft_freqs(fft_size/2 + 1);
        for (int i = 0; i <= fft_size/2; ++i) {
            fft_freqs[i] = i * sample_rate / fft_size;
        }
        // 构建滤波器权重矩阵 (n_mels x (fft_size/2+1))
        mel_filterbank.resize(n_mels * (fft_size/2 + 1), 0.0);
        for (int i = 0; i < n_mels; ++i) {
            double left = mel_points[i];
            double center = mel_points[i+1];
```

```
                double right = mel_points[i+2];
                for (int j = 0; j <= fft_size/2; ++j) {
                    double freq = fft_freqs[j];
                    if (freq >= left && freq <= center) {
                        mel_filterbank[i * (fft_size/2+1) + j] = (freq - left) / (center - left);
                    } else if (freq >= center && freq <= right) {
                        mel_filterbank[i * (fft_size/2+1) + j] = (right - freq) / (right - center);
                    }
                }
            }
        }
        // 构建 DCT 矩阵（用于将 log-mel 谱转换为 MFCC）
        void buildDctMatrix() {
            dct_matrix.resize(n_mfcc * n_mels);
            for (int i = 0; i < n_mfcc; ++i) {
                for (int j = 0; j < n_mels; ++j) {
                    dct_matrix[i * n_mels + j] = sqrt(2.0 / n_mels) * cos(M_PI * i * (j + 0.5) / n_mels);
                }
            }
        }
        // 对一帧（已经预加重、加窗）计算 MFCC
        std::vector<double> computeFrameMfcc(const std::vector<double>& frame) {
            // 1. 执行 FFT
            fftw_complex *in, *out;
            fftw_plan p;
            in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * fft_size);
            out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * fft_size);
            // 将 frame 数据拷贝到 in（实数部分）
            for (int i = 0; i < fft_size; ++i) {
                in[i][0] = (i < frame.size()) ? frame[i] : 0.0;
                in[i][1] = 0.0;
            }
            p = fftw_plan_dft_1d(fft_size, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
            fftw_execute(p);
            // 2. 计算功率谱
            std::vector<double> power_spectrum(fft_size/2 + 1);
            for (int i = 0; i <= fft_size/2; ++i) {
                power_spectrum[i] = out[i][0]*out[i][0] + out[i][1]*out[i][1];
            }
            // 3. 应用梅尔滤波器组，得到梅尔能量
            std::vector<double> mel_energy(n_mels, 0.0);
            for (int i = 0; i < n_mels; ++i) {
```

```
                    for (int j = 0; j <= fft_size/2; ++j) {
                        mel_energy[i] += mel_filterbank[i * (fft_size/2+1) + j] * power_spectrum[j];
                    }
                    // 避免 log(0)
                    if (mel_energy[i] < 1e-10) mel_energy[i] = 1e-10;
                    mel_energy[i] = log(mel_energy[i]);
                }
                // 4. DCT 得到 MFCC
                std::vector<double> mfcc(n_mfcc, 0.0);
                for (int i = 0; i < n_mfcc; ++i) {
                    for (int j = 0; j < n_mels; ++j) {
                        mfcc[i] += dct_matrix[i * n_mels + j] * mel_energy[j];
                    }
                }
                // 清理 FFTW 资源
                fftw_destroy_plan(p);
                fftw_free(in);
                fftw_free(out);
                return mfcc;
            }
            // 对整段音频提取 MFCC 序列
            std::vector<std::vector<double>> extractMfccSequence(const std::vector<double>& audio, int frame_len, int frame_step) {
                auto frames = framing(audio, frame_len, frame_step);
                std::vector<std::vector<double>> mfcc_sequence;
                for (const auto& frame : frames) {
                    auto mfcc = computeFrameMfcc(frame);
                    mfcc_sequence.push_back(mfcc);
                }
                return mfcc_sequence;
            }
        };
```

（4）说话人特征向量生成（GMM 均值超向量）

预先训练一个 UBM（通用背景模型），包含 256 个高斯分量，每个高斯分量在 39 维 MFCC 空间上有均值向量 μ_k 和协方差矩阵（通常为对角阵）。对于待识别的音频，提取其 MFCC 序列 X=x_1,...,x_T}，计算每个 MFCC 向量属于第 k 个高斯分量的后验概率：

$$\gamma_k(x_t) = \frac{w_k \mathcal{N}(x_t; \mu_k, \Sigma_k)}{\sum_{j=1}^{K} w_j \mathcal{N}(x_t; \mu_j, \Sigma_j)}$$

然后计算一阶统计量：

$$N_k = \sum_{t=1}^{T} \gamma_k(x_t), \quad F_k = \sum_{t=1}^{T} \gamma_k(x_t) x_t$$

均值超向量（也称 GMM 超向量或 i-vector 的基础）由 F_k/N_k 拼接而成，维度为 K×D。为了降低传输负载，可以进一步使用 PCA 降维至 512 维。

由于训练 UBM 和降维矩阵需要大量数据，建议在 PC 上离线完成，将 UBM 参数（均值、方差、权重）和 PCA 投影矩阵导出为二进制文件，部署到鲲鹏开发板上。开发板运行时只需加载这些参数并计算当前音频的均值超向量。

```
#C++实现均值超向量计算
#include <Eigen/Dense>
#include <vector>
#include <cmath>
// 假设已从文件加载 UBM 参数
struct UBM {
    int n_gaussians;                    // K = 256
    int feature_dim;                    // D = 39
    std::vector<double> weights;       // 权重, size K
    std::vector<Eigen::VectorXd> means;     // 均值, size K
    std::vector<Eigen::VectorXd> inv_vars; // 逆方差(对角), size K
};
// 计算单个高斯分量的对数概率密度
double gaussianLogLikelihood(const Eigen::VectorXd& x, const Eigen::VectorXd& mean,
                             const Eigen::VectorXd& inv_var) {
    double loglik = 0.0;
    for (int i = 0; i < x.size(); ++i) {
        double diff = x[i] - mean[i];
        loglik += diff * diff * inv_var[i];
    }
    return -0.5 * loglik;
}
// 计算均值超向量
Eigen::VectorXd computeMeanSupervector(const std::vector<Eigen::VectorXd>& mfcc_sequence,
                                       const UBM& ubm) {
    int K = ubm.n_gaussians;
    int D = ubm.feature_dim;
    int T = mfcc_sequence.size();
    // 统计量
    std::vector<double> N(K, 0.0);
    std::vector<Eigen::VectorXd> F(K, Eigen::VectorXd::Zero(D));
```

```
        for (const auto& x : mfcc_sequence) {
            // 计算所有分量的似然
            std::vector<double> log_lik(K);
            double max_log_lik = -1e9;
            for (int k = 0; k < K; ++k) {
                log_lik[k] = log(ubm.weights[k]) + gaussianLogLikelihood(x, ubm.means[k], ubm.inv_vars[k]);
                if (log_lik[k] > max_log_lik) max_log_lik = log_lik[k];
            }
            // 软最大化，计算后验概率
            double sum_posterior = 0.0;
            std::vector<double> posterior(K);
            for (int k = 0; k < K; ++k) {
                posterior[k] = exp(log_lik[k] - max_log_lik);
                sum_posterior += posterior[k];
            }
            for (int k = 0; k < K; ++k) {
                double gamma = posterior[k] / sum_posterior;
                N[k] += gamma;
                F[k] += gamma * x;
            }
        }
        // 拼接均值超向量
        Eigen::VectorXd supervector(K * D);
        for (int k = 0; k < K; ++k) {
            if (N[k] > 0) {
                supervector.segment(k*D, D) = F[k] / N[k];
            } else {
                supervector.segment(k*D, D) = ubm.means[k]; // 回退到 UBM 均值
            }
        }
        return supervector;
    }
```

（5）说话人特征加密与传输

生成的均值超向量（例如 9984 维双精度浮点数，约 80KB）需要与视频元数据一起加密。由于超向量数据量较大，建议先进行量化压缩（如转换为 16 位整数）或使用 PCA 降维至 512 维（约 4KB），再放入 JSON 并与其它元数据一并加密。

在发送端集成流程大致如下:从音频采集缓冲区获取一段固定长度(如 2 秒)的 PCM 音频；对音频进行预处理（预加重、分帧、加窗）；提取 MFCC 序列；加载 UBM 参数，计算均值超向量；（可选）应用 PCA 降维；将说话人特征向量转换为 base64 字符串或浮点数组，加入元数据 JSON；调用 SM4 加密模块对整个 JSON 进行加密；发送加密包。

（6）优化思路

MFCC 提取和 GMM 后验概率计算在鲲鹏开发板上可能较慢。可采用以下优化：使用 NEON 指令加速向量运算；预先计算 UBM 的常数部分（如高斯分量的归一化因子）；降低帧率（如每秒提取一次特征，而不是每帧）。此外，UBM 参数（256 个高斯，39 维）约 256×39×8≈80KB，加上协方差约 80KB，可以接受。建议在单独线程中执行说话人特征提取，避免阻塞音视频采集。

如果开发板资源紧张，可将说话人特征提取任务放在接收端。但此时需要在网络上传输原始音频（加密后），会增大带宽。根据安全层设计，在发送端提取特征，更符合“最小化敏感数据暴露”原则。

通过以上实现，学生可以在鲲鹏开发板上完成从音频采集到说话人特征提取的全流程，为多模态认证提供可靠的声纹信息。该模块与视频人脸检测模块并行工作，最终在接收端融合，构建完整的国产化音视频传输认证系统。

1.4 接收端设计内容

1.4.1 网络接收与解包（基础层，必选）

接收端运行在学生笔记本电脑上，采用双线程分离接收架构：一个线程负责 UDP 接收（视频+音频），另一个线程负责 TCP 接收（元数据+控制信令），两者独立运行，互不阻塞。

（1）UDP 接收端（视频/音频）

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <cstring>
#include <thread>
class UdpReceiver {
private:
    int sockfd;
```

```
    bool running = false;
    std::thread recv_thread;

    // 回调函数，由外部实现具体的帧处理逻辑
    std::function<void(const uint8_t*, size_t, int)> on_data_received;
public:
    UdpReceiver(int port, std::function<void(const uint8_t*, size_t, int)> callback)
        : on_data_received(callback) {
        sockfd = socket(AF_INET, SOCK_DGRAM, 0);
        if (sockfd < 0) {
            perror("socket creation failed");
            return;
        }
        struct sockaddr_in addr;
        memset(&addr, 0, sizeof(addr));
        addr.sin_family = AF_INET;
        addr.sin_port = htons(port);
        addr.sin_addr.s_addr = INADDR_ANY;  // 接收来自任意 IP 的数据
        if (bind(sockfd, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
            perror("bind failed");
            return;
        }
    }
    void start() {
        running = true;
        recv_thread = std::thread([this]() {
            uint8_t buffer[65536];  // 足够容纳一个完整 UDP 包
            struct sockaddr_in src_addr;
            socklen_t addr_len = sizeof(src_addr);

            while (running) {
                ssize_t len = recvfrom(sockfd, buffer, sizeof(buffer), 0,
                                       (struct sockaddr*)&src_addr, &addr_len);
                if (len > 0 && on_data_received) {
                    // 将数据转发给回调函数处理
                    on_data_received(buffer, len, ntohs(src_addr.sin_port));
                }
            }
        });
    }
    ~UdpReceiver() {
        running = false;
        if (recv_thread.joinable()) recv_thread.join();
        close(sockfd);
```

```
    }
};
```

（2）TCP 接收端（元数据）

TCP 接收端作为服务器监听元数据连接，接收来自鲲鹏开发板的加密元数据，并传递给解密模块处理。

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <thread>
#include <functional>
class MetaDataTcpServer {
private:
    int listen_fd;
    int client_fd = -1;
    bool running = false;
    std::thread server_thread;
    std::function<void(const uint8_t*, size_t)> on_metadata;
public:
    MetaDataTcpServer(int port, std::function<void(const uint8_t*, size_t)> callback)
        : on_metadata(callback) {
        listen_fd = socket(AF_INET, SOCK_STREAM, 0);
        if (listen_fd < 0) {
            perror("socket creation failed");
            return;
        }
        // 设置端口复用，便于调试时快速重启
        int opt = 1;
        setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
        struct sockaddr_in addr;
        memset(&addr, 0, sizeof(addr));
        addr.sin_family = AF_INET;
        addr.sin_port = htons(port);
        addr.sin_addr.s_addr = INADDR_ANY;
        if (bind(listen_fd, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
            perror("bind failed");
            return;
        }
        if (listen(listen_fd, 1) < 0) {
            perror("listen failed");
            return;
        }
```

```
        }
        void start() {
            running = true;
            server_thread = std::thread([this]() {
                struct sockaddr_in client_addr;
                socklen_t addr_len = sizeof(client_addr);
                client_fd = accept(listen_fd, (struct sockaddr*)&client_addr, &addr_len);
                if (client_fd < 0) {
                    perror("accept failed");
                    return;
                }
                // 为 TCP 连接禁用 Nagle 算法
                int flag = 1;
                setsockopt(client_fd, IPPROTO_TCP, TCP_NODELAY, &flag,
sizeof(int));
                uint8_t len_buf[4];
                while (running) {
                    // 读取 4 字节长度头
                    ssize_t recv_len = recv(client_fd, len_buf, 4, MSG_WAITALL);
                    if (recv_len != 4) break;
                    uint32_t data_len = ntohl(*(uint32_t*)len_buf);
                    if (data_len == 0 || data_len > 10 * 1024 * 1024) break;  // 异常大小
保护
                    std::vector<uint8_t> data(data_len);
                    recv_len = recv(client_fd, data.data(), data_len, MSG_WAITALL);
                    if (recv_len != (ssize_t)data_len) break;
                    if (on_metadata) on_metadata(data.data(), data.size());
                }
                close(client_fd);
            });
        }
        ~MetaDataTcpServer() {
            running = false;
            if (client_fd != -1) close(client_fd);
            if (listen_fd != -1) close(listen_fd);
            if (server_thread.joinable()) server_thread.join();
        }
    };
```

（3）数据流整合

接收端需要将 UDP 收到的 H.264 视频数据和 AAC 音频数据分别送入对应的解码器，同时将 TCP 收到的元数据送入 SM4 解密模块。推荐采用队列+回调模式，确保各模块间解耦。

```
// 接收端主程序示例
int main() {
    // 视频解码器和音频解码器（参考 4.4.1 节）
    VideoDecoder video_dec;
    AudioDecoder audio_dec;
    // 视频 UDP 接收：端口 5004，数据送入视频解码器
    UdpReceiver video_receiver(5004, [&](const uint8_t* data, size_t len, int src_port) {
        // 将 UDP 包作为 AVPacket 送入解码器
        AVPacket pkt;
        av_init_packet(&pkt);
        pkt.data = (uint8_t*)data;
        pkt.size = len;
        video_dec.decode(&pkt);
    });
    // 音频 UDP 接收：端口 5006，数据送入音频解码器
    UdpReceiver audio_receiver(5006, [&](const uint8_t* data, size_t len, int src_port) {
        AVPacket pkt;
        av_init_packet(&pkt);
        pkt.data = (uint8_t*)data;
        pkt.size = len;
        audio_dec.decode(&pkt);
    });
    // TCP 元数据接收：端口 5008，数据送入解密模块
    MetaDataTcpServer meta_server(5008, [&](const uint8_t* data, size_t len) {
        // 调用 SM4 解密模块（参考 4.4.7 节）
        decrypt_and_process_metadata(data, len);
    });
    video_receiver.start();
    audio_receiver.start();
    meta_server.start();
    // 主线程等待（实际应处理窗口事件等）
    while (true) {
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
    return 0;
}
```

1.4.2 音视频解码（基础层，必选）

接收端音视频播放可以采用采用多线程架构，主要包含以下几个环节：从网络缓冲区读取压缩数据（H.264 视频包、AAC 音频包），通过 FFmpeg 解码为原始帧；将解码后的 YUV 帧转换为 OpenCV Mat 格式（BGR），用于显示和人脸识别；将解码后的 PCM 数据（可能为浮点格式）重采样为 SDL2 支持的 S16 格

式，送入音频播放缓冲区；视频通过 OpenCV 窗口按帧率显示，音频通过 SDL2 回调机制持续输出；以音频时钟为基准，视频帧根据 PTS 时间戳动态调整显示时机，实现音画同步。

（1）视频解码（H.264 → YUV）

```
#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include <libavutil/imgutils.h>
#include <libswscale/swscale.h>
#include <opencv2/opencv.hpp>
// 视频解码器初始化
class VideoDecoder {
private:
    AVCodecContext* codec_ctx = nullptr;
    AVFrame* frame = nullptr;
    SwsContext* sws_ctx = nullptr;

public:
    bool init(int codec_id, int width, int height) {
        // 1. 查找 H.264 解码器
        const AVCodec* codec = avcodec_find_decoder((AVCodecID)codec_id);
        if (!codec) return false;
        // 2. 分配解码器上下文
        codec_ctx = avcodec_alloc_context3(codec);
        if (!codec_ctx) return false;
        // 3. 设置解码参数（从发送端协商获取）
        codec_ctx->width = width;
        codec_ctx->height = height;
        codec_ctx->pix_fmt = AV_PIX_FMT_YUV420P;
        // 4. 打开解码器
        if (avcodec_open2(codec_ctx, codec, NULL) < 0) return false;

        // 5. 分配 AVFrame 存储解码后的帧
        frame = av_frame_alloc();
        // 6. 初始化图像格式转换器（YUV420P → BGR24）
        sws_ctx = sws_getContext(width, height, AV_PIX_FMT_YUV420P,
                                 width, height, AV_PIX_FMT_BGR24,
                                 SWS_BILINEAR, NULL, NULL, NULL);
        return true;
    }
    // 解码一帧：将压缩的 AVPacket 解码为 AVFrame
    bool decodeFrame(AVPacket* pkt, cv::Mat& bgr_frame, int64_t& pts) {
        // 发送压缩数据包到解码器
```

```
        int ret = avcodec_send_packet(codec_ctx, pkt);
        if (ret < 0) return false;

        // 循环接收解码后的帧（一个 packet 可能产生多帧）
        ret = avcodec_receive_frame(codec_ctx, frame);
        if (ret == AVERROR(EAGAIN) || ret == AVERROR_EOF) return false;
        if (ret < 0) return false;
        // 获取 PTS（显示时间戳）
        pts = frame->pts;
        // YUV → BGR 格式转换
        cv::Mat yuv_frame(cv::Size(codec_ctx->width, codec_ctx->height), CV_8UC3);
        uint8_t* dest[4] = {yuv_frame.data, NULL, NULL, NULL};
        int dest_linesize[4] = {3 * codec_ctx->width, 0, 0, 0};
        sws_scale(sws_ctx, frame->data, frame->linesize, 0, codec_ctx->height,
                  dest, dest_linesize);
        bgr_frame = yuv_frame.clone();
        av_frame_unref(frame);
        return true;
    }
    ~VideoDecoder() {
        if (sws_ctx) sws_freeContext(sws_ctx);
        if (frame) av_frame_free(&frame);
        if (codec_ctx) avcodec_free_context(&codec_ctx);
    }
};
```

其中，avcodec_send_packet 和 avcodec_receive_frame 是 FFmpeg 推荐的异步解码 API；avcodec_receive_frame 需要循环调用，因为一个 AVPacket 可能对应多个解码帧，返回值 AVERROR(EAGAIN)表示需要继续发送新的 packet，AVERROR_EOF 表示解码结束；视频解码后得到 YUV420P 格式数据，使用 sws_scale 转换为 OpenCV 可显示的 BGR24 格式，可直接通过 OpenCV 窗口显示。

```
#include <opencv2/opencv.hpp>
class VideoPlayer {
private:
    cv::Mat current_frame;
    cv::Mat display_frame;
public:
    void displayFrame(const cv::Mat& frame) {
        current_frame = frame.clone();
        cv::imshow("Video Stream", current_frame);
        cv::waitKey(1);   // 1ms 等待，允许窗口刷新
```

```
        }
    };
```

（2）音频解码（AAC → PCM）

```
    #include <libavcodec/avcodec.h>
    #include <libswresample/swresample.h>

    class AudioDecoder {
    private:
        AVCodecContext* codec_ctx = nullptr;
        AVFrame* frame = nullptr;
        SwrContext* swr_ctx = nullptr;
        uint8_t** converted_data = nullptr;
        int converted_linesize;

    public:
        bool init(int sample_rate, int channels, AVSampleFormat sample_fmt) {
            // 1. 查找 AAC 解码器
            const AVCodec* codec = avcodec_find_decoder(AV_CODEC_ID_AAC);
            if (!codec) return false;
            // 2. 分配解码器上下文
            codec_ctx = avcodec_alloc_context3(codec);
            codec_ctx->sample_rate = sample_rate;
            codec_ctx->channels = channels;
            codec_ctx->sample_fmt = sample_fmt;
            // 3. 打开解码器
            if (avcodec_open2(codec_ctx, codec, NULL) < 0) return false;
            // 4. 分配 AVFrame
            frame = av_frame_alloc();
            // 5. 初始化音频重采样器（将解码后的格式转换为 S16，供 SDL 播放）
            swr_ctx = swr_alloc();
            av_opt_set_int(swr_ctx,                                    "in_channel_layout",
av_get_default_channel_layout(channels), 0);
            av_opt_set_int(swr_ctx,                                   "out_channel_layout",
av_get_default_channel_layout(channels), 0);
            av_opt_set_int(swr_ctx, "in_sample_rate", sample_rate, 0);
            av_opt_set_int(swr_ctx, "out_sample_rate", sample_rate, 0);
            av_opt_set_sample_fmt(swr_ctx, "in_sample_fmt", sample_fmt, 0);
            av_opt_set_sample_fmt(swr_ctx, "out_sample_fmt", AV_SAMPLE_FMT_S16,
0);
            swr_init(swr_ctx);

            return true;
        }
```

```
        // 解码音频包并重采样
        bool decodeAudio(AVPacket* pkt, std::vector<uint8_t>& pcm_data, int64_t& pts) {
            int ret = avcodec_send_packet(codec_ctx, pkt);
            if (ret < 0) return false;
            ret = avcodec_receive_frame(codec_ctx, frame);
            if (ret == AVERROR(EAGAIN) || ret == AVERROR_EOF) return false;
            if (ret < 0) return false;
            pts = frame->pts;
            // 重采样：将解码后的音频（可能是 FLTP 浮点格式）转换为 S16 格式
            int out_samples = swr_get_delay(swr_ctx, codec_ctx->sample_rate) +
frame->nb_samples;
            int max_out_samples = out_samples * 2;  // 预留空间
            av_samples_alloc(&converted_data, &converted_linesize, codec_ctx->channels,
                             max_out_samples, AV_SAMPLE_FMT_S16, 0);
            int samples_converted = swr_convert(swr_ctx, converted_data,
max_out_samples, (const uint8_t**)frame->data, frame->nb_samples);
            if (samples_converted > 0) {
                int data_size = samples_converted * codec_ctx->channels *
av_get_bytes_per_sample(AV_SAMPLE_FMT_S16);      pcm_data.assign(converted_data[0],
converted_data[0] + data_size);
            }
            av_freep(&converted_data[0]);
            av_frame_unref(frame);
            return samples_converted > 0;
        }

        ~AudioDecoder() {
            if (swr_ctx) swr_free(&swr_ctx);
            if (frame) av_frame_free(&frame);
            if (codec_ctx) avcodec_free_context(&codec_ctx);
        }
    };
```

AAC 解码后的音频格式通常是 AV_SAMPLE_FMT_FLTP（32 位浮点平面格式），不能直接送给 SDL 播放，需要用 swr_convert 重采样为 AV_SAMPLE_FMT_S16（16 位有符号整型）；swr_convert 的输入输出格式设置需要仔细匹配，否则会产生噪声或解码失败；重采样后的 PCM 数据以交错格式存储，可直接送入 SDL 音频回调。av_samples_alloc 用于分配输出缓冲区内存，其参数包括声道数、采样数、采样格式和对齐方式。swr_convert 函数的返回值是转换后输出的样本数，乘以声道数和每个样本的字节数即可得到实际 PCM 数据大小。

SDL2 采用回调模式播放音频：SDL 内部维护一个音频线程，当音频设备需要更多数据时自动调用回调函数填充缓冲区。这种模式适合流式音频数据，延迟更低，缓冲控制更精细。

```
#include <SDL2/SDL.h>
#include <queue>
#include <mutex>
class AudioPlayer {
private:
    SDL_AudioDeviceID audio_dev;
    std::queue<std::vector<uint8_t>> audio_queue;  // 音频数据队列
    std::mutex queue_mutex;
    bool playing = false;
    // 静态回调函数
    static void audioCallback(void* userdata, Uint8* stream, int len) {
        AudioPlayer* player = (AudioPlayer*)userdata;
        player->fillAudioBuffer(stream, len);
    }
    void fillAudioBuffer(Uint8* stream, int len) {
        std::lock_guard<std::mutex> lock(queue_mutex);
        SDL_memset(stream, 0, len);  // 清空缓冲区
        if (audio_queue.empty()) return;
        int remaining = len;
        int offset = 0;
        while (remaining > 0 && !audio_queue.empty()) {
            std::vector<uint8_t>& pcm_data = audio_queue.front();
            int copy_len = std::min(remaining, (int)pcm_data.size());
            memcpy(stream + offset, pcm_data.data(), copy_len);
            if (copy_len < pcm_data.size()) {
                // 数据未完全复制，保留剩余部分
                std::vector<uint8_t> remaining_data(pcm_data.begin() + copy_len,
pcm_data.end());
                audio_queue.pop();
                audio_queue.push(std::move(remaining_data));
            } else {
                audio_queue.pop();
            }
            remaining -= copy_len;
            offset += copy_len;
        }
    }
public:
    bool init(int sample_rate, int channels) {
        if (SDL_Init(SDL_INIT_AUDIO) < 0) return false;
```

```
        SDL_AudioSpec desired;
        desired.freq = sample_rate;
        desired.format = AUDIO_S16SYS;    // 系统字节序的 S16 格式
        desired.channels = channels;
        desired.samples = 4096;                  // 缓冲区大小
        desired.callback = audioCallback;
        desired.userdata = this;
        audio_dev = SDL_OpenAudioDevice(NULL, 0, &desired, NULL, 0);
        if (audio_dev == 0) return false;
        playing = true;
        SDL_PauseAudioDevice(audio_dev, 0);  // 开始播放
        return true;
    }
    void pushAudio(const std::vector<uint8_t>& pcm_data) {
        std::lock_guard<std::mutex> lock(queue_mutex);
        audio_queue.push(pcm_data);
    }
    ~AudioPlayer() {
        if (audio_dev) {
            SDL_PauseAudioDevice(audio_dev, 1);
            SDL_CloseAudioDevice(audio_dev);
        }
        SDL_Quit();
    }
};
```

回调函数 fillAudioBuffer 由 SDL 音频线程自动调用，必须在短时间内返回，不能执行阻塞操作。SDL_MixAudio 可用于将多路音频混合，本系统可采用单路音频，直接 memcpy 填充即可，AUDIO_S16SYS 表示使用系统原生字节序的 16 位有符号整数，跨平台兼容性好。

（3）音视频同步（PTS 时间戳机制）

PTS（Presentation TimeStamp）是渲染用的时间戳，表示视频帧或音频帧应该被展示/播放的时间点。在 FFmpeg 中，每个 AVPacket 和 AVFrame 都携带 PTS 值，其单位是时间基（time_base）。time_base 是一个有理数（分子/分母），表示 PTS 每个单位对应的秒数。

时间基换算：要将 PTS 转换为实际秒数，需要乘以时间基：

$$\text{实际时间(秒)} = \text{pts} \times (\text{time\_base.num} / \text{time\_base.den})$$

可以考虑采用音频为基准、视频向音频同步的方案，因为人耳对音频的卡顿和不连续比视觉更敏感。具体实现如下：维护一个音频时钟（audio_clock），记录当前正在播放的音频的 PTS 值；视频解码时，记录每一帧的 PTS 值；在显示视频帧之前，获取当前音频时钟，计算视频帧的 PTS 与音频时钟的差值；如果视频帧 PTS 超前于音频（视频太快），则延迟显示；如果视频帧 PTS 落后于音频（视频太慢），则跳过该帧。

```
#include <chrono>
#include <thread>
class AVSyncPlayer {
private:
    VideoDecoder video_dec;
    AudioDecoder audio_dec;
    VideoPlayer video_player;
    AudioPlayer audio_player;

    double audio_clock = 0.0;          // 当前音频播放的时间点（秒）
    int64_t last_audio_pts = 0;
    AVRational audio_time_base;

    double video_clock = 0.0;          // 当前视频播放的时间点（秒）
    int64_t last_video_pts = 0;
    AVRational video_time_base;

    std::queue<cv::Mat> video_frame_queue;
    std::queue<std::pair<int64_t, AVRational>> video_pts_queue;   // 存储帧的 PTS 和 time_base
public:
    // 更新音频时钟（在音频解码后调用）
    void updateAudioClock(int64_t pts, AVRational time_base) {
        if (pts != AV_NOPTS_VALUE) {
            audio_clock = pts * av_q2d(time_base);
        }
    }
    // 同步显示视频帧
    void displayVideoSync() {
        if (video_frame_queue.empty()) return;
        cv::Mat frame = video_frame_queue.front();
        auto [pts, time_base] = video_pts_queue.front();
        double video_pts_seconds = pts * av_q2d(time_base);
        // 计算视频帧与音频时钟的差值
        double diff = video_pts_seconds - audio_clock;
```

```
            if (diff > 0) {
                // 视频超前于音频，需要延迟显示
                if (diff > 0.01) {   // 差值大于 10ms 才延迟
std::this_thread::sleep_for(std::chrono::milliseconds((int)(diff * 1000)));
                }
                video_player.displayFrame(frame);
                video_frame_queue.pop();
                video_pts_queue.pop();
            } else if (diff < -0.05) {
                // 视频落后于音频超过 50ms，丢弃当前帧，快速追赶
                video_frame_queue.pop();
                video_pts_queue.pop();
            } else {
                // 同步误差在可接受范围内，正常显示
                video_player.displayFrame(frame);
                video_frame_queue.pop();
                video_pts_queue.pop();
            }
        }
};
```

（4）音视频播放示例

```
#include <thread>
#include <queue>
int main() {
    // 1. 初始化解码器和播放器
    VideoDecoder video_dec;
    AudioDecoder audio_dec;
    AudioPlayer audio_player;
    AVSyncPlayer sync_player;
    // 2. 初始化（参数需从发送端协商获取）
    video_dec.init(AV_CODEC_ID_H264, 1280, 720);
    audio_dec.init(44100, 2, AV_SAMPLE_FMT_FLTP);
    audio_player.init(44100, 2);
    // 3. 接收循环（从网络获取音视频包，分别解码）
    while (true) {
        // 接收音频包
        AVPacket audio_pkt = receive_audio_packet();
        if (audio_pkt.size > 0) {
            std::vector<uint8_t> pcm_data;
            int64_t pts;
            if (audio_dec.decodeAudio(&audio_pkt, pcm_data, pts)) {
                // 送入音频播放队列
                audio_player.pushAudio(pcm_data);
```

```
                    // 更新音频时钟
                    sync_player.updateAudioClock(pts, audio_time_base);

                    // 同时将解码后的音频数据送入说话人识别模块
                    process_speaker_recognition(pcm_data);
                }
            }
            // 接收视频包
            AVPacket video_pkt = receive_video_packet();
            if (video_pkt.size > 0) {
                cv::Mat bgr_frame;
                int64_t pts;
                if (video_dec.decodeFrame(&video_pkt, bgr_frame, pts)) {
                    // 送入同步显示队列
                    sync_player.queueVideoFrame(bgr_frame, pts, video_time_base);
                    // 同时将解码后的视频帧送入人脸识别模块
                    process_face_recognition(bgr_frame);
                }
            }
            // 音视频同步显示
            sync_player.displayVideoSync();
        }
        return 0;
    }
```

提取出的比特流按帧顺序拼接，解码成原始水印字符串。

（5）常见问题与解决思路

具体见表 1.12。

表 1.12 解码常见问题与解决思路

问题现象	可能原因	解决思路
视频解码花屏	H.264 参数不匹配（SPS/PPS 缺失）	确保发送端发送关键帧时附带 extradata，接收端在解码前解析并设置 codec_ctx->extradata
音频播放有噪音	重采样格式不匹配	检查 swr_convert 的输入输出格式设置，确保与 SDL 期望的 AUDIO_S16SYS 一致
音画不同步	PTS 未正确传递或计算错误	确认解码后 AVFrame->pts 有值，若为 AV_NOPTS_VALUE，需根据帧率自行计算
画面卡顿或闪烁	帧率控制不当	cv::waitKey()的延迟时间应与视频帧率匹配（如 30fps 约 33ms），或用 PTS 精确控制
SDL 初始化失败	音频设备被占用或格式不支持	检查 SDL_OpenAudioDevice 返回值，可先用 SDL_GetError()获取详细错误信息
解码器找不到	FFmpeg 未编译对应解码器	使用 avcodec_find_decoder(AV_CODEC_ID_H264)前确保 FFmpeg 启用了 libx264 支持

通过以上步骤，学生可以在接收端完整实现网络音视频流的解码分离、同步播放，构建完整的国产化音视频传输系统。

1.4.3 SM4 解密（安全层，可选）

接收端需要从数据包中提取盐值、IV 和密文，使用相同的用户密码执行 PBKDF2 派生密钥后进行解密。

```
class SM4Receiver {
public:
    // 从接收到的数据包解密元数据
    bool processReceivedPacket(const uint8_t* received_data, size_t data_len,
                               const std::string& password,
                               std::string& decrypted_json) {
        // 1. 解析数据包
        if (data_len < 32) return false;
        EncryptedPacket packet;
        memcpy(packet.salt, received_data, 16);
        memcpy(packet.iv, received_data + 16, 16);
        packet.ciphertext.assign(received_data + 32, received_data + data_len);
        // 2. 使用密码和盐值派生密钥
        uint8_t key[16];
        if (!deriveKeyFromPassword(password, packet.salt, key)) {
            return false;
        }
        // 3. 解密
        std::vector<uint8_t> plaintext;
        int ret = sm4_cbc_decrypt(packet.ciphertext.data(),
                                  packet.ciphertext.size(),
                                  key, packet.iv, plaintext);
        if (ret < 0) return false;

        decrypted_json = std::string(plaintext.begin(), plaintext.end());
        return true;
    }
};
```

需要注意的是，发送端和接收端必须使用相同的用户密码，才能派生出相同的密钥进行解密。在本系统中，密码由学生自行确定。

1.4.4 视频水印提取（溯源层，可选）

需要从视频帧中提取水印，用于验证完整性及溯源。

```
std::vector<uint8_t> extractWatermark(cv::Mat& frame) {
    // 转换为 YUV，提取 Y 通道
```

```
        cv::Mat yuv;
        cv::cvtColor(frame, yuv, cv::COLOR_BGR2YUV);
        cv::Mat y_channel;
        cv::extractChannel(yuv, y_channel, 0);
        int rows = y_channel.rows / 8;
        int cols = y_channel.cols / 8;
        const std::vector<cv::Point> mid_freq = { {4,4}, {4,5}, {5,4}, {5,5}, {4,6}, {5,6},
{6,4}, {6,5} };
        std::vector<uint8_t> bits;
        for (int idx = 0; idx < 32; idx++) {
            int i = idx / cols;
            int j = idx % cols;
            cv::Rect roi(j*8, i*8, 8, 8);
            cv::Mat block = y_channel(roi);
            block.convertTo(block, CV_32F, 1.0/255.0);
            cv::Mat dct_block;
            cv::dct(block, dct_block);
            cv::Point pos = mid_freq[idx % mid_freq.size()];
            float coeff = dct_block.at<float>(pos.y, pos.x);
            // 根据系数正负判断比特（假设原始系数均值接近 0）
            uint8_t bit = (coeff > 0) ? 1 : 0;
            bits.push_back(bit);
        }
        return bits;
    }
    // 解码水印比特为字符串
    std::string decodeWatermark(const std::vector<uint8_t>& bits) {
        std::string result;
        for (size_t i = 0; i + 8 <= bits.size(); i += 8) {
            char c = 0;
            for (int j = 0; j < 8; j++) {
                if (bits[i+j]) c |= (1 << j);
            }
            if (c != 0) result += c;
        }
        return result;
    }
```

1.4.5 音频水印提取（溯源层，可选）

提取过程无需原始音频，只需知道量化步长 Δ 和嵌入位置。

```
    int extract_bit(float coeff, float delta) {
        int q_int = round(coeff / delta);
        return q_int % 2;
    }
```

提取出的比特流按帧顺序拼接，解码成原始水印字符串。

1.4.6 人脸识别（智能层，可选）

接收端从网络接收到加密包，解密后得到 send_feature。从数据库中读取该用户（或所有用户）的注册特征 registered_feature。计算 send_feature 与 registered_feature 的余弦相似度（或欧氏距离）。若相似度超过预设阈值，则认证通过；否则不通过。

```
// 解密后的发送端特征向量（假设已获得）
std::vector<float> send_feature = decrypted_metadata.face_feature;
// 获取本地注册特征（例如用户名为 Alice）
std::vector<float> registered_feature = getRegisteredFeature("Alice");
if (registered_feature.empty()) {
    std::cout << "No registered face for this user." << std::endl;
    return false;
}
// 计算余弦相似度
double similarity = cosineSimilarity(send_feature, registered_feature);
double threshold = 0.75;
if (similarity >= threshold) {
    std::cout << "Face verification passed. Similarity = " << similarity << std::endl;
    // 认证通过，允许后续操作
} else {
    std::cout << "Face verification failed. Similarity = " << similarity << std::endl;
    // 触发异常处理
}
```

1.4.7 说话人识别（智能层，可选）

接收端收到解密后的元数据，获得说话人特征向量，与预先注册的说话人模型进行比对。比对可采用余弦相似度或计算对数似然比。

```
//比对发送来的特征与注册特征
double cosineSimilarity(const Eigen::VectorXd& feat1, const Eigen::VectorXd& feat2) {
    return feat1.dot(feat2) / (feat1.norm() * feat2.norm());
}
```

若相似度超过阈值（如 0.7），则说话人认证通过，与人脸识别结果进行多模态融合决策。

1.4.8 多模态融合（智能层，可选）

在接收端的多模态认证模块中，最终决策可融合人脸比对分数和声纹比对分数：

```
double face_score = cosineSimilarity(send_face_feature, registered_face_feature);
double voice_score = cosineSimilarity(send_voice_feature, registered_voice_feature);

// 动态权重融合（示例）
double fusion_score = 0.6 * face_score + 0.4 * voice_score;
bool authenticated = (fusion_score >= 0.75);
```

发送端上传的特征已经经过 SM4 加密，接收端解密后才进行比对，防止中间人窃取特征。可对 openGauss 数据库中的特征 BLOB 进行加密存储（如使用 aes_encrypt 和 aes_decrypt。）。在元数据中加入时间戳和随机数，接收端验证其时效性，可以抵抗防重放攻击。

二、设计报告要求

设计报告要围绕“设计过程截图+流程图+对比测试分析+UI 界面”几个核心要素，撰写时应重点突出技术路线的合理性、实现过程的完整性和测试分析的充分性。报告的大致框架描述如下：

（1）设计题目概述（5 分）

简要概述题目背景，说明智慧城市、智能安防等应用场景下音视频传输与身份认证的实际需求。切勿长篇大论，控制在 1 页以内。

（2）具体任务（5 分）

简要概述本系统要实现的具体功能与目标，包括：发送端功能，如音视频采集、人脸检测、编码压缩、水印嵌入、SM4 加密、网络传输；接收端功能，如数据接收、解密、水印验证、多模态认证、音视频播放；四层递进任务中选择的层次及完成目标。简要说明，切勿长篇大论。

（3）技术方案及关键问题（30 分）

围绕技术路线，配图说明发送端与接收端架构，标注各模块使用的关键技术（OpenCV/YOLO、FFmpeg、SM4、openGauss、鲲鹏开发板等），列出拟解决的核心问题（如多模态融合准确性、水印鲁棒性、国密算法性能优化、国产技术适配等）。注意设计报告不是技术科普，需简明扼要。

（4）系统设计实现及测试（50 分）

此部分为报告核心，需详细阐述，可参考以下提纲组织：

①系统总体设计

给出系统架构图，说明发送端与接收端模块划分；给出数据流程图，展示“采集→处理→传输→计算→存储→应用”全闭环流程；说明四层递进任务的设计思路及本课题所选层次。

②发送端设计与实现

音视频采集模块需截图说明采集界面，给出流程图；人脸检测模块需说明YOLO/OpenCV 实现方式，给出检测结果截图；编码压缩模块需说明 FFmpeg 编码参数，给出流程图；水印嵌入模块需给出水印嵌入算法流程图，说明鲁棒性设计；加密模块需说明 SM4 算法调用方式，给出加解密流程图；网络传输模块要说明 TCP/UDP 协议选择，给出 Socket 编程流程图。

③接收端设计与实现

数据接收模块应给出接收流程图；解密模块需说明 SM4 解密流程；水印提取与验证模块要给出水印提取流程图，记录提取成功率；多模态认证模块要描述人脸识别实现方式，说话人识别实现方式，交叉融合策略（附公式、流程图）；数据存储模块应说明 openGauss 数据库表结构设计，给出截图；音视频同步播放需给出播放界面截图。

④测试与分析

给出测试环境，鲲鹏开发板配置、openGauss 版本、依赖库版本等。列表说明各功能模块测试结果。记录响应时间、识别准确率、水印提取成功率、加解密耗时等数据。进行至少 2 种方法的对比测试，如不同 YOLO 版本（v5s vs v5n）在 NPU 上的推理速度与精度，SM4 vs 轻量级对称加密算法（如 TEA）的性能对比，不同嵌入强度下音视频水印的 PSNR 与鲁棒性对比等，给出表格和图表。

⑤UI 界面设计

给出发送端和接收端的 UI 界面截图，说明各功能按钮。

（5）课程设计总结与心得体会（5 分）

①设计总结（1 页，不要图、表）

说明四层任务中已完成的功能模块，在原方案基础上的改进或创新（如优化水印算法、改进融合策略、提升性能等），未完成或可优化的部分，后续改进方向等。

②心得体会（1 页，不要图、表）

阐述通过此次设计取得的收获，对国产根技术（鲲鹏、openGauss、SM4 等）的认识，对系统设计方法论的体会，对工程伦理与隐私保护的思考等。

（6）参考文献（5 分）

要求 5 篇以上，文中正确标注引用（上标格式），类型包括正规出版的书籍、期刊论文、学术会议论文等，格式参考国家标准 GB/T 7714-2015《信息与文献·参考文献著录规则》。

三、评价标准

采用设计报告与现场验收相结合的考核方式，各占总成绩的 50%。评价标准围绕系统能力培养的核心要素进行设计，具体如下：

（1）设计报告评价标准（50%）

设计报告重点考察学生对系统设计的整体把握、技术思路的清晰程度以及总结反思的能力，见表 3.1。

表 3.1 设计报告评价标准

评价维度	优秀（90-100）	良好（80-89）	中等（60-79）	不合格（0-59）
报告规范性与按时提交	报告撰写规范、结构清晰、图文并茂，按时提交	报告撰写规范，按时提交	报告撰写基本规范，按时提交	报告撰写不规范，或未按时提交
系统设计思路	系统设计思路清晰，完整反映“采集→处理→传输→计算→存储→应用”全闭环流程，体现全局观与模块化思维	正确反映系统设计技术思路	基本反映系统设计技术思路	未能清晰反映系统设计思路
创新与拓展	在原方案基础上增加了新的设计内容或功能，体现独立思考与拓展能力	——	——	未体现创新或拓展
测试与验证	测试用例设计充分、覆盖典型场景，测试数据记录完整，分析深入	测试用例适中，记录完整	测试用例一般，基本能验证功能	测试用例不充分，缺乏有效验证

（2）验收评价标准（50%）

现场验收重点考察系统实现质量、工程实践能力、表达沟通能力以及学生的应变能力，见表 3.2。

表 3.2 验收评价标准

评价维度	优秀（90-100）	良好（80-89）	中等（60-79）	不合格（0-59）

系统实现质量	技术路线合理、代码规范、调试流程熟练，系统稳定可靠，功能完整	技术路线合理、代码规范、调试流程熟练，系统运行正常	技术路线较合理、代码较规范、能完成基本调试，系统基本功能实现	技术路线不合理、代码不规范、调试流程不熟悉，系统无法运行或主要功能缺失
系统表达与沟通	能流畅、清晰阐述系统设计思路、技术选型理由、创新点与不足，表达有条理	能流畅表达设计思路，分析可能存在的问题	表达较流畅，基本能说明设计思路	表达不清晰，思路混乱
问题分析与应变	能准确分析系统存在的问题及成因，提出改进思路，正确回答教师追问	能正确回答教师提问	能基本回答教师提问	无法正确回答教师提问
系统集成与完整性	模块协同良好，体现“采集→处理→传输→计算→存储→应用”全闭环设计，系统能够“跑得通、看得见”	系统功能完整，模块集成良好	系统基本功能实现，模块间能基本协同	模块间无法协同，或系统功能缺失严重

（3）最终成绩评定

设计报告与现场验收各占 50%，两项加权计算总评成绩；四层递进任务按层次设定总评成绩上限：基础层最高 70 分，安全层 80 分，溯源层 90 分，智能层 100 分，鼓励学生挑战高阶任务；现场验收环节设置企业导师参与质询，从产业视角评估系统设计的工程性与实用性。