Here is the English translation of the document.

---

final.md (Page 1/6)

Source Code for High-Security UAV Data Exchange System Based on Physical Layer Security Technology

Author: Tsou Ying-Chi | **Date:** 2023/6/9

This document organizes the source code used in this project. To facilitate future modifications by users, the author has removed the parts of the code actually used in the deployment and reorganized some naming conventions. Please be sure to familiarize yourself with the usage of the classes.

Code Module List:

- **datastream:** Contains all functions for data exchange between different interfaces. Includes the transmission interface for ESP32 CSI data via serial, and the transmission interface for Raspberry Pi data exchange via socket.
- **IoD_UI:** The user interface module designed to present the implementation results of the project. Created using QtDesigner.
- **plkg:** Implements PLKG-related code and data encryption code.

Non-module Parts:

- The files stored in the `demo` folder demonstrate the basic functions of the modules. You only need to drag them into the main folder (the folder where the modules are located) to see their effects.
- The remaining unpacked external programs are demonstration files for the project implementation results. They were written in a relatively ad-hoc manner. It is recommended that future users, when building upon this research, use the three provided modules to reconstruct their own test environment to optimize performance.

Environment:

- python 3.8
- Package manager: conda

datastream

chat.py

Used to establish a socket between devices for data exchange. Devices must be connected to the same network. Data exchange can proceed after entering the IP/PORT of the device on that network. (For details, please refer to an Introduction to Networks textbook).

Required Modules:

- socket
- threading
- re

Code Explanation:

Function Name	Description
`chat_manager(<string ip>,<int port>)`	Both devices enter the IP/PORT of the device they wish to exchange data with to proceed. The default PORT is 5000. If you wish to establish two connections with the same device, please use different PORTs (Networking basics).

final.md (Page 2/6)

Function Name	Description
`chat_manager.receive_task()`	Continuously performs the task of receiving data.
`chat_manager.recv_init()`	Initializes the data receiving function.
`chat_manager.send_init()`	Initializes the data transmission function.
`chat_manager.chat_init()`	Initializes both receiving and transmission functions simultaneously.
`chat_manager.close_socket()`	Closes both receiving and sending sockets simultaneously.
`chat_manager.pop()`	Returns the first character using FIFO method. Return format is utf-8 code.
`chat_manager.pop_line()`	Returns continuous data transmitted via `chat_manager.send_line()` using FIFO method. Please avoid using "-end" as this is the termination symbol. Return format is string.
`chat_manager.read_queue()`	Reads all data currently in memory at once.
`chat_manager.queue_clear()`	Clears all accumulated data in memory.
`chat_manager.send(<string message>)`	Transmits data to the receiver. Automatically performs utf-8 encoding.
`chat_manager.send_line(<string message>)`	Transmits data to the receiver. The receiver can use `chat_manager.pop_line()` to retrieve the entire segment of data.
`chat_manager.send_original(<utf-8 message>)`	Transmits directly after receiving utf-8 encoded data.

Basic Function Demonstration:

```
from datastream import chat
import time
import threading

def show(chat):
    while True:
        time.sleep(0.3)
        message = chat.read_queue()
        if len(message) > 0:
            print(message.decode("utf-8"))

Alice = chat.chat("192.168.0.143") # Fill in the other party's IP
Alice.chat_init()
show_thread = threading.Thread(target=show, args=(Alice,))
show_thread.start()

while True:
    Alice.send(input('>>'))
```

---

final.md (Page 3/6)

The code above implements simultaneous data transmission and reception and can be used as a reference for your design.

csi_interface.py

The interface for exchanging data with the ESP32 when collecting CSI data. It also implements the Channel probing interface here. Those who wish to modify the data collection strategy in the future should look here.

Required Modules:

- serial
- threading
- platform
- re
- csv

Code Explanation:

Function Name	Description
`send(<string comport>, <string message>)`	Transmits data to other serial devices via serial.
`read(<string comport>)`	Receives data from a specific serial port.
`savetocsv(<string filename>, <string data>)`	Saves the data obtained from `com_esp.aquire_csi()`. First input the filename, then input the object to save.
`com_esp(<string comport>, <int baudrate>)`	Controls the data exchange protocol. Interacts with the protocol of the setup ESP32. Input comport and baud rate.
`com_esp.set_port(<string comport>,<string baudrate>)`	Resets the comport and baud rate, adjusting the target receiving device.

Function Name	Description
`com_esp.set_channel(<int channel>)`	Sets the channel for CSI data collection.
`com_esp.set_ping_f(<int frequency>)`	Sets the data collection frequency of the ESP32.
`com_esp.set_timeout(<int t>)`	Sets the duration for data collection.
`com_esp.__monitor()`	Optimized data collection core.
`com_esp.start_monitor()`	Starts the data collection thread.
`com_esp.stop_monitor()`	Stops data collection.
`com_esp.clear_queue()`	Clears the data collection buffer.
`com_esp.send(<string message>)`	Transmits data to the configured ESP32.
`com_esp.send_command(<string command>)`	Transmits specific commands to the ESP32: `ping`: start pinging CSI for the receiver to collect; `recv`: start collecting CSI data; `check`: confirm if the ESP32 side is alive; `restart`: restart the ESP32 program.

---

final.md (Page 4/6)

Function Name	Description
`com_esp.aquire_csi()`	Organizes received data into a series of strings and returns them. Format is "string", "string" ... "string", "string".
`com_esp.run_collection(<bool priority>,<int frequency>, <int timeout>)`	Executes a designed data collection protocol. `priority` determines the data collection order, `frequency` determines the collection rate, `timeout` ensures the system calculation time is long enough to guarantee system stability.

load.py

Converts data from `csi_interface.py` into arrays for convenient use. It removes unused CSI subcarriers.

Required Modules:

- csv
- re
- numpy
- math

Function Name	Description
`amplitude(<float real>, <float imag>)`	Calculates the magnitude by adding the real and imaginary parts.

Function Name	Description
`transform(<CSI data from com_esp.aquire_csi()>)`	Converts data into a usable format and returns `[CSIdata],[CSIdata],[CSIdata]...[CSIdata]`.
`load(<string filename>)`	Retrieves previously saved data to reproduce past experimental results.

plkg

aes.py

Used for AES encryption. Inputs the key and message to perform encryption/decryption.

Required Modules:

- Crypto
- binascii

Function Name	Description
`encrypt(<string plain_text>,<string secret_key>)`	Performs data encryption.
`decrypt(<utf-8 encrypted_text>,<string secret_key>)`	Decrypts received utf-8 encrypted data.
`byte_check(<unKnownTypeText>)`	Checks the type of the string.

ecc.py

Bit error correction module.

---

final.md (Page 5/6)

Required Modules:

- bchlib
- random
- math

Function Name	Description
`bit_flip(<string binary>)`	Randomly flips 0/1 in binary data with a 10% probability.
`rand_sequence(<int num>)`	Randomly generates binary coding.
`binary_byte_convertor(<string data>)`	Converts string binary data to byte.
`byte_binary_convertor(<string data>)`	Converts byte binary data to string.
`run_xor(<string binary_sequence1>,<string binary_sequence1>)`	Performs XOR between binary strings.

Function Name	Description
`check_same(<string binary_sequence1>, <string binary_sequence1>)`	Checks if two binary data entries are exactly the same.
`BCH_gen()`	Generates a random code equipped with bit error correction codes.
`reconciliation_encode(<string quantization_result>)`	Encodes the quantization result.
`reconciliation_decode(<string quantization_result>)`	Decodes the quantization result.

greycode_quantization.py

Quantization strategy set.

Required Modules:

- copy

Function Name	Description
`average(<transform data>)`	Accumulates CSI data into an average.
`swap(bit,x)`	Greycode algorithm.
`gray_code_gen(<int number_of_bits>)`	Function to generate greycode, returns n-bit grey code.
`quantization_1(<string probing_result>,<int number_of_bits_in_greycode>,<int how_many_bits_for_one_greycode>)`	Executes the quantization strategy and returns the quantization result.

sha256.py

Performs privacy amplification.

Required Modules:

---

final.md (Page 6/6)

- hashlib

Function Name	Description
`sha_byte(<string quantization_result>)`	Performs privacy amplification on the quantization result using sha256.

plkg.py

Integrated sub-module. Can run the complete PLKG protocol.

Required Modules:

- time

Function Name	Description
`end_device(<string device_tag>)`	Sets up an independent data collection device. Device name needs to be set: UAV "U", IoT device "I".
`end_device.set_chatmanager(<class chatmanager>)`	Sets the `chat_manager` for data transmission.
`end_device.save_probing_result(<string filename>)`	Sets whether to save the quantization result to a file.
`end_device.time_synchronize()`	When collecting and exchanging data, performs time synchronization first to ensure stability.
`end_device.channel_probing()`	Executes channel probing.
`end_device.quantization()`	Executes quantization.
`end_device.information_reconciliation()`	Executes information reconciliation.
`end_device.privacy_amplification()`	Executes privacy amplification.
`end_device.plkg()`	Executes complete plkg.
`end_device.key`	The plkg key.
`end_device.quantization_result`	The quantization result.