CS 111 week 9
Project 4C: IOT security

Tianxiang Li

Project 4C: Overview

Connect the BeagleBone Board to the assignment server via network

PartA:

1. Receive commands from the server
2. Report temperature to the server

PartB:

1. Encrypt the communication between the BeagleBone Board and the server with SSL.

lab4c_tcp

- builds and runs on your Beaglebone.
- based on the temperature sensor app (project4b)
  - including the --period=, --scale= and --log= options
- accepts the following (mandatory) new parameters:
  - --id=*9-digit-number*
  - --host=*name or address*
  - --log=*filename*
  - *port number*
- accepts the same commands and generates the same reports as Proj4b
- but now I/O from/to a network connection to a server.
  - open a TCP connection to the server
  - immediately send (and log) an ID terminated with a newline: **ID=***ID-number*
  - as before,
    - send (and log) temperature reports *over the connection*
    - process (and log) commands received *over the connection*
    - the last command sent by the server will be an **OFF**.
  - unlike the previous project,
    - button will not be used for manual shutdown.

PartA: Receive Commands/Send temperatures to the server

- BeagleBone talks to the server via network
  - Server name (--host) and port number will be passed via command line arguments

	TCP Logging Server	TLS Logging Server
HOST	lever.cs.ucla.edu	lever.cs.ucla.edu
PORT	18000	19000
STATUS	URL	URL

- Network APIs for Client side:
  - Create a socket (socket(2))
  - If don't know the IP address of the server, get it (gethostbyname(3))
  - initiate the connection to the server (connect(2))

Network Programming Primer: Client side code

For brevity, error handling is omitted

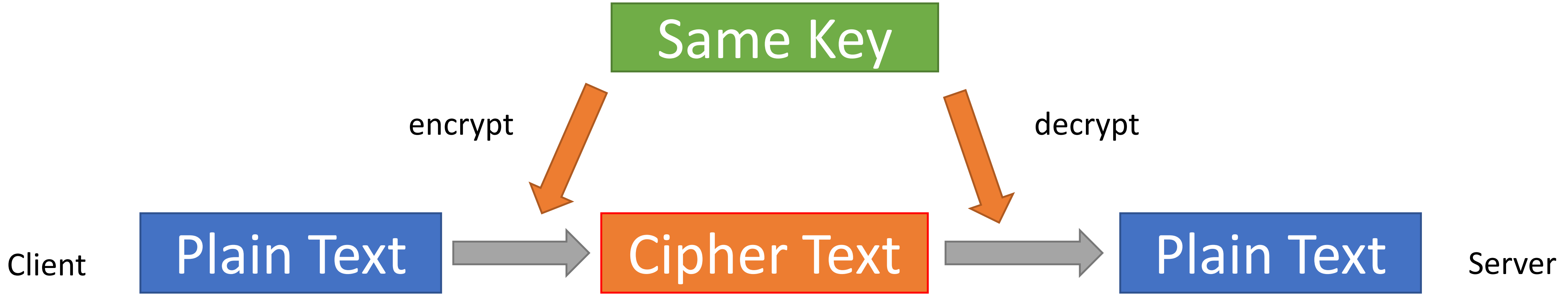
```
int client_connect(char * host_name, unsigned int port)
//e.g. host_name:"lever.cs.ucla.edu", port:18000, return the socket for subsequent communication
{
        struct sockaddr_in serv_addr; //encode the ip address and the port for the remote server
        int sockfd = socket(AF_INET, SOCK_STREAM, 0);
        // AF_INET: IPv4, SOCK_STREAM: TCP connection
        struct hostent *server = gethostbyname(host_name);
        // convert host_name to IP addr
        memset(&serv_addr, 0, sizeof(struct sockaddr_in);
        serv_addr.sin_family = AF_INET; //address is Ipv4
        memcpy(&serv_addr.sin_addr.s_addr, server->h_addr, server->h_length);
        //copy ip address from server to serv_addr
        serv_addr.sin_port = htons(port); //setup the port
        connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));  //initiate the connection to server
        return sockfd;
}
```

Overview

```
int main(int argc, char * argv[]) {
        id, log, host, port = process_cmd_line_arg(argc, argv);
        socket = client_connect(host, port);
        Write id to socket; //nine digit uid for debugging ("ID=%s\n")
        initialize_the_sensors();
        while (true) {
                if (it is time to report temperature && !stop)
                        read from temperature sensor,
                        convert and report to logfile and socket
                // use poll syscalls, 1s or smaller timeout interval
                if (there are input from socket) {
                        read from socket till encountering '\n' (thus we get an command)
                        process the command.
                }
        }
}
```

PartB: Encrypt the communication with TLS

- Symmetric Encryption
  - Same key needs to be provided to both client and server → How to safely exchange the key? → Asymmetric encryption

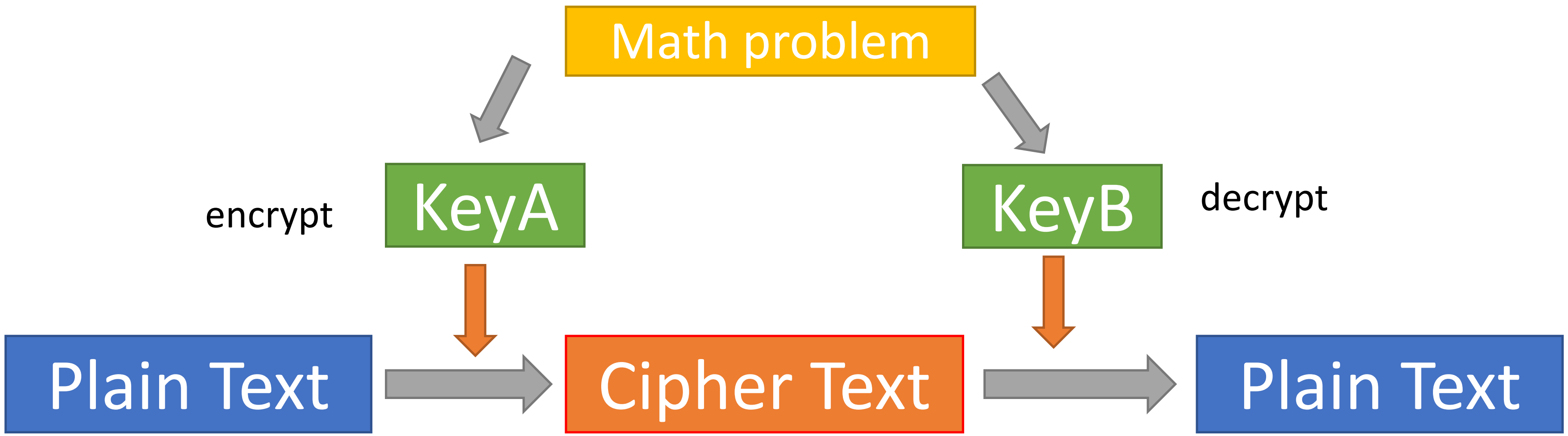


- *Note: these slides just introduces what's going on in the SSL/TLS connection, it is for your own reading interest*

Asymmetric encryption

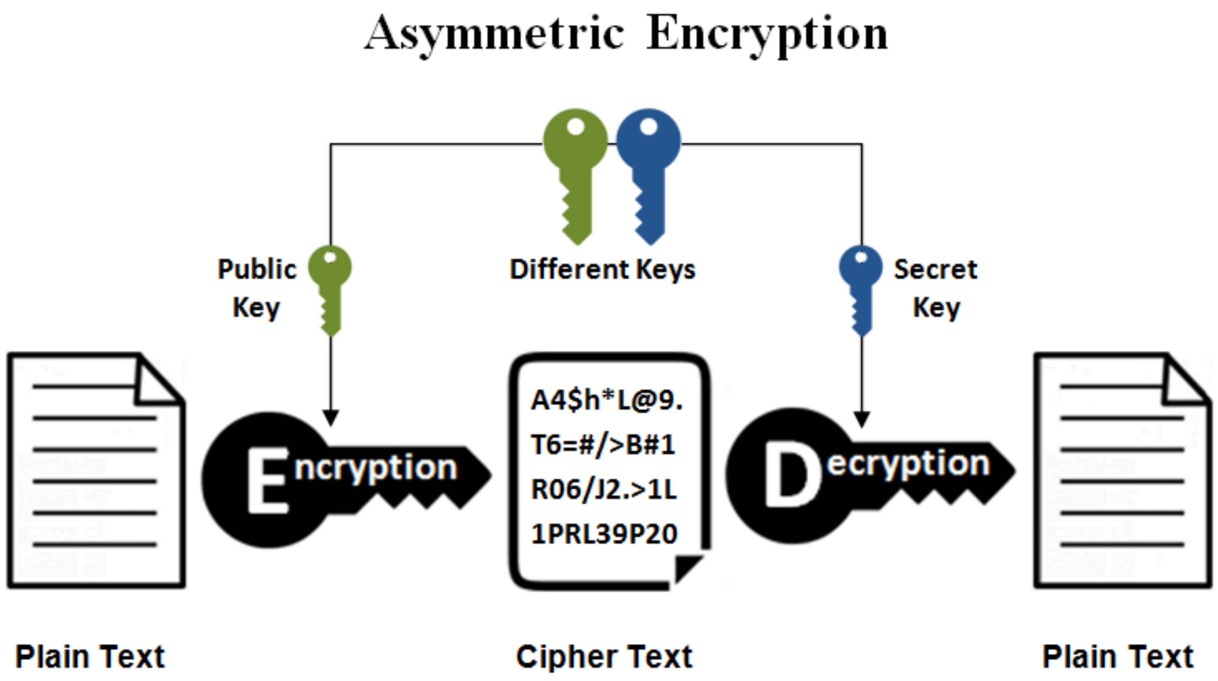
- Different key to encrypt and decrypt.
  - Step1: generate a pair of keys based on a difficult to solve math problem
  - Step2: Use one key to encrypt and a different key to decrypt
  - Note: We can also use KeyB to encrypt and KeyA to decrypt

- Normally, one key is distributed to other computers (public key). Another key is kept privately (private key).



Public/Private Key

- The **public key** is made freely available to anyone who might want to send you a message.
- The **private key** is kept a secret so that you can only know.



Key Exchange with Asymmetric Encryption

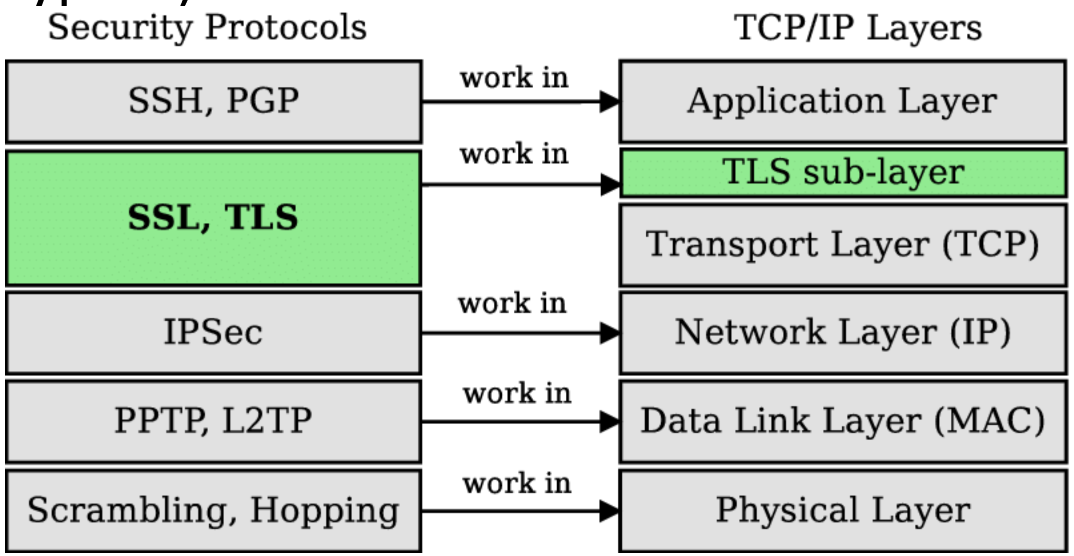
1. Client inform the server for connection
2. Server generates a new asymmetric key pairs, send the public key to the client
3. Client chooses a random symmetric key: K, encrypt it with the public key
4. Client sends the encrypted K to server over network
5. Server decrypt the message with its private key and get client's symmetric key K
6. Server and client can encrypt and decrypt message with K.

Q: Is there a need to exchange the key? Why not just use public key for encryption for all sessions?

A: There is a need for key exchange as the asymmetric en/decryption is much slower than the symmetric one.

TLS: Secure network protocol

- Encrypt/Decrypt the communication with Socket:
    - Asymmetric encryption for key exchange
    - Use symmetric encryption for later communication.
- Https: Use TLS to encrypt/decrypt the communication (via http protocol) between you and the website
    - http://microsoft.com (All the commutation is not encrypted)
    - https://microsoft.com (All the commutation is encrypted)



- Future of the Internet:
    - 33.2% of Alexa top 1,000,000 websites use HTTPS as default
    - 57.1% of the Internet's 137,971 most popular websites have an https version
    - 70% of page loads use HTTPS

TLS: APIs

- **Initialization:**

```
SSL_library_init(); //performs initialization of libcrypto and libssl
SSL_load_error_strings(); //loads error strings from both libcrypto and libssl
OpenSSL_add_all_algorithms();
SSL_CTX *newContext = SSL_CTX_new(TLSv1_client_method()); //one context per server
```

-

SSL_CTX:
This is the global context structure created once per program life-time.

Various options regarding certificates, algorithms etc. can be set in this object.

TLS: APIs

- **Initialization:**

```
SSL_library_init(); //performs initialization of libcrypto and libssl
SSL_load_error_strings(); //loads error strings from both libcrypto and libssl
OpenSSL_add_all_algorithms();
SSL_CTX *newContext = SSL_CTX_new(TLSv1_client_method()); //one context per server
```

- **Attach the SSL to a socket:**

```
SSL *sslClient = SSL_new(newContext);
SSL_set_fd(sslClient, socket);
SSL_connect(sslClient);
```

- **Read/Write:**

```
SSL_read(sslClient, buffer, sizeof(buffer));
SSL_write(sslClinet, buffer, sizeof(buffer));
```

SSL_new()
creates a new **SSL** structure to hold the data for a TLS/SSL connection.
The new structure inherits the settings of the underlying **context ctx**: connection method, options

SSL_set_fd()
sets the file descriptor **fd** as the **input/output** facility for the TLS/SSL (typically socket fd of connection)

SSL_connect() initiates TLS/SSL handshake

TLS: APIs

- **Initialization:**

```
SSL_library_init(); //performs initialization of libcrypto and libssl
SSL_load_error_strings(); //loads error strings from both libcrypto and libssl
OpenSSL_add_all_algorithms();
SSL_CTX *newContext = SSL_CTX_new(TLSv1_client_method()); //one context per server
```

- **Attach the SSL to a socket:**

```
SSL *sslClient = SSL_new(newContext);
SSL_set_fd(sslClient, socket);
SSL_connect(sslClient);
```

- **Read/Write:**

```
SSL_read(sslClient, buffer, sizeof(buffer));
SSL_write(sslClinet, buffer, sizeof(buffer));
```

- **Clean up:**

```
SSL_shutdown(sslClient); // shuts down TLS/SSL connection, send notice to peer.
SSL_free(sslClient);
```

TLS: Works on top of socket layer

- SSL_set_fd(sslClient, socket);
  SSL_connect(sslClient):
  → Randomly choose key K, encrypt and store into buf
  → write(socket, buf, sizeof(buf));
  …
- SSL_read(SSLClient, buffer, sizeof(buffer));
  → read(socket, buffer, sizeof(buffer));
  → decrypt content of buffer and store in buffer.
- SSL_write(SSLClinet, buffer, sizeof(buffer));
  → encrypt content of buffer and store in buffer
  → write(socket, buffer, sizeof(buffer);

TLS sample code

Note: all error handling code is omitted for brevity

```
SSL_CTX * ssl_init(void) {
        SSL_CTX * newContext = NULL;
        SSL_library_init();
        //Initialize the error message
        SSL_load_error_strings();
        OpenSSL_add_all_algorithms();
        //TLS version: v1, one context per server.
        newContext = SSL_CTX_new(TLSv1_client_method());
        return newContext;
}
SSL * attach_ssl_to_socket(int socket, SSL_CTX * context) {
        SSL *sslClient = SSL_new(context);
        SSL_set_fd(sslClient, socket);
        SSL_connect(sslClient);
        return sslClient;
}
```

TLS sample code (cont)

```
void ssl_clean_client(SSL* client) {
        SSL_shutdown(client);
        SSL_free(client);
}
```

Putting everything together: Send “hello, world!” with SSL to the server

```
char * host = “server.com”
int port = 6324
char * buffer = “hello, world!”
int main(void) {
        int socket = client_connect(host, port); // setup socket (addr, port), make socket connection
        SSL_CTX * context = ssl_init(); // newContext, init library, error string, algorithms
        SSL * ssl_client = attach_ssl_to_socket(socket, context); //SSL_new(), SSL_set_fd(),SSL_connect()
        ssl_write(ssl_client, buffer, strlen(buffer) );
        ssl_clean_client(ssl_client);
}

//implement ssl_read for reading commands from server
```