Politenico di Milano

Dipartimento Elettronica, Informazione e Bioingegneria

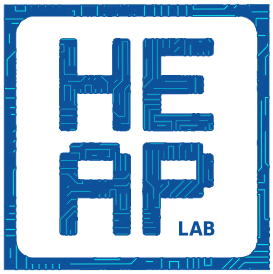
HEAPLab Project Report

LibSmoke

Author:
Kostandin Caushi
Filippo Collini

Supervisor:
Dr. Giusppe Massari
Domenico Iezzi
Michele Zanella

April 14, 2019



Abstract

Nowadays systems try to exploit parallelism to achieve better and better performances and concurrency. Moreover, the proliferation of small powerful and poor energy consuming embedded systems, led new computing paradigms explore the possibility of exploiting idle resources of these mobile devices and small objects connected through a network, by offloading to them part of the computation.

For this purpose, frameworks for distributed computation on embedded systems has been already deployed. They allow client applications, running on a General Purpose CPU-based node, to offload part of its tasks and data buffers to a remote device, where an instance of the daemon is running, to perform operations like : reading and writing buffers, waiting for specific events and running computations.

We will present a library, called LibSmoke, that allows embedded systems to communicate between each others in an efficient and secure way using in particular SKNX (Securing KNX), to create the network between devices, and AES-256, to encrypt and decrypt the packets exchanged.

1 Introduction

The goal of our project was to make the the communication between distributed embedded systems more secure. In order to do that, we use the *SKNX* protocol with *Multi-party Key Agreement Scheme* and *AES256_CTR* encryption algorithm.

1.1 SKNX - Securing Konnex protocol

It is composed of these subsystems:

- **Backend** which handles the low-level side of a bus like initializing the communication channel, broadcasting data or reading it.
- **Packetizer** which wraps long packets inside small KNX telegrams, having 15 bytes of payload each.
- **Connection Logger** which is a subsystem cooperating with the Packetizer. Its role is to keep track of active connections and store all small chunks of packets so that the packetizer can recompose them.
- **Key Exchanger** which exchanges crypto keys (as the name says).
- **Node Counters** whose role is to count and collect IDs of nodes that are connected on the bus and communicate with the core when they are ready, so that the key exchange could start.
- **Logging** which logs messages and infos on the standard output.
- **Timing** which collects timing information.

And, of course, there is the SKNX core which links all subsystems together. It is mostly composed by a finite state machine (FSA). At first it should be initialized, then it has to be updated in the software main loop.

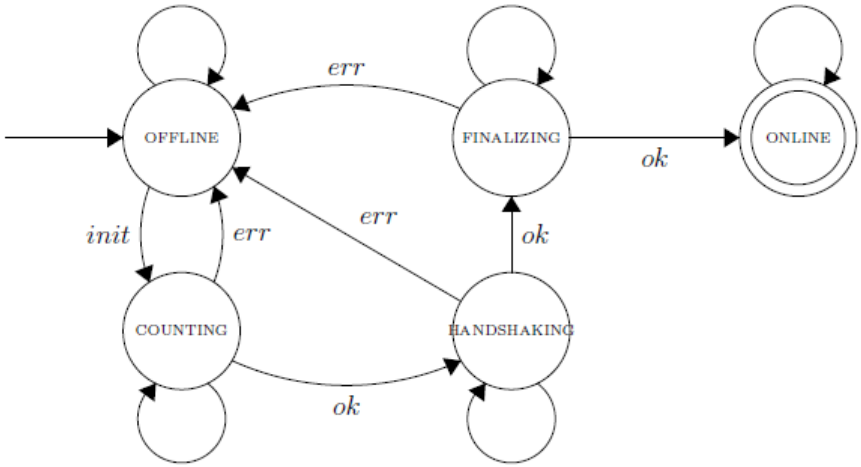


Figure 1: SKNX Core FSA

1.2 AES-256

AES is short for Advanced Encryption Standard. It's a symmetric block cipher used by the American government to encrypt sensitive data.

AES is built from three block ciphers: AES-128, AES-192 and AES-256 which encrypts data in chucks of 128 bits using cryptographic keys of 128, 192, 256 bits. All symmetric encryption ciphers use the same key for encrypting and decrypting data, which means the sender and the receiver must both have the same key.

The first step in the AES encryption process is substituting the information using a substitution table, the second transmutation changes data rows and the third shifts columns. The last transformation is a basic exclusive XOR process done on each column using a different part of the encryption key. The longer the encryption key, the more of these rounds of processing steps are needed (for AES-256, which is our choice, 14 rounds are needed).

AES can have different modes of operation, for our purposes we choose the Counter Mode (CTR). Counter mode turns a block cipher into a stream cipher. It generates the next key-stream block by encrypting successive values of a "counter". It can ensure us software efficiency (pipe-lining), hardware efficiency (it's fully parallelizable), quickness (thanks to the random access), provable security and simplicity since the decryption circuit is the same of the encryption circuit.

2 Design and Implementation

LibSmoke is composed essentially, as told before, by SKNX and AES.

Secure KNX - Allows the user to create the *backend* connection between different devices, in particular in this paper/project we have considered the LinuxTCP one. It includes also a *pktwrapper* that handles the packets exchange : splitting data in pkts of 15 bytes payload and recomposing next them together. There are also available multiple KeyExchange algorithms that allows to calculate a shared key; we have here test the *mka* (Multi-party Key Agreement Algorithm).

AES256_CTR - Given that *mka* returns a key of 32 bytes we choose AES256, but in the *AES Library* there are also available AES128 and AES192. It includes all the implementations of AES ECB, CTR and CBC, in particular we use CTR cause it also provides padding.

Given this background we decide to split LibSmoke in 2 classes : ClientSmoke and ServerSmoke.

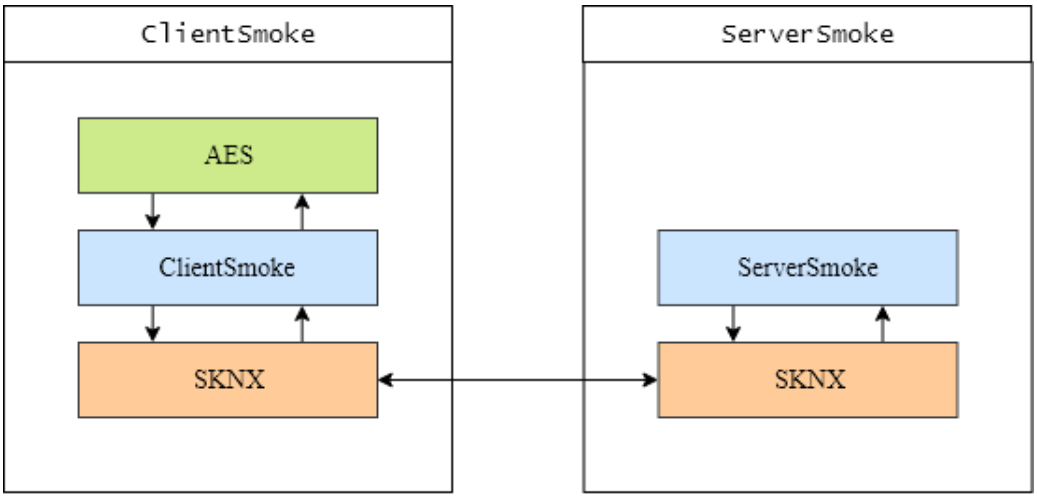


Figure 2: LibSmoke Architecture

2.1 ClientSmoke

Initialized by devices that needs to send and receive pkts, given also the functionality to encrypt and decrypt data with the encryption algorithm [AES256_CTR].

Listing 1: ClientSmoke

```
/**
* Class of Libsmoke Client.
*
* @param KeyAlgorithm - is the KeyExchange Algorithm
*                       needed for SKNX to generate the key.
* @param PORT - is the PORT used by sockets.
* @param numClients - is the #Clients connected.
*/
template<typename KeyAlgorithm, uint16_t PORT,
                       size_t numClients>
class ClientSmoke {
public:
/**
* Constructor of Libsmoke Client.
*
* @param addr const char* - is a pointer to the IP
*                       to which socket client has to connect.
*/
ClientSmoke(const char *addr);

/**
* Initializes _backend and sknx.
* When sknx is ONLINE retrieves the KeyAlgorithm.
*
* \return     TRUE - if all initializations are completed
*                       succesfully and key is retrieved.
*             FALSE - if there's an error on initialization
*                       or key retrieving.
*/
bool init();
```

Listing 2: ClientSmoke

```
/**
* Uses pktwrapper in order to get the oldest pkt received
*                        from other clients.
* It decrypts the body using AES_CTR with the shared key
*                        of the client.
*
* @param data KNX::pkt_t& - pointer to a pkt that the method
*                        uses to pass the oldest recieved one.
* @return TRUE - if there is actually a recieved pkt in the
*                        queue of pktwrapper.
*         FALSE - if no pkt is received.
*/
bool receive(KNX::pkt_t &data);

/**
* Uses pktwrapper in order to send data (buf), with a specific
*       length (len) and command (cmd),
*       to a specific destination (dest).
* It encrypts the body using AES_CTR with the shared key
*       of the client.
*
* @param dest - the destination of the data.
* @param cmd - the command to send to the client(s).
* @param buf - the data, if there are, to send.
* @param len - the length of the data (can be also 0).
*/
void send(uint16_t dest, uint8_t cmd, uint8_t *buf,
uint16_t len);
```

NB : the user can choose the KeyExchange Algorithm from the ones already implemented and available inside the SKNX library.

2.2 ServerSmoke

It's main features are to instantiate socket connections and broadcast the received pkts from the clients.

Listing 3: ServerSmoke

```
class ServerSmoke{
public:
/**
* Constructor of Libsmoke Server.
* Sets _ready to false in order to say that the server
*       is not yet initialized.
*/
ServerSmoke() : _ready(false);

/**
* Checks if the server is already initialized.
* If not, creates the socket with the given IP and PORT.
*
* @param addr - IP addr of the socket to create.
* @param port - PORT of the socket to create.
* @return TRUE - if socket is created with no errors
*       and is listening.
*         FALSE - otherwise.
*/
bool init(const char *addr, int port);

/**
* Handles connections : registering the new ones
*       or deleting the closed ones.
* Also used to broadcast messages to all connected clients.
*/
void run();

/**
* Destructor of Libsmoke Server.
* Used to close all sockets and clear the vector
*       of the registered clients.
* Sets also _ready to false, so the server
*       can be initialized again.
*/
~ServerSmoke();
```

2.3 Instructions to Use

Git repo : https://github.com/KostandinCaushi/LibSmoke.

There you can find this paper ("LibSmoke Documentation") and the "Project" folder. Inside it there is the "libsmoke" directory, download and add it to your project.

The available API are the one reported above and in order to use them you need to :

- ClientSmoke : '*#include "libsmoke_client.h"*'
- ServerSmoke : '*#include "libsmoke_server.h"*'

NB : in the repository is also available a 'tests' directory, where inside can find 'client_test' and 'server_test', that show how to use the library.

3 Experimental Results

To explain the usage of the library we build test files for both client and server.

3.1 Server_test.cpp

This class builds an instance of the ServerSmoke and tries to initialize it. Then if the initialization goes well, the server can now be run. It is now able to handle tcp connections from other devices, can register and delete connection and it's used to broadcast messages over the network until the procedure of shutdown ends it.

An interesting aspect of the run() method is that it's running in a loop until its shutdown, always listening if there will be modifications in the network, i.e. a new client comes or another one disconnects.

If the client succeeds in connecting to the socket it's pushed into the clients vector. It will now receive all the broadcast messages in the network. When it will disconnect it will be popped out from the vector and removed from the network.

Listing 4: ServerSmoke

```
...

// Handle new connections
if(FD_ISSET(_sock, &_rfds)) {
        Client *c = new Client();

        socklen_t len = sizeof(c->in);
        c->sock = accept(_sock, (struct sockaddr *) &(c->in),
                &len);

        if(c->sock < 0) {
                fprintf(stderr,
                "[ERROR] Cannot accept a new connection\n");
                delete c;
        }

        printf("[NEWCONN] Say welcome to %s:%d\n",
        inet_ntoa(c->in.sin_addr), c->in.sin_port);

        clients.push_back(c);
}
```

```
...

// Remove disconnected clients
for(size_t i = 0; i < clients.size(); i++) {
        Client *c = clients[i];
        if(c->mustDelete) {
                printf("[DISCONN] Say goodbye to %s:%d\n",
                inet_ntoa(c->in.sin_addr), c->in.sin_port);
                if(c->sock > 0) close(c->sock);
                clients.erase(clients.begin() + i--);
                delete c;
        }
}

...
```

3.2 Client_test.cpp

This class first of all builds a ClientSmoke. In order to do that it's fundamental to specify the key exchange algorithm, the ip and the port to which we want to connect and also the total number of clients that will form up the network.

In the init() method the client will try to connect to the server and it will participate at the construction of the network. When the predefined number of clients will be connected to the socket, the chosen key exchange algorithm will start, giving the same key to all the clients participating, so building up a way to make them communicate safely and making the network secure.

To implement the send method, in order to be able to send packets also bigger than 16 Bytes, we decided to use the write function of _packetwrapper that will automatically build the right packets. At this point we'll only need to encrypt, using AES256 with the shared key, the body of the message, before to use it as a parameter in the send function.

As last step we call the update method of _packetwrapper to start the process of sending and receiving the pending packets. In both send and receive methods, to encrypt/decrypt the messages, we need to initialize AES.

Listing 5: ClientSmoke

```
...

void send(uint16_t dest, uint8_t cmd, uint8_t *buf,
uint16_t len) {
        printf("\nSent MSG\n");
        Debug::printArray(buf, len);

        // Encrypt MSG
        AES_init_ctx_iv(&_ctx, _key.key(), iv);
        AES_CTR_xcrypt_buffer(&_ctx, buf, len);
        printf("\nEncrypted MSG\n");
        Debug::printArray(buf, len);

        // Send MSG
        _pktwrapper.write(dest, cmd, buf, len);
        _pktwrapper.update();
}

...
```

3.3 Test Results

Here we report some results of the test we run.

We made 1 executable of the *Server_test* and 2 of the same *Client_test* (this means that both clients will send to each other the same msg).

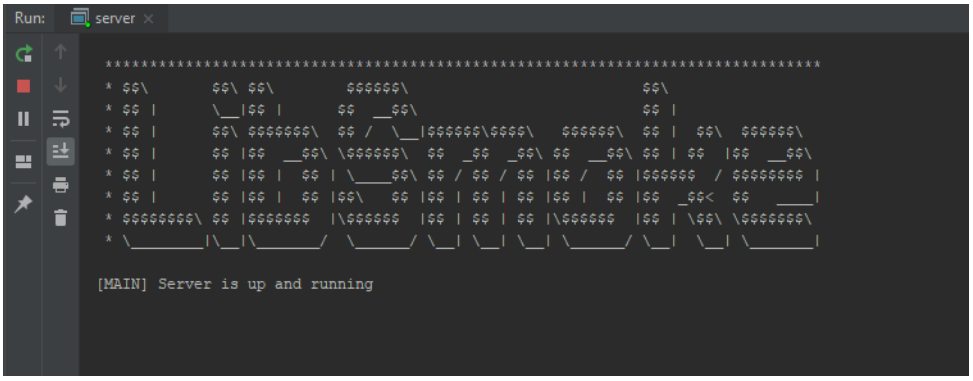


Figure 3: Server Initialization Test

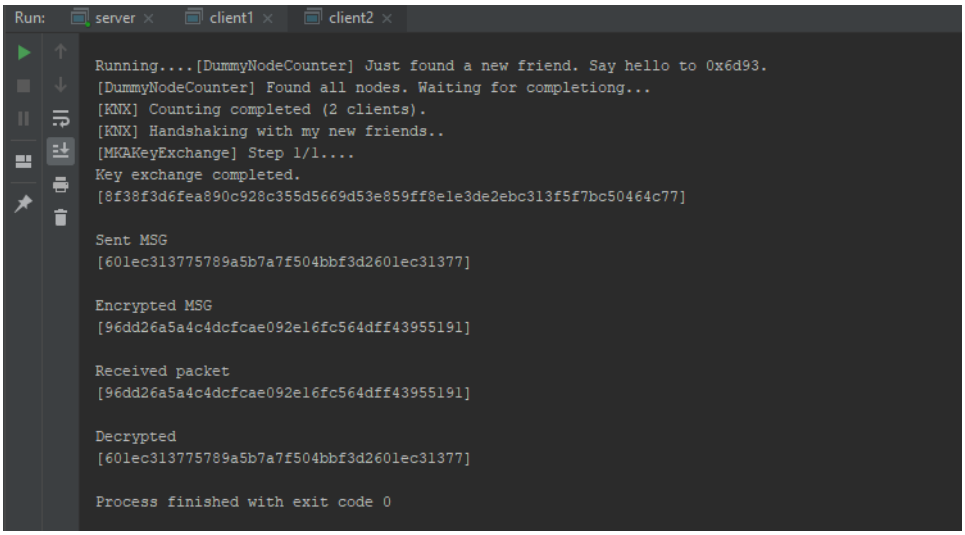


Figure 4: Client Test

4 Conclusions

In this project we developed a library which can guarantee the encryption of the communication in a network of devices, in our case in a distributed computation framework.

At the beginning we had client applications connected to running instances of the server daemon without any kind of security and authentication, using a plain TCP connection.

Now a fixed number of entities, unknown to each other, can establish a shared secret key over a public, insecure channel and then communicate between each other, through the broadcast server, through encrypted messages.

5 Future Work

The system can be improved in many different ways.

1. The broadcast communication could become a point to point communication to send the packet directly to the designed destination.

2. The algorithm for the key exchange has the limit of a fixed number of clients that can participate that must be known at the beginning.

3. The initialization vector used in AES encryption/decryption is constant for testing purposes and also because our system has not a secure channel where the exchange of the vector can be done. This can be a path to follow to improve the security of the library.

4. As last point, may be taken into account the possibility to be more flexible with the dimension of the packets to be sent and received.

6 References

1. Beer Paper
2. SKNX Paper
3. AES Paper