Digital Bond – Circuit Approach

Final Report

Presented by:

Slava Ustinov 309930006

Ron Meiry 301224283

Contents

Introduction

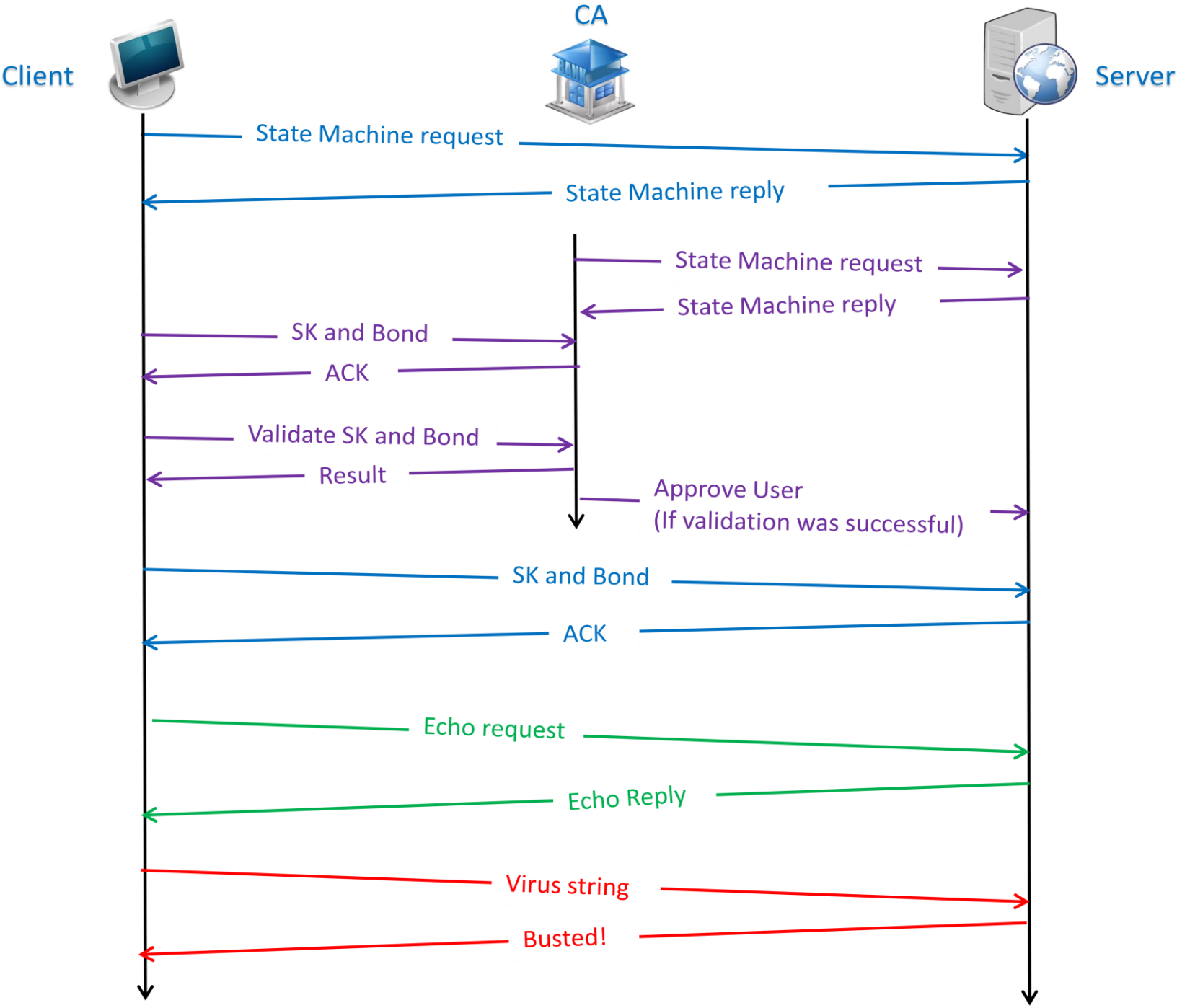
Please note that this document assumes that the reader has read the preliminary report and is familiar with the cryptographic scheme this project implements.

Cryptographic Scheme Overview

The project 's cryptographic scheme is based on the article:
Functional Encryption for Regular Languages, Brent Waters, University of Texas at Austin.
Our implementation follows the Algorithms (3.1) section as described in the article, with the following changes:

1. We define a maximal length for a message between a client and the server (`MAX_MSG_LENGTH`). Messages longer than `MAX_MSG_LENGTH` will be shortened to the initial `MAX_MSG_LENGTH` bytes. Messages shorter than `MAX_MSG_LENGTH` will be padded with '0's until they contain a total of `MAX_MSG_LENGTH` bytes.
2. Regarding Encrypt $(PP, w = (w_1, \dots, w_l), m)$:
    a. Our implementation generates a partial CT and not a normal CT.
    b. The secret the client wishes to protect is the Bond element (C_m).
    c. The scheme is built in a way that any w that is accepted by the state machine is sufficient to complete the partial encryption (of course there a need for the Secret Key also) and decrypt the entire message, revealing the Bond element.
    d. l is always taken as the value of `MAX_MSG_LENGTH`.
    e. $C_{i,2}$ is calculated for every possible h_{w_i} (There'll be ALPHABET_SIZE $C_{i,2}$ values).
    f. w is the client's secret also.
3. Regarding Decrypt (SK,CT):
    a. When the server identifies a virus (a specific w), It must choose the $C_{i,2}$ elements (from the partial CT) that correspond with the specific w, transform the partial CT to a normal CT and decrypt the message.
4. Regarding the State Machine:
    Since the message strings are padded, the state machine must be designed such that once it has reached an acceptance state, it stays in an acceptance state (so that the padding won't affect the virus-check result for the given string)

Communication Scheme



Machine Types

The program can be run as one of 3 machines, depending on the user-given arguments:

Client:

Wishes to receive services from the server.
Follows the following steps:

1. Requests a state machine from the server.
2. Generates a Secret Key (SK) and a Bond (a partially encrypted secret plaintext).
3. Send the SK and the Bond to the CA for validation.
4. Upon receiving a validation approval from the CA, sends the SK and Bond to the server.
5. Initiates the operational stage: sends messages to the server and receives replies.
6. At some point, tries to attack the server.

Server:

Wishes to provide services to clients.

1. Initializes a state machine.
2. Upon reception of a request for a state machine, sends a copy of the state machine.
3. Upon reception of a user validation message from the CA, updates the user database.
4. Upon reception of an SK and Bond from a client, updates the user database if that user was approved by the CA. Marks that user as operational.
5. Upon reception of a legal message from an operational user, echoes the message back to the user.
6. Upon reception of an illegal (virus) message from a user, completes that user's Bond encryption, decrypts the Bond and notifies the user.

Certificate Authority (CA)

Provides validation services for clients that wish to use the server's services.

1. Requests a state machine from the server.
2. Receives SK and Bond from potential clients.
3. Upon client request, validates the client's SK and Bond.
4. If the validation was successful, gives the server a 'green light' to provide services to the client.

Implementation Notes

1. Our implementation relies on the PBC (Pairing Bases Cryptography) open-source library to provide us with methods for performing calculations over finite groups.
2. The Client UI is written in Python, while the rest of the project is written in C++.
3. We've put an emphasis on correct OO design to allow maximum modularity for future enhancements.
4. The code is highly documented and browsing through the comments will provide a detailed insight into our thought process and logic (hopefully…).
5. We recommend a full theoretical understanding of the cryptographic scheme before trying to understand our implementation.

Code structure overview

Eng_final_proj.cpp:

Entry point: main()
Switches between the 3 different machine types (Client,Server,CA), according to the user given parameters.

State Machine related:

Transition3Tuple.cpp/h:

Defines a triple that represents a single transition:
<Current state, Next state, Input>.

State.cpp/h:

Defines a single state in a state machine.
Uses the Transition3Tuple class to represent all available transition from this state.

StateMachine .cpp/h:

Defines a finite state machine.
Uses the State class to represent all available states in the given machine.

Encryption related:

BilinearMappingHandler.cpp/h

A convenient wrapper for all required <pbc.h> functionality.
Allows convenient access to the all required mathematical and I/O methods.

EncryptionHandler .cpp/h

1. Uses a BilinearMappingHandler to perform more complex cryptography-related operation, such as:

setup, createPartialEncryption , completePartialEncryption , decrypt

2. Contains the following nested classes:
    a. SK - Represents and Secret Key, as described in the article.
    b. MSK - Master Key, as described in the article.
    c. CT - Represent a Cipher Text, as described in the article.

Communication related:

SocketWrapper.cpp/h

A wrapper that provides convenient socket creation the usage.

WelcomeSocket.cpp/h

Defines a convenient wrapper for a welcome-socket (i.e. a web server's socket that listens on port #80)

Object serialization:

We've chosen Google's Protocol Buffers mechanism to provide us with a comfortable way of serializing complex objects (like a state machine, secret key and cipher text) to strings that can be easily sent over a socket.
The mechanism required us to write a .proto file (which must contain a description of the desirable classes in a special format) which was later compiled with Google's protobuf application and resulted in a C++ file, containing several classes.
The protobuf compiler must be installed if you wish to change and recompile the .proto file.
The generated classes provide convenient methods for setting and converting our complex objects into strings and the other way around.

StateMachineAndKey.pb.cc/h

This class is the result of a compilation by protobuf over our .proto file.

StateMachineAndKey.proto

Describes all the objects we wished to serialize in our project, according to the compiler's expected format.

ObjectSerializer.cpp/h

Uses StateMachineAndKey.
Responsible for serializing and de-serializing the StateMachine, SecretKey and CipherText objects so they can be easily sent and received via socket.

Compilation cmd:

- protoc -I=$SRC_DIR --cpp_out=$DST_DIR $SRC_DIR/addressbook.proto
- In our private case: protoc -I=. --cpp_out=./src/ ./StateMachineAndKey.proto

Machine level:

BasicMultithreadedServer.cpp/h

An abstract class that defines a basic multi-threaded server.
The server listens on a WelcomeSocket and dispatches worker threads to handle established connections.
execOnWorkerThread() must be overridden and this method is the entry point for a worker thread.

CA_Machine.cpp/h

Inherits: BasicMultithreadedServer
Contains the entire CA functionality:
- Gets a state machine from the server.
- Receives SK and Bond form clients.
- Verifies SKs and Bonds on client request.
- Notifies the server in cases of successful verifications.

The entry point is run().

ServerMachine.cpp/h

Inherits: BasicMultithreadedServer
Contains the entire Sever functionality:
- Generates a state machine.
- Sends a copy of the state machine on request.
- Receives notifications about users from the CA.
- Receives SK and Bond from verified users.
- Provides services to legitimate users.

The entry point is run().

Client_UI_Server.cpp/h

Inherits: BasicMultithreadedServer
Handles all communication with the Client UI program.
Receives requests via socket and performs the requested operation.
Used within a ClientMachine.

ClientMachine.cpp/h

Contains the entire Client functionality:
Runs a Client_UI_Server to handle all communication with the Client UI.
Contains methods to be used by the Client_UI_Server according to the user's wishes.

Misc.

Constants.h
Contains all program-wide parameters.

General_Func.h
Contains all program-wide functions.

Messages.h
Contains all message associated parameters and definition.
Contains a detailed message-format explanation.

StateMachineAndKey.proto
Describes all the objects we wished to serialize in our project, according to the compiler's expected format.

3rd party libraries

1. Pairing Bases Cryptography (PBC)
2. Google's Protocol Buffers: https://developers.google.com/protocol-buffers/

Running the program