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.

Nowadays, cyber-attacks and many types of digital fraud are almost always risk free. Sitting in the comfort and safety of his home, an attacker can mount one attack after the other, often protected from identification by the virtual anonymity of the Internet and from legal proceedings by being in a different jurisdiction or even different country from the target.
The project suggests a new paradigm for deterring remote IT attackers, based on [1], with the following change: the bond encryption and decryption is done based on a Deterministic Finite Automata Functional Encryption system [2]. The basic idea is to add a dimension of liability to the interaction between a client and an IT service.

Scheme overview:

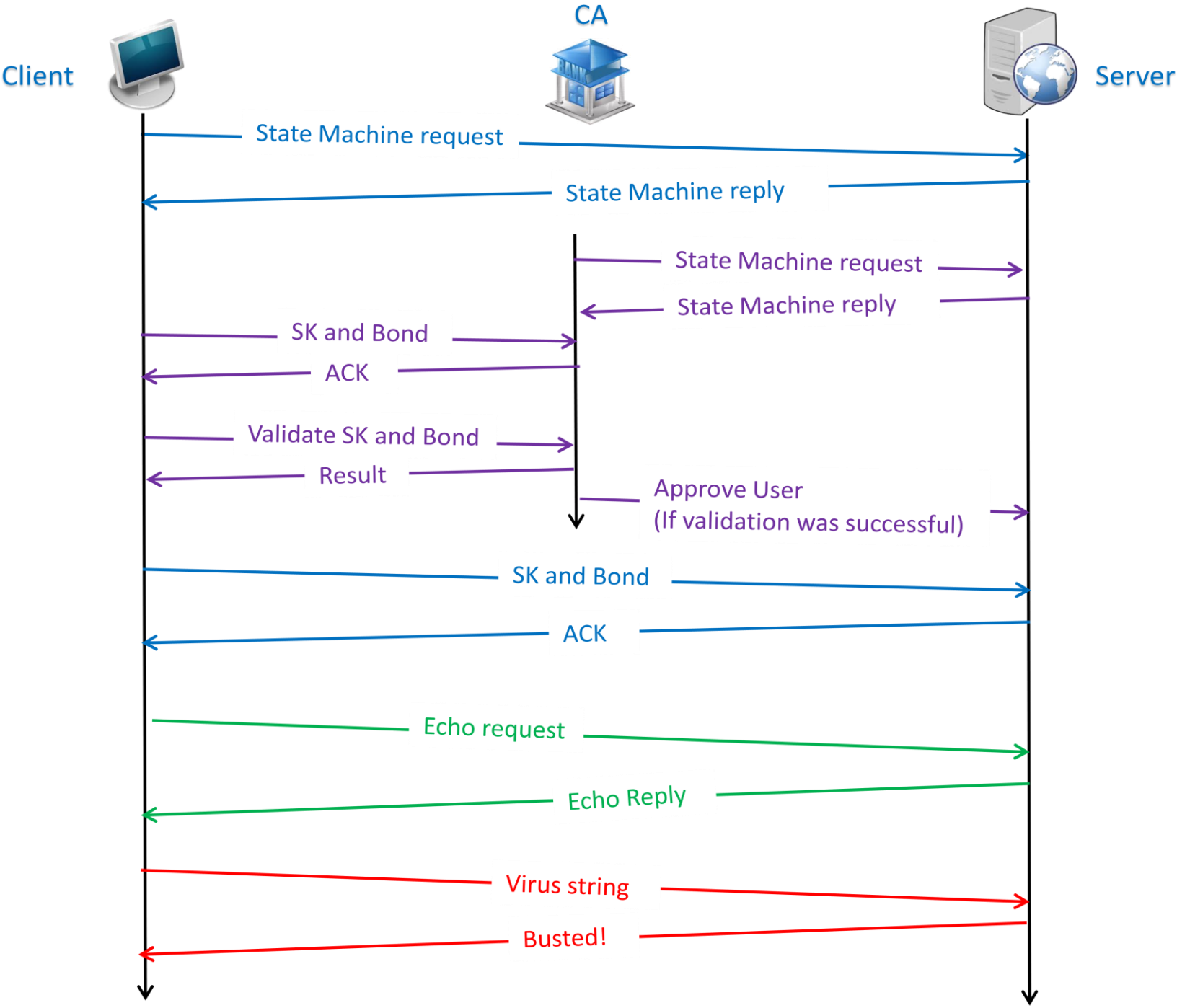
- *Setup stage:*
  - The client and the server reach an agreement that defines boundaries to the client's behavior, meaning demanding no cyber-attacks or fraud attempts against the server.
  - The client must provide the server with a digital bond, e.g. a cashier's check, which the server can cash if the user violates the policy.
  - A 3rd party is required to vouch for the validity of the client's digital bond.
  - The client sends an encrypted bond to the server.
- *Operational stage:*
  - At this point the operational stage begins. The client and the server communicate freely.
- *Bond Recovery stage:*
  - In case the server identifies an illegal message, the attacker's offensive message is used to recover the attacker's bond.

Cryptographic Scheme Overview

The project's cryptographic scheme is based on the articles [1] and [2]:
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).
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_Funcs.h
Contains all program-wide functions.

ParamFileHandler.cpp/h
Contains methods for reading and parsing the global_param_file.

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.

Parameter File

Under the **'param'** folder you'll find the **'global_param_file'**.
This file contains two major sections:

1. **Global parameters to run the program according to:**
   a. max_message_len - according to *Functional Encryption for Regular Languages,* this is l,
   meaning the maximal length of a message between the client and the server in the operational stage. Note that this value has a direct influence on the encryption creation time (large message size means a long encryption process).
   b. num_of_states_in_SM - The total number of states in the programs state machine (which should be described later on in the param file).
   c. virus_string - A single string that is accepted by the state machine.
2. **State machine description:**
   A state machine is a collection of states, when every state should be described according to the following pattern:

```
*SM Start*
State [state number] [isAcceptance]
[input value] [next state number]
[input value] [next state number]
[input value] [next state number]
[input value] [next state number]
.
.
.

*SM End*
```

Note that unmentioned input possibilities will be treated as self-loops by default (meaning these inputs will keep the machine in it's current state).

An example of a valid **'global_param_file'**:

```
# comments are ignored
# Global params:
max_message_len = 15
num_of_states_in_SM = 6
virus_string = virus

# State machine description:
# An SM is a collection of states, when every state should be described according to the following pattern:
#
#  *SM Start*
#  State [state number] [isAcceptance]
#  [input value] [next state number]
#  [input value] [next state number]
#  [input value] [next state number]
#  [input value] [next state number]
#  .
#  .
#  .
#
#  *SM End*
#
# Note that unmentioned input possibilities will be treated as self-loops by default
#(meaning these inputs will keep the machine in it's current state).

*SM Start*
State 0 false
v 1
State 1 false
i 2
State 2 false
r 3
State 3 false
u 4
State 4 false
s 5
State 5 true
*SM End*
```

3rd party libraries

1. Pairing Based Cryptography (PBC) - Provides all finite-group related methods.
2. Google's Protocol Buffers: https://developers.google.com/protocol-buffers/
   Provides a compiler that generates C++ classes that allow convenient serialization of complex objects.
3. tkinter (python package) - GUI related.
4. ttk (python package) - GUI related.
5. easygui (python package) - GUI related.

Compiling the program

In case you'll wish to make changes to the source code and recompile the program, the following steps must be taken:

1. Make sure you have a C++ compiler installed (i.e. g++ for linux)
2. Make sure you have a Python 3.3.1 interpreter installed (other Python version may not work)
3. Install all the required 3rd party libraries that are described in the previous page.
4. The project is saved as an Eclipse project and can easily be added to it.
5. Meaningful files and folders:
    a. Src - Contains the C++ source code part of this project.
    b. gui.py - Contains the Python source code for the GUI
    c. Param - contains all required parameter files to init the program.

Running the program

1. The following folder and file must exist in the directory the program is launched from (note that this is not necessarily the path of the program file, but rather the pwd - Present Working Directory):
   a. 'param' folder - must contain the text files that are provided with the PBC library and the 'global_param_file' (which was described previously).
   b. gui.py file - will be run if the program is run as a client.

2. The program must be provided with parameters when run via the command line. Running the program without any parameters will result in a help screen splash that describes the expected parameters:

```
==============================================================================
                    **************************
                    **Welcome to CryptoBond!**
                    **************************
-----------------------------------------------------------------------
    This is the program instruction:
-------------------------------------------------------------
     To run as Client:
            client [Server IP]:[Server PORT] [CA IP]:[CA PORT] [Name]
    Example: client 10.0.0.1:12345 10.0.0.2:12346 Mr.User
    -> Please note that the maximum user-name length is 30 chars!
    -------------------------------------------------------
     To run as Server:
            server [Self port] [CA IP]:[CA PORT]
    Example: server 8000 10.0.0.2:12346
    -------------------------------------------------------
     To run as CA:
            ca [Self port] [Server IP]:[Server PORT]
    Example: ca 8000 10.0.0.1:12345
    -------------------------------------------------------
==============================================================================
```

Detailed explanation:

1. To run the program as a client, the program should be run according to the following pattern: client [Server IP]:[Server PORT] [CA IP]:[CA PORT] [Name]
   When:
   a. Server IP - The server's IP address
   b. Server PORT - The server's port number
   c. CA IP - The CA's IP address
   d. CA PORT - The CA's port number
   e. Name - the client's user-name. Could be anything.

For example, if the program file's name is 'crypto_bond' the following line can be executed via terminal: **./crypto_bond client 10.0.0.1:8001 10.0.0.2:8002 Mr. User**

2. To run the program as a server, the program should be run according to the following pattern: server [Self port] [CA IP]:[CA PORT]
   When:
   a. Self port - The port on which the server listens for incoming requests.
   b. CA IP - The CA's IP address
   c. CA PORT - The CA's port number

For example, if the program file's name is 'crypto_bond' the following line can be executed via terminal: **./crypto_bond server 8001 10.0.0.2:8002**

3. To run the program as a CA, the program should be run according to the following pattern: ca [Self port] [Server IP]:[Server PORT]
   When:
   a. Self port - The port on which the CA listens for incoming requests.
   b. Server IP - The server's IP address
   c. Server PORT - The server's port number

For example, if the program file's name is 'crypto_bond' the following line can be executed via terminal: **./crypto_bond ca 8002 10.0.0.1:8001**

Future Enhancements

Based on our work, we've thought on several feature that can be added to this project, thus making it's cryptographic scheme more robust and suitable for commercial use:

1. At the client's side, the encryption process might take a long time when dealing with long words (when l is big enough), since we're performing $l * Alphabet_size$calculations. The nature of these calculations is such that the encryption process can be parallelized fairly simply. A good idea might be to distribute the calculation across different CPUs if possible.

2. The server must be able to cash a check only if it possesses a virus string signed by the client. Meaning, the state machine should handle only client-signed inputs.
   This'll prevent the server from abusing the scheme by generating random strings, feeding them into the machine and hoping to decrypt the client's bond.

3. The communication between the server and the CA should be established with SSL, thus providing authentication and encryption for both parties.

4. Since nothing prevents the client from sending a valid bond to the CA and then an invalid bond to the server, we suggest that the CA should send a hash of the bond it received from the client to the server, In this way, the server can perform the same hash over the bond it received from the client and be assured, with high probability, that he has received the same valid bond that was sent to the CA.

Bibliography

1. Deterring Attacks and Fraud in the Digital World
   Asaf Cohen, Shlomi Dolev and Niv Gilboa
   Ben-Gurion University of the Negev, Beer-Sheva, Israel

2. Functional Encryption for Regular Languages
   Brent Waters
   University of Texas at Austin