How to Design APIs for Cryptographic Protocols

Lior Malka

Crypto Group, University of Maryland, USA

What is a Cryptographic Protocol?

- Usually two parties.
- Terminology: sender - receiver / server – client.
- Parties has **input** and **output** (possibly `null`).
- Threat model.
- Security parameters.

Bob

Alice

API – Application Programming Interface

A **software library** is a collection of modules that provides services to independent programs via an **API** *(Abstract Programming Interface)*

Example: Class Encryption uses Class Vector.

Class Vector provides services to Class Encryption.

Class Encryption provides services to other classes.

The methods and variables through which a class provides services are the API of this class.

Software Design Without API

Pros

- Fast to implement in small projects.
- Agile – can serve as a starting point for API design.
- No need to consider how code interfaces with other software.
- Can be appropriate for small “dead end” projects.

Cons

- Inappropriate for large projects.
- Code has a limited (as opposed to general) functionality.
- Code is not reusable.
- Code is hard to maintain/modify.
- Prone to errors and bugs.

Why a Good API is hard to Design

- Forces designer to anticipate future usage of code.
- Requirements are incomplete (may never be complete).
- Requires abstraction.
- Requires modularization.
- Requires skills in programming languages.
- Requires code rewrites – time consuming and labor intensive.

The Benefits of API Driven Design

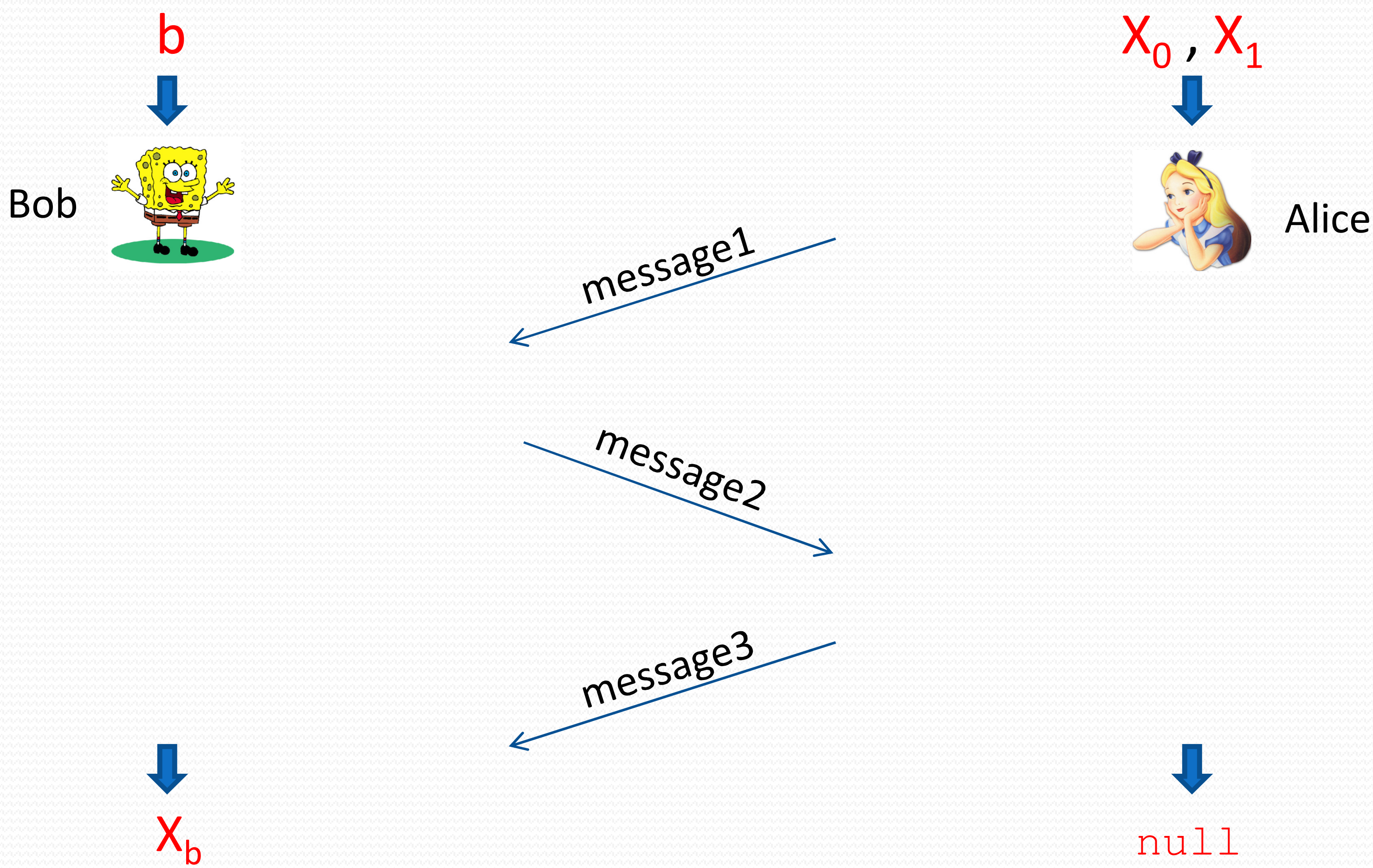
When an API is used in a project, it

- Allows to focus on the project.
- Saves development time.
- Reduces errors and debugging.
- Facilitates modular design.
- Provides a consistent development platform.

Case Study

Oblivious Transfer (OT)

Oblivious Transfer (OT) - Visually



Oblivious Transfer (OT) - Formally

In an *oblivious transfer protocol*, Alice allows Bob to learn only one of her inputs, but Bob does not tell Alice which one it is. Formally:

- The input of Alice is two strings: X_0 and X_1.
- The input of Bob is a bit b (0 or 1).

At the end of the protocol:

- The output of Bob is X_b.
- The output of Alice is `null`.

Security properties:

Alice does not learn b, and Bob does not learn X_{1-b}.

Implementing OT : 1st Attempt

```
class Alice {
            String X0,X1;
            encryption_key_length;

            main(){
              // open a socket and wait to hear from Bob..
              :
              send_first_message();
            }

            // methods for sending and receiving messages
            void send(byte[] m) { … }
            byte[] receive() { … }

            /* methods for constructing the messages of Alice
            and processing the replies of Bob */
            void send_first_message(){
              m1 = new byte[encryption_key_length];
                         :
              send(m1);
              receive_second_message();
            }
            void receive_second_message(){
              byte[] m2 = receive();
            :
            }
}
```

What is Wrong With This Code?

- Mixes several unrelated functions.
- Has no API.

<u>Exercise:</u>

- *List all the flaws in the OT protocol.*
- *How would you modify the code so that it has an API?*

<u>Next slides:</u>

- Improve design and provide an API.
- We will only consider `class Alice`; the same improvements apply to `class Bob`.

Design Flaw: mixed functionality

```
class Alice {
            String X0,X1;
            encryption_key_length;

            main(){
              // open a socket and wait to hear from Bob..
              :
              send_first_message();
            }

            // methods for sending and receiving messages
            void send(byte[] m) { … }
            byte[] receive() { … }

            /* methods for constructing the messages of Alice
            and processing the replies of Bob */
            void send_first_message(){
              m1 = new byte[encryption_key_length];
                        :
              send(m1);
              receive_second_message();
            }
            void receive_second_message(){
              byte[] m2 = receive();
            :
            }
}
```

Networking functions have nothing to do with OT. They should be in a separate class called "Server"

Redesign - Phase 1

First Objective

Separate networking functionality from the protocol.

Writing The Client and The Server

```
class Client {

            Socket socket;

            // methods for sending and receiving messages
            void send(byte[] m) { … }
            byte[] receive() { … }
}


 class Server{

            ServerSocket socket;

            // methods for sending and receiving messages
            void send(byte[] m) { … }
            byte[] receive() { … }
 }
```

Removing `send & receive` from `class Alice`

```
class Alice {
            String X0,X1;
            Server server;
            encryption_key_length;

            main(){
              Alice alice = new Alice();
              alice.server = new Server(..);
              send_first_message();
            }

            // methods for sending and receiving messages
            void send(byte[] m) { … }
            byte[] receive() { … }

            /* methods for constructing the messages of Alice
            and processing the replies of Bob */
            void send_first_message(){
              m1 = new byte[encryption_key_length];
                        :
              server.send(m1);
              receive_second_message();
            }
            void receive_second_message(){
              byte[] m2 = server.receive();
            :
            }
}
```

Implementing OT : 2nd Attempt

```
class Alice {
            String X0,X1;
            Server server;
            encryption_key_length;

            main(){
              :
            }

            /* methods for constructing the messages of Alice
            and processing the replies of Bob */
            void send_first_message(){
              m1 = new byte[encryption_key_length];
                        :
              server.send(m1);
              receive_second_message();
            }
            void receive_second_message(){
              byte[] m2 = server.receive();
            :
            }
}
```

The problem with `Class Alice` is that it can only be used with `class Server`. It cannot be used with any other class, even if that class has `send` and `receive` methods.

Fixing the Flaws

- `class Alice` only needs the network support: `send` and `receive` methods.
- We do not want to tie users of `class Alice` to `class Server`.
- <u>Solution:</u> use an interface to define what we expect from a network module. Then, replace `class Server` with the interface.

Rewriting The Client and The Server

```
Interface Party {
        // Interface methods (notice: methods are declared, but not defined)
        void send(byte[] m)
        byte[] receive()
}

class Client implements Party {

        Socket socket;

        // Implementation of send and receive from class Party
        void send(byte[] m) { … }
        byte[] receive() { … }
}

class Server implements Party {

        ServerSocket socket;

        // Implementation of send and receive from class Party
        void send(byte[] m) { … }
        byte[] receive() { … }
}
```

Interfaces and *abstract classes* (also known as *virtual classes*) are a sign of a healthy API driven design.

Removing “Boilerplate Code”

```
Interface Party {
            void send(byte[] m)
            byte[] receive()
}


class Client implements Party {

            Socket socket;

            // Implementation of send and receive from class Party
            void send(byte[] m) { … }
            byte[] receive() { … }
}

class Server implements Party {

            ServerSocket socket;

            // Implementation of send and receive from class Party
            void send(byte[] m) { … }
            byte[] receive() { … }
}
```

Redundancy: `send` and `receive` methods of `class Client` do the same as those in `class Server`. We will fix this using inheritance (next slide)

Removing Redundancy Using Inheritance

`send` and `receive` are declared in the interface, *but not defined*.

interface Party
- send
- receive

`class MyParty` provides a specific implementation of `send` and `receive`

class MyParty implments Party
- send {…}
- receive {…}

class Client extends MyParty

class Server extends MyParty

`class Client` and `class Server` inherit `send` and `receive` from `class MyParty`

Finishing the Client and The Server

```
Interface Party {
            // Declaration of send and receive
            void send(byte[] m);
            byte[] receive();
}

class MyParty implements Party {
            // Implementation of send and receive
            void send(byte[] m) { … }
            byte[] receive() { … }
}


class Client extends MyParty {
            Socket socket;
            :
}

class Server extends MyParty {
            ServerSocket socket;
            :
}
```

Summary of Improvements So Far

```
Interface Party {
          // Declaration of send and receive
          void send(byte[] m);
          byte[] receive();
}
```

> `class Alice` can be used from any client/server implementing `interface Party`.

```
class MyParty implements Party {
          // Implementation of send and receive
          void send(byte[] m) { … }
          byte[] receive() { … }
}
```

> `Send` and `receive` are easy to maintain/modify because they only appear in one place.

```
class Client extends MyParty {
          Socket socket;
          :
}
```

> Network I/O exceptions can be handled here, instead of in `class Alice`.

```
class Server extends MyParty {
          ServerSocket socket;
          :
}
```

> `class Client` and `class Server` can be used in any client/server application

Implementing OT : 3rd Attempt

```
class Alice {
           String X0,X1;
           Party party;
           encryption_key_length;

           main(){
             Alice alice = new Alice();
             alice.party = new Server(..);
             send_first_message();
           }

           /* methods for constructing the messages of Alice
           and processing the replies of Bob */
           void send_first_message(){
             m1 = new byte[encryption_key_length];
                      :
             server.send(m1);
             receive_second_message();
           }
           void receive_second_message(){
             byte[] m2 = server.receive();
           :
           }
}
```

Redesign – Phase 2

- Remove main(). We are building a library; not an application.
- Decouple message construction from protocol flow.
- Encapsulate the class; provide methods that set the input and get the output.

Removing main()

```
class Alice {
           String X0,X1;
           Party party;
           encryption_key_length;

           main(){
             Alice alice = new Alice();
             alice.party = new Server(..);
             send_first_message();
           }

           /* methods for constructing the messages of Alice
           and processing the replies of Bob */
           void send_first_message(){
             m1 = new byte[encryption_key_length];
                      :
             party.send(m1);
             receive_second_message();
           }
           void receive_second_message(){
             byte[] m2 = party.receive();
           :
           }
}
```

The server will be passed to the constructor of `class Alice`, and `send_first_message` will be handled in a "run" method.

Removing main()

```
class Alice {
            String X0,X1;
            Party party;
            encryption_key_length;

            public Alice(Party party) {
              this.party = party;
            }
            /* methods for constructing the messages of Alice
            and processing the replies of Bob */
            void send_first_message(){
              m1 = new byte[encryption_key_length];
                         :
              party.send(m1);
              receive_second_message();
            }
            void receive_second_message(){
              byte[] m2 = party.receive();
            :
            }
}
```

This is better. Now `Alice` can be used with *any* class implementing interface `Party`

This is not good. Messages are constructed and sent / received in various locations in the code.

Decoupling Message Construction From Protocol Flow

```
class Alice {
            String X0,X1;
            Party party;
            encryption_key_length;

            public Alice(Party party) {
              this.party = party;
            }

            void run() {
              party.send(first_message());
              process_second_message(party.receive());
              :
            }
            void byte[] first_message(){
              m1 = new byte[encryption_key_length];
              :
              return m1;
            }
            void process_second_message(byte[] m2){
            :
            }
}
```

Method `run` centrally defines message flow in the protocol. This makes the code easier to maintain and read.

Methods `first_message` and `process_second_message` are now defined purely in terms of input and output. This enables us to test their correctness independently (without having to run the protocol).

Encapsulating Data Members

```
class Alice {
          private String X0,X1;
          Party party;
          encryption_key_length;

          public Alice(Party party) {
            this.party = party;
          }

          void run() {
            party.send(first_message());
            process_second_message(party.receive());
            :
          }
          void setInput(String X0, String X1) {
            this.X0 = X0;
            this.X1 = X1;
          }
          String getOutput() {
            return null;
          }
}
```

Method `setInput` sets Alice's input. It should be called before the protocol starts (method `run`). Method `getOutput` returns Alice's output. Alice could also return an error code (0 or -1 depending on successful completion).

How The API Will be Used

```
class MyProjectServer {
          main() {
            Server server = new Server();
            Alice alice = new Alice(server);
            // Alice has two messages: "secret1" and "secret2".
            alice.setInput("secret1","secret2");
            alice.run();
          }
}

class MyProjectClient {
          main() {
            Client client = new Client();
            Bob bob = new Bob(client);
            // Bob chooses message 0.
            bob.setInput(0);
            bob.run();
            System.out.println("Message 0 is " + bob.getOutput());
          }
}
```

Recap of Case Study

- We turned class `Alice` from an application into a library.
- Developers can integrate `Alice` in any project. They only need to provides a class implementing interface `Party`.
- We wrote general purpose classes `Client` and `Server` that can be reused in other projects.
- The message functions of class `Alice` can be tested independently, without having to run the full protocol.
- The API driven design resulted in code that is easier to maintain, read, use, and debug.

New Problems: A Growing Library

How can we guarantee consistency among our protocols?

It would be ideal if all protocols have a method `run` and methods `setInput` and `getOutput` just like in our oblivious transfer protocol.

Solution: Abstraction

We define **Protocol** as an ***abstract class***.

Abstract classes play a similar role to that of interfaces.

Java interfaces can only declare functions. Java abstract classes can do that, and in addition provide implementations and data members. A class can implement several interfaces, but only extend one class (either abstract or not).

Abstract Class Protocol

All of our protocols will be of the following form:

```
abstract class Protocol {

          Party party;

          Protocol(Party party) {
            this.party = party;
          }

          abstract void setInput(String[] input);
          abstract void run();
          abstract String getOutput();
}
```

`party` is a data member because all protocols need to have means for sending and receiving messages.

Classes extending `class Protocol` must implement these methods.

Class B can extend abstract class A without implementing all abstract methods of class A. However, only sub classes of A (or B) implementing *all abstract methods* of A can be instantiated.

!

Extending Class Protocol

We rewrite `class Alice` in terms of `class Protocol`.

```
class Alice extends Protocol {

        private String X0,X1;
        encryption_key_length;

        public Alice(Party party) {
          super(party); // invoking constructor of super class
        }

        void run() {
          :
        }
        void setInput(String X0, String X1) {
          this.X0 = X0;
          this.X1 = X1;
        }
        String getOutput() {
          return null;
        }
}
```

Abstract classes are extended, whereas interfaces are implemented.

Abstract Class Protocol – Not Generic

```
abstract class Protocol {

            Party party;

            Protocol(Party party) {
              this.party = party;
            }

            abstract void setInput(String[] input);
            abstract void run();
            abstract String getOutput();
}
```

`setInput` and `getOutput` are defined in terms of Strings. This works for `class Alice`, but may not work for protocols with other input/output types.

Meta Programming

We define the class with non-fixed types:

```
abstract class Protocol<I, O> {

          Party party;

          Protocol(Party party) {
            this.party = party;
          }

          abstract void setInput(I input);
          abstract void run();
          abstract O getOutput();
}
```

`I` and `O` are parameters. They take a type value during compilation time, when we define `class Alice`.

The input and output types are defined in terms of the parameters passed for `I` and `O`.

Java and C++ are *strongly typed languages*. They disallow writing code that ignores types. In some cases this is a disadvantage. To overcome this issue, Java provides *Generics* and C++ provides *Templates*.

!

Rewriting `class Alice`

```
class Alice extends Protocol<String[], String> {

        private String X0,X1;
        encryption_key_length;

        public Alice(Party party) {
          super(party); // invoking constructor of super class
        }

        void run() {
          :
        }

        void setInput(String[] X) {
          X0 = X[0];
          X1 = X[1];
        }

        String getOutput() {
          return null;
        }
}
```

This is good. The type of `I` is `String[]` and the type for `O` is `String`. Each protocol can use different types as necessary.

Advantages of The New Design

- Modularity: all protocols extend `class Protocol`.
- Consistency: all protocols are started with the `run` method. Input is always set with `setInput`, and output is always obtained with `getOutput`.
- Protocol input and output is managed via type safe functions.

Summary

We started with an Oblivious Transfer application. Then,

- We wrote a generic client and a server.
- Our client & server can be used in other projects.
- We separated networking from the OT protocol.
- Our OT protocol can be integrated in any project.
- We separated protocol flow from message construction.
- Our code is easier to maintain and understand.
- We standardized all protocols using an abstract class.
- Our library is type safe and consistent.

Conclusion

- API driven design requires planning and programming skills.
- API driven design is costly initially, but it pays in the long run.
- Agile approach is still useful as a basis for API driven design.