Signal

Theme Song: Text Me

In this assignment, you'll compose the cryptographic primitives we've been exploring to build a secure communication framework. In particular, we build a system wherein an adversary cannot decrypt or modify messages without our knowledge. You'll become much more familiar with cryptographic libraries and be prepared to take up larger protocols later in the course.

Due Date: *Friday, February 13th.*

NOTE: You should start early on this assignment: it is harder than the last (warmup) assignment!

Contents

1 Background Knowledge

In this assignment, you will build a secure communication platform. On startup, each client specifies whether it will *listen* or *connect* to another client. Once two clients have connected, they will run a **key exchange protocol** to reach a shared secret. Once we have a shared secret, both parties run a **key derivation function** to derive two secret keys. One key will be used for a **block cipher**, which they can use to encrypt and decrypt messages. They use the second key to tag each message with a **message authentication code** to ensure that messages aren't tampered with. We go over each of these components in detail below.

1.1 Key Exchange

We've already explored **key exchange** in the previous assignment, but it's worth revisiting. Often, when two parties wish to communicate securely, it is useful to have some shared secret key that they can use to encrypt and decrypt their messages. Indeed, the block cipher that we use in this assignment requires such a secret key. **Diffie-Hellman key exchange** is one method for coming to a shared secret key.

Diffie-Hellman works in three steps. First, system parameters are generated and shared amongst the parties. These parameters include a group $\mathbb{G}$ of order q (e.g., subgroup of $\mathbb{Z}_p^*$ of prime order q) and a generator g. Once these parameters are shared, the two parties randomly sample secrets $a, b \in \mathbb{Z}_q$ respectively, and share the public values $g^a, g^b \in \mathbb{G}$ with each other. Finally, the secret and public values are combined to create a shared secret $g^{ab} \in \mathbb{G}$.

1.2 Block Ciphers

We may wish to encrypt large amounts of data without having to generate new secret keys each time nor compromising the security of our existing key. A **block cipher** is a symmetric-key encryption scheme that works on fixed-length **blocks**, one at a time. They are very useful for encrypting large or arbitrary-length data such as messages or videos: **AES (Advanced Encryption Standard)** is one such block cipher. It was adopted by NIST in 2001 after winning a 5-year public competition to become the new standard secure block cipher. To use AES, simply provide it with a message and a suitable key. It will then output the ciphertext (the API for AES is quite simple despite its complexity). The inner workings of block ciphers will be covered later in the semester, but for now, use it as a black box!

There are many modes in which a block cipher can operate, but the one we'll be using is known as **cipher block chaining mode**, or **CBC mode**. One nuance of this mode is that it requires each encryption to be initialized with an **initialization vector**, or IV for short, and this IV will be used during decryption. The IV does not need to be kept secret, but it should be chosen at random.

1.3 Message Integrity

An important aspect of communication is message integrity: we want to be sure that our messages haven't been tampered with in transit. A **MAC (Message Authentication Code)** is one way of cryptographically ensuring message integrity. MAC *generation* takes in a secret key and a message and outputs a tag for the message. MAC *verification* takes in the same secret key, a message, a MAC tag, and outputs "Verified" (or some value to indicate so) if and only if the MAC tag is valid for the given message. Otherwise, it rejects the tag, indicating that the message has been tampered with or that the MAC tag was generated incorrectly. It must be computationally hard for an adversary without the secret key to generate a valid MAC tag for any message (otherwise, this wouldn't be secure). The MAC tag that is computed on some message can be thought of as a signature but with symmetric keys. In this assignment, we use **HMAC (Hashed Message Authentication Code)**, a widely used MAC scheme.

We need to be careful about the order in which we apply our cryptographic primitives. In particular, do we compute MACs on the plaintext and then encrypt the plaintext along with the tag, or do we encrypt our plaintext first and then compute MAC on the resulting ciphertext? It turns out that only the latter approach gives a secure authenticated encryption scheme, achieving both CCA-security (for message secrecy) and unforgeability (for message integrity).

1.4 Key Derivation

The key we generate from the Diffie-Hellman key exchange may not be sufficient for AES or HMAC. For example, it may not be long enough, it may be of the wrong size, or may have a distribution that doesn't preserve the security guarantees of AES. Moreover, we want to use different keys for AES and HMAC to preserve security. To this end, we use a **secure key derivation function** to convert a shared secret into an acceptable key. In this assignment, we use **HKDF (HMAC-based Key Derivation Function)**, a widely used key derivation function, to generate secure keys. To ensure that HKDF generates different keys for AES and HMAC, we **salt** the Diffie-Hellman shared secret in the HKDF calls. Salts have been provided for you already in the stencil code, and should be passed into the HKDF call as a parameter. We will use HKDF two times in

this assignment, once to generate a key for AES and once to generate a key for HMAC.

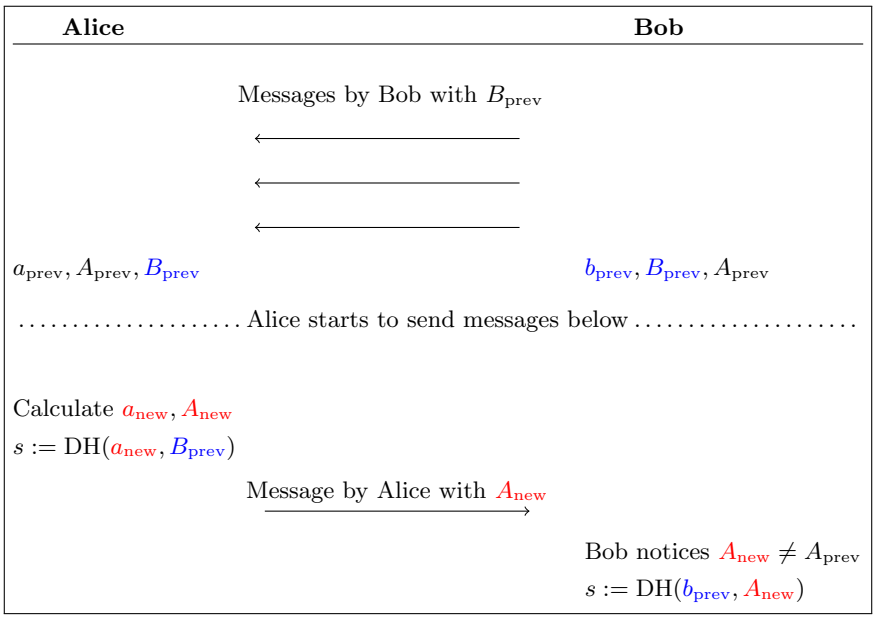
1.5 Diffie-Hellman Ratchet

An undesirable property of using a single key throughout the whole session is that if the key is leaked or cracked at any point, an adversary could read all messages in the entire session. How can we remedy this?

A solution can be found in the **Double Ratchet Algorithm** developed by the **Signal** messaging app. Criticially, they implement what is known as a **Diffie-Hellman Ratchet** where parties constantly update their shared secret. Then, even if an adversary breaks in at time T, the keys will eventually be switched, limiting the number of messages the adversary can read before needing to break in again. Please read the Diffie-Hellman Ratchet section of the spec, linked above.

In short, the ratchet works as follows. Each time the direction of communication changes (e.g., Alice starts to send messages after receiving messages), the sender will generate a new Diffie-Hellman key pair and send their new public value alongside their message.

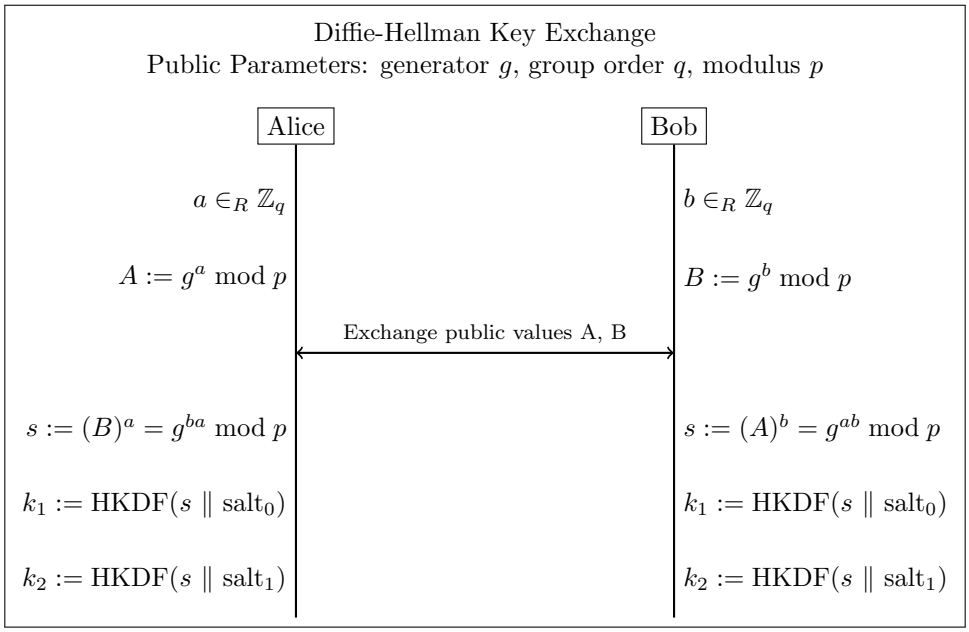
The sender will generate a new encryption key by combining their new private value with the receiver's old (i.e., last heard of) public value, and the receiver will generate a new decryption key by combining the sender's new public value with their old (i.e., last generated/current) private value. Both parties will use this shared key in AES and HMAC by first running it through HKDF, as before.



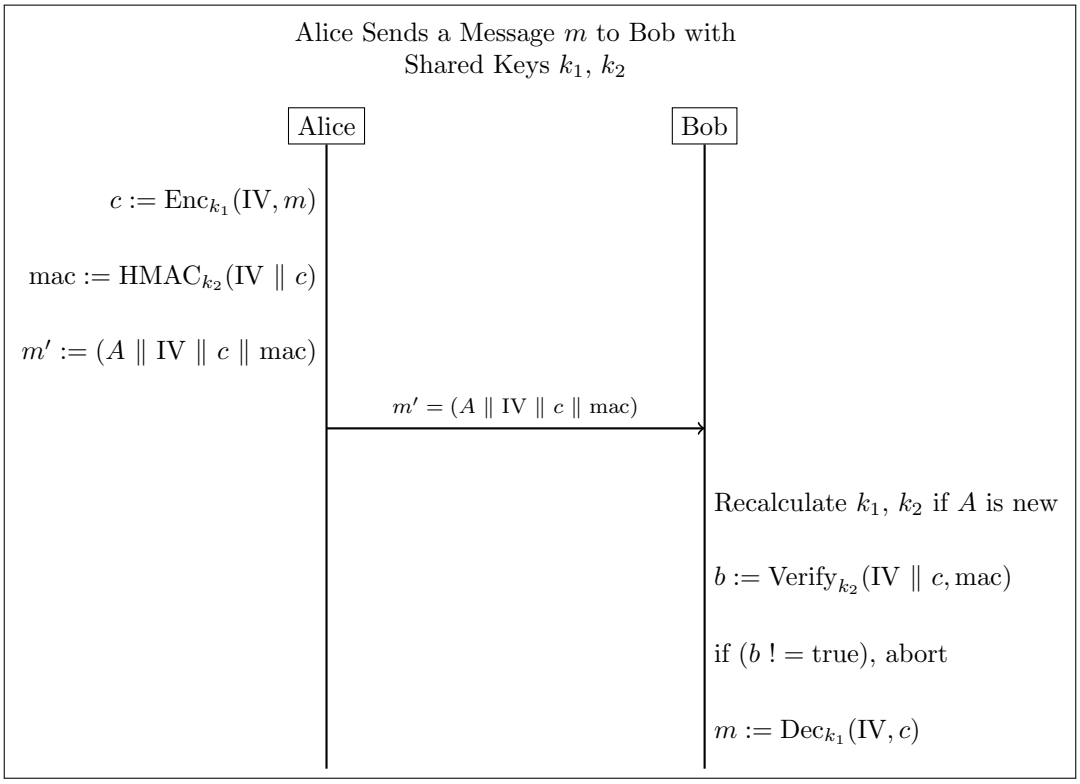
1.6 Putting it All Together

The following diagrams explain how the protocol works together.

First, the parties need to run the Diffie-Hellman key exchange and HKDF to generate two secret keys.



Then, the parties are able to securely communicate. One secret key is used for AES encryption/decryption and the other is used for MAC generation/verification. The following diagram shows the sequence for the event where Alice sends a message to Bob. The event where Bob sends a message to Alice is symmetrical.



In short, we proceed in two steps: key exchange and then secure communication.

1.6.1 Key Exchange

When our client starts up, we'll first connect to the other client we're interested in communicating with, then we'll set up to exchange keys. One of the clients (the one called "connect") will generate and send the Diffie-Hellman parameters (p, g, q) to the other (`DH_generate_params`), which will listen for them. Once the parameters are received, both clients will generate their secret and public values (`DH_generate_shared_key`) and send the public value to the other client. Lastly, each client will combine the value they received with their secret value to reach a shared secret.

1.6.2 Communication

Using the Diffie-Hellman shared secret, each client derives a key for each of AES and HMAC using HKDF (`AES_generate_key` and `HMAC_generate_key`, respectively). Each time a client wishes to send a message, they first encrypt it using AES (`AES_encrypt`)

and tag the *entire* ciphertext (including IV) with an HMAC (`HMAC_generate`). Each time a client receives a message, they verify that the HMAC is valid (`HMAC_verify`) and decrypt it using AES (`AES_decrypt`).

2 Assignment Specification

Please note: you may NOT change any of the function headers defined in the stencil. Doing so will break the autograder: if you don't understand a function header, please ask us what it means and we'll be happy to clarify.

2.1 Functionality

You will need to edit `src/drivers/crypto_driver.cxx` and `src/pkg/client.cxx`. The following is an overview of relevant files:

- `src/cmd/main.cxx` is the primary entrypoint to the program, running the Client defined in `client`.
- `src/drivers/crypto_driver.cxx` contains all of the cryptographic protocols we use in this assignment.
- `src/pkg/client.cxx` glues together the crypto and networking functions into a cohesive application.

The following roadmap should help you compartmentalize the project's main goals:

- **Diffie-Hellman Key Exchange**: Establish shared secrets using Diffie-Hellman key exchange.
- **AES Encryption**: Derive keys for use in AES using HKDF key derivation function, then encrypt and decrypt messages.
- **HMAC Authentication**: Derive keys for use in HMAC using HKDF key derivation function, then tag and verify messages.
- **Diffie-Hellman Ratchet**: Implement the ratchet to switch keys around. You should implement the prior sections before starting this one.

Some tips:

- **Read through all of our support code before beginning**. In particular, understand the drivers, message structs, and utility functions we provide. We try to flag which functions you should use in the code: however, understanding the codebase will help you problem solve better in this and future assignments, as we will reuse many of the tools we develop here.

- If a protocol fails for any reason (e.g. invalid signature, incorrect keys, decryption failed, etc.), you **must** indicate so by throwing an `std::runtime_error`. This will be true for all future assignments: we may not tell you this again!

2.1.1 Diffie-Hellman Key Exchange

Implement the Diffie-Hellman Key Exchange by editing the following functions. Once you do so, clients will be able to establish a shared key upon connection with each other.

Cryptographic functions:

- `CryptoDriver::DH_generate_params()`
- `CryptoDriver::DH_initialize(...)`
- `CryptoDriver::DH_generate_shared_key(...)`

Application functions:

- `Client::HandleKeyExchange(...)`

2.1.2 AES Encryption

Implement AES encryption by editing the following functions. Once you do so, clients will be able to encrypt and decrypt messages.

Cryptographic functions:

- `CryptoDriver::AES_generate_key(...)`
- `CryptoDriver::AES_encrypt(...)`
- `CryptoDriver::AES_decrypt(...)`

Application functions:

- `Client::prepare_keys(...)`
- `Client::send(...)`

- `Client::receive(...)`
- `Client::HandleKeyExchange(...)`

2.1.3 HMAC Authentication

Implement HMAC authentication by editing the following functions. Once you do so, clients will be able to tag and verify messages.

Cryptographic functions:

- `CryptoDriver::HMAC_generate_key(...)`
- `CryptoDriver::HMAC_generate(...)`
- `CryptoDriver::HMAC_verify(...)`

Application functions:

- `Client::prepare_keys(...)`
- `Client::send(...)`
- `Client::receive(...)`
- `Client::HandleKeyExchange(...)`

2.2 Diffie-Hellman Ratchet

Implement the Diffie-Hellman Ratchet to allow cycling of keys.

Application functions:

- `Client::prepare_keys(...)`
- `Client::send(...)`
- `Client::receive(...)`
- `Client::HandleKeyExchange(...)`

Please note that in our protocol, we run full Diffie-Hellman key exchange *first*, then continue using the Diffie-Hellman Ratchet for future messages. You can think of the first messages being sent in `HandleKeyExchange` as dummy messages, then future ones being real messages. This allows you to build the project incrementally without having to remove key exchange later on.

2.3 Support Code

Read the support code header files before coding so you have a sense of what functionality we provide. This isn't a networking class, nor is it a software engineering class, so we try to abstract away as many of these details as we can so you can focus on the cryptography.

The following is an overview of the functionality that each support code file provides.

- `src/drivers/network_driver.cxx` provides a convenient wrapper around low level network calls for you to use. All data is sent as `std::vector<unsigned char>` and have provided helper functions to convert between this type and `std::string`. Use the `send` and `read` functions to send and receive data safely.

- `src/drivers/cli_driver.cxx` provides a CLI. You may find it helpful to use the provided `CLIDriver` to print values out to the left (`print_left`) and right (`print_right`) of the screen.

- `src-shared/logger.cxx` provides logging utility functions that may be useful for debugging.

- `src-shared/messages.cxx` provides the message types that we can send and receive over the wire, including functions to serialize and deserialize these types from `std::vector<unsigned char>`. See the associated header file for the struct definitions.

- `src-shared/util.cxx` contains useful debugging and conversion primitives. Check back here first if you're not sure how to convert types: chances are we've already taken care of it!

2.4 Messaging

The following example shows how to use our message library alongside our networking library. We'll be using this pattern for the entire course, so it's good to get it down

now.

```
// --- Party 1: Sending a message ---

// Declare the message struct and populate its fields
MsgType_Message msg_s;
msg_s.value_1 = "foo";
msg_s.value_2 = 1515;

// Declare a data vector and serialize the message
std::vector<unsigned char> msg_data;
msg_s.serialize(msg_data);

// Send the data
this->network_driver->send(msg_data);

// --- Party 2: Receiving a message ---

// Read the data
std::vector<unsigned char> their_msg_data = this->↩
    network_driver->read();

// Declare a message struct and deserialize the data
MsgType_Message their_msg_s;
their_msg_s.deserialize(their_msg_data);
```

2.5 Libraries: CryptoPP

Most of the code you'll be writing in this assignment will be similar to that in the links provided below. We highly recommend perusing those wikis before starting. In particular, the Pipelining wiki page is useful for understanding how the CryptoPP library works.

You may find the following wiki pages useful during this assignment:

- CryptoPP Diffie-Hellman
- CryptoPP HKDF
- CryptoPP AES

- CryptoPP HMAC
- CryptoPP Pipelining
- CryptoPP SecBlock

  Note: A `SecByteBlock` is a variable where we can securely store keys in memory. Many library functions that you will be calling require this type. When initializing `SecByteBlock`s, *you should pass in the length of the value it will be holding as the second argument.* For example, `dh.PrivateKeyLength()` or `dh.AgreedValueLength()` or `SHA256::DIGESTSIZE`. We've included helper functions for you to print out the contents of a `SecByteBlock`.

3 Getting Started

To get started, get your stencil repository here and clone it into the `devenv/home` folder. From here you can access the code from both your computer and from the Docker container. We suggest you also check out the Debugging Guide for tips on debugging common issues.

3.1 Running

To build the project, `cd` into the `build` folder and run `cmake ..`. This will generate a set of Makefiles to build the whole project. From here, you can run `make` to generate a binary you can run, and you can run `make check` to run any tests you write in the 🗀`test` folder.

To run the binary, run `./signal_app <listen|connect> [address] [port]` (in the 🗀`build` folder) on two terminals with the same port number, one using `listen` and the other `connect` (run listen before connect). This should run the app and allow you to communicate.

> Tip: Use `localhost` as the address, and choose any number from 1025 up to 65,535 for the port. (e.g. `./signal_app listen localhost 5000`)

3.2 Testing

You may write tests in any of the `test/**.cxx` files in the Doctest format. We provide `test/test_provided.cxx`, feel free to write more tests in this file directly. If you want to add any new test *files*, make sure to add the file to the `cmake` variable, `TESTFILES`, on line 7 of `test/CMakeLists.txt` so that `cmake` can pick up on the new files. Examples have been included in the assignment stencil. To build and run the tests, run `make check` in the `build` directory. If you'd like to see if your code can interoperate with our code (which is what it will be tested against), feel free to download our binaries here - we try to keep these up to date, so if you're unsure about the functionality of our binaries, please ask us on Ed!

3.3 Collaboration

Collaboration is allowed and encouraged! However, at the end of the day, you *must* write up the solutions yourself and acknowledge any collaborators in your `README.md`.