Yaos

Theme Song: Nerdy Love

In this assignment, you'll implement oblivious transfer and garbled circuits to allow two parties to jointly compute a function without learning the other party's inputs.

Due Date: *Monday, April 28th*

Contents

1 Background Knowledge

In this assignment, you'll implement a simple version of Yao's garbled circuits using a simple version of oblivious transfer. This assignment leaves a lot of room for optimizations, which we leave to the reader to explore. However, in your final submission, **do not** submit with any optimizations as it will not be compatible with the autograder.

We highly recommend reading the first few pages of The Simplest Protocol for Oblivious Transfer for OT, and A Gentle Introduction to Yao's Garbled Circuits for garbled circuits. It will help immensely in understanding the mathematics of this assignment.

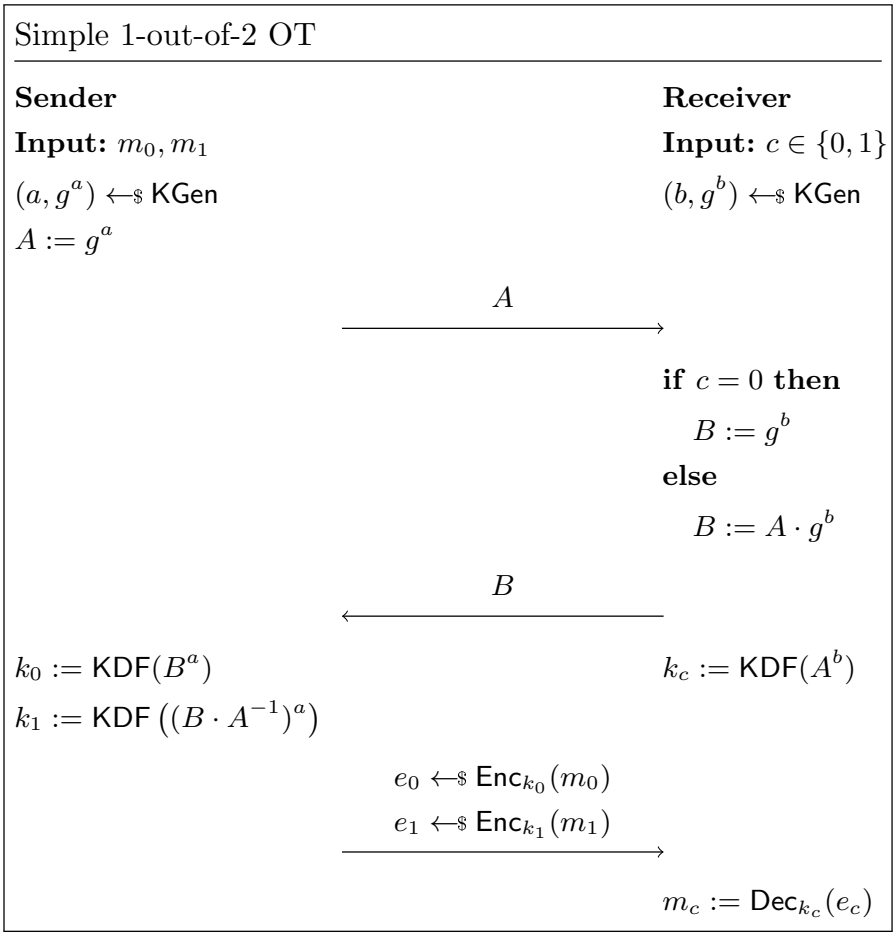
1.1 Oblivious Transfer

A foundational building block in secure multi-party computation is oblivious transfer (OT), which allows a receiver to select a message from a sender without the sender learning which message they selected, nor the receiver learning more than the message they selected. At first glance, OT seems impossible to achieve, but as we will see, we can use basic primitives that we've seen already to build a simple yet secure protocol for 1-out-of-2 OT.

We present an implementation of OT based on the Diffie-Hellman key exchange. The protocol proceeds as follows:

- The sender prepares two messages m_0 and m_1 to send to the receiver.
- The sender generates a Diffie-Hellman keypair, (a, g^a), and sends $A = g^a$ to the receiver.
- The receiver generates a Diffie-Hellman keypair (b, g^b).
  - If they wish to receive m_0, they send back $B = g^b$.
  - If they wish to receive m_1, they send back $B = A \cdot g^b$.
- The sender generates $k_0 = \mathsf{HKDF}(B^a)$ and $k_1 = \mathsf{HKDF}((B/A)^a)$ and encrypts $e_0 \leftarrow \mathsf{Enc}_{k_0}(m_0)$ and $e_1 \leftarrow \mathsf{Enc}_{k_1}(m_1)$, sending both to the receiver.
- The receiver generates shared key $k_c = \mathsf{HKDF}(A^b)$.
  - Notice that depending on the receiver's choice bit $c \in \{0, 1\}$, the key for the message they selected will be equal to k_c.

- The receiver decrypts the ciphertext they selected, retrieving $m_c := \mathsf{Dec}_{k_c}(e_c)$.



1.2 Yao's Garbled Circuits

Yao's garbled circuits allow two parties to jointly compute over a Boolean circuit without learning any intermediate values or the other party's inputs. This is an immensely useful primitive as it allows two parties to jointly compute any function securely. We'll describe a secure two-party computation protocol that uses garbled circuits (without any optimization) and our OT implementation above, along with some other primitives we've been interacting with.

All of our circuits are specified using Bristol Format, which consists of three types of gates: AND, XOR, and NOT. We provide parsers to ease development. We highly recommend reading up on the format, should you need to debug a particular circuit or wish to write your own.

We present a simple construction of garbled circuits. The **garbler** and **evaluator** are the two parties. The protocol proceeds as follows:

- The garbler parses circuit C and obtains a set of gates and wires to process.
- For each wire, the garbler samples a random 0-label and a random 1-label, each being a λ-bit string ($\lambda = 128$).
- For each gate, the garbler will produce a garbled gate consisting of 4 distinct ciphertexts per AND/XOR gate and 2 distinct ciphertexts per NOT gate, where each ciphertext is a double encryption of the corresponding output label (tagged with λ trailing 0's so we can identify when decryption was successful) using the two input labels as keys. We instantiate the double encryption by a hash function. Concretely, let's say we're garbling an AND gate with input wires w_x, w_y and output wire w_z. Then, the ciphertexts will be as follows:

$$
\begin{aligned}
c_{0,0} &:= \mathsf{Enc}_{w_x^0, w_y^0}(w_z^0) = \overbrace{\mathsf{SHA256}(w_x^0 \mid\mid w_y^0)}^{256 \text{ bits}} \oplus (\overbrace{\ w_z^0\ }^{128 \text{ bits}} \mid\mid \overbrace{0 \ldots 0}^{128 \text{ bits}}) \\
c_{0,1} &:= \mathsf{Enc}_{w_x^0, w_y^1}(w_z^0) = \mathsf{SHA256}(w_x^0 \mid\mid w_y^1) \oplus (w_z^0 \mid\mid 0 \ldots 0) \\
c_{1,0} &:= \mathsf{Enc}_{w_x^1, w_y^0}(w_z^0) = \mathsf{SHA256}(w_x^1 \mid\mid w_y^0) \oplus (w_z^0 \mid\mid 0 \ldots 0) \\
c_{1,1} &:= \mathsf{Enc}_{w_x^1, w_y^1}(w_z^1) = \mathsf{SHA256}(w_x^1 \mid\mid w_y^1) \oplus (w_z^1 \mid\mid 0 \ldots 0)
\end{aligned}
$$

- The garbler randomly permutes all 4 (or 2) ciphertexts for each garbled gate and sends all of them to the evaluator.
- The garbler also sends labels corresponding to the garbler's input.
- The evaluator runs OT with the garbler to retrieve the labels corresponding to its input.
- The evaluator evaluates the garbled circuit gate by gate, from the input labels all the way to output labels.
- The evaluator sends the labels corresponding to the output wires to the garbler, which then reveals the final output to the evaluator.

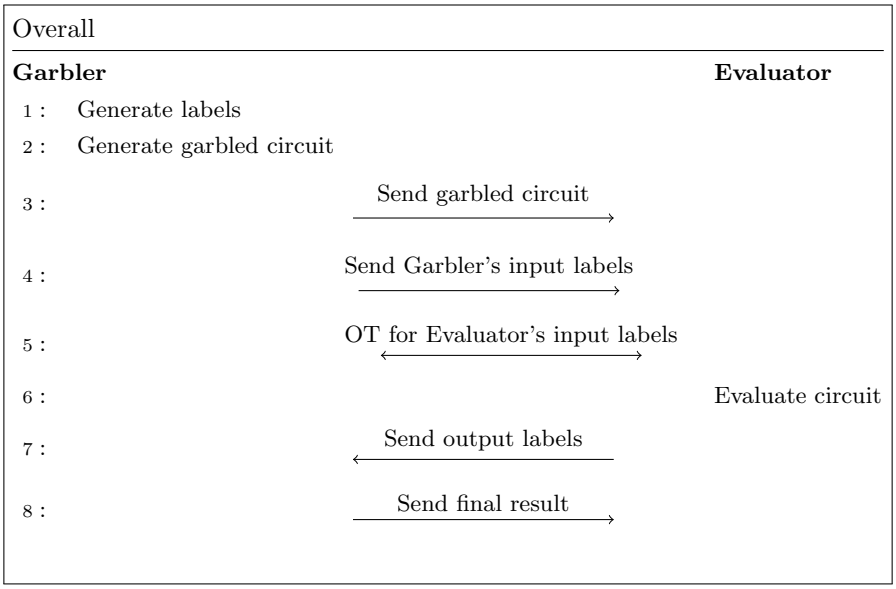
1.3 Some Resources

The following papers are incredibly useful for gaining a full understanding of protocols like ours:

- The Simplest Protocol for Oblivious Transfer
- A Gentle Introduction to Yao's Garbled Circuits
- Faster Secure Two-Party Computation Using Garbled Circuits

1.4 Putting It All Together

The following diagram explains how the protocol works together.



We don't give a detailed protocol description as it doesn't differ meaningfully from the base protocol described above.

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 primarily need to edit `src/drivers/ot_driver.cxx`, `src/pkg/garbler.cxx`, and `src/pkg/evaluator.cxx`. The following is an overview of relevant files:

- `src/cmd/garbler.cxx` is the main entrypoint for the `yaos_garbler` binary. It calls the `Garbler` class.
- `src/cmd/evaluator.cxx` is the main entrypoint for the `yaos_evaluator` binary. It calls the `evaluator` class.
- `src/drivers/ot_driver.cxx` contains a driver for OT.
- `src/pkg/garbler.cxx` Implements the `Garbler` class.
- `src/pkg/evaluator.cxx` Implements the `Evaluator` class.

The following roadmap should help you organize concerns into a sequence:

- **Oblivious Transfer**: Implement 1-out-of-2 OT.
- **Garbled Circuit Generation**: Implement label and gate generation.
- **Garbled Circuit Evaluation**: Implement gate evaluation.

Some tips:

- Use the `DUMMY_RHS` value for a dummy wire when you need to evaluate a `NOT` gate.
- When evaluating a gate in the `Evaluator`, we recommend looking at how labels are generated as reference.
- Remember to use the constants provided for `LABEL_LENGTH` and `LABEL_TAG_LENGTH`.

- You may have to use `DH_generate_shared_key` and `AES_generate_key` a few times in OT: this is expected.

- Use `byteblock_to_integer` and `integer_to_byteblock` to convert to and from integers and byteblocks.

2.1.1 Oblivious Transfer

Implement oblivious transfer. Once you do so, your clients will be able to transfer one of two values obliviously.

Cryptographic functions:

- `OTDriver::OT_send(...)`

- `OTDriver::OT_recv(...)`

2.1.2 Garbled Circuit Generation

Implement garbled circuit generations. Once you do so, a valid garbled circuit will be generated, ready to be evaluated.

Cryptographic functions:

- `GarblerClient::generate_gates(...)`

- `GarblerClient::generate_labels(...)`

- `GarblerClient::encrypt_label(...)`

Application functions:

- `GarblerClient::run(...)`

- `EvaluatorClient::run(...)`

2.1.3 Garbled Circuit Evaluation

Implement garbled circuit evaluation. Once you do so, your two parties will be able to perform generic 2PC. Hooray!

Cryptographic functions:

- `EvaluatorClient::evaluate_gate(...)`

Application functions:

- `GarblerClient::run(...)`
- `EvaluatorClient::run(...)`

2.2 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-shared/circuit.cxx` contains type definitions and loaders for a Boolean circuit written in Bristol format.
- Everything else from prior assignments is unchanged.

2.3 Libraries: CryptoPP

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

- CryptoPP nbtheory

3 Getting Started

Important: If the docker has been finicky or slow for you, you might also try out Github Codespaces! See our section in development environment handout here.

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.

3.1 Running

To build the project, `cd` into the `build` folder and run `cmake ..`. This will generate a set of Makefiles building 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.

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!