CS 4803/8803 PC: Programmable Cryptography **Fall 2026**

Homework 1

Due: 6pm, Friday, 4 September 2026 (submit via Gradescope)

Instructions: You **must** typeset your solution in LaTeX using the provided template:

https://faculty.cc.gatech.edu/~aozdemir/class/pc26/homework.tex

Submission Instructions: You must submit your problem set via Gradescope. Please use course code **4DN8ZD** to sign up. Note that Gradescope requires that the solution to each problem starts on a **new page**.

Bugs: We make mistakes! If it looks like there might be a mistake in the statement of a problem, please ask a clarifying question on Ed.

Problem 1: A PRG hybrid argument [15 points]. Let G be a PRG from $\{0,1\}^\lambda$ to $\{0,1\}^{2\lambda}$. Use a hybrid argument to prove that

$$G'((s_1, s_2)) = (G(s_1), G(s_2))$$

is a PRG.

Problem 2: PRG true-or-false [15 points]. Let G be a PRG from $\{0,1\}^\lambda$ to $\{0,1\}^{2\lambda}$. Let G' be PRG from $\{0,1\}^{2\lambda}$ to $\{0,1\}^{3\lambda}$. Is each of the following G_i a PRG: true or false? If true, give a 1-2 sentence justification. If false, describe the attack in 1-2 sentences.

1. $G_1(s) = (s, G(s))$
2. $G_2((s_1, s_2)) = (s_1, G(s_2))$
3. $G_3(s) = G'(G(s))$
4. $G_4(s) = G(s) \oplus G(s)$
5. $G_5(s) = (G(s), G(s))$

Problem 3: Multi-Commitments [15 points]. Let $\mathbb{G}$ be a group of prime order q in which the discrete logarithm problem is hard. Let g and h be generators of $\mathbb{G}$. As we saw in class, the Pedersen commitment scheme commits to a message $m \in \mathbb{Z}_q$ using randomness $r \in \mathbb{Z}_q$ as $\mathsf{Commit}(m; r) := g^m h^r \in \mathbb{G}$. The "public parameters" associated with the Pedersen commitment scheme are the description of the prime-order group $\mathbb{G}$ and the group elements g and h.

(a) Use $\mathbb{G}$ to construct an additively homomorphic commitment scheme $\mathsf{Commit}_n(m_1, \ldots, m_n; r)$ that commits to a length-n vector of messages $(m_1, \ldots, m_n) \in \mathbb{Z}_q$ using randomness $r \in \mathbb{Z}_q$. The output of the commitment should be short – only a single group element. You should specify both the public parameters of your scheme (which may be different from that of the basic Pedersen commitment scheme) as well as the description of the Commit_n function.

(b) Prove that your commitment scheme is perfectly hiding and computationally binding (assuming hardness of discrete log in $\mathbb{G}$).

Problem 4: Implementation: coin flipping [15 points]. This problem is a way to learn a little Rust while implementing a cool secure-computation protocol. (The Rust will come in handy for future problems.) Your task is to implement Blum's coin-flipping protocol and Pedersen commitments, as we described in class. The protocol is:

Algorithm 1 Alice (speaks first)

1: $b_A \xleftarrow{\$} \{0,1\}, \quad r \xleftarrow{\$} \mathbb{Z}_q$
2: $c \leftarrow \mathsf{Commit}(b_A; r)$
3: send c to Bob
4: receive b_B from Bob
5: send the opening (b_A, r) to Bob
6: **return** $b_A \oplus b_B$

Algorithm 2 Bob (speaks second)

1: receive c from Alice
2: $b_B \xleftarrow{\$} \{0,1\}$
3: send b_B to Alice
4: receive (b_A, r) from Alice
5: **if** $c \neq \mathsf{Commit}(b_A; r)$ **then** abort
6: **return** $b_A \oplus b_B$

Toolchain You need Rust, `make`, `zip`, `unzip`, and a bash shell. Install Rust with rustup (https://rustup.rs) on every platform; pre-installed Rust version may be too old. For the rest of the toolchain:

- Linux: you likely already have it, if not, use your package manager, e.g. `sudo apt install build-essential zip unzip`.
- macOS: `xcode-select --install` supplies `make` and the linker; `zip`/`unzip` are already installed.
- Windows: use WSL and follow the Linux instructions there. Keep the crate in the Linux filesystem (`~/coin_flip`) rather than under `/mnt/c/...`.

If you are new to Rust, I recommend using VS Code with the **rust-analyzer** extension, which works on every platform. (On WSL, install VS Code on Windows together with its **WSL** extension, then run "`code .`" from a WSL shell and accept the prompt to install rust-analyzer *into* WSL.)

Note: we'll be writing somewhat 'simple' Rust code, but the language can still be challenging. Feel free to ask for help at office hours or from each other.

What to implement Start from `coin_flip-starter.zip`, which unpacks into a Rust crate in a directory named `coin_flip`. You will implement `commit` and the Alice and Bob branches of `main`, inside `src/main.rs`. Many of the basics are implemented for you: the group (`Group`), its scalars (`Scalar`), operations over them (`+`, `-`, `*`, ...), the generators (`g()`, `h()`), sampling (`random_bit()`, `random_scalar()`), conversion (`bit_to_scalar()`), connection setup (`connect_to_peer`), and serialization (`read`/`write`, which send any `arkworks` value—or a tuple of them—over the socket).

Resources You may find the following resources helpful:

- The Rust Book, if the language is new to you: https://doc.rust-lang.org/book/
- arkworks field documentation: https://docs.rs/ark-ff/

- arkworks group documentation: https://docs.rs/ark-ec/: the group operations live here: `+` between two group elements, and `*` between a group element and a `Scalar`.
- documentation for other arkworks crates is also online

Testing Your `main` is a binary that takes the party name as its only argument, and each party prints the agreed-upon bit (as `0` or `1`) and exits. Alice listens and Bob connects, so start Alice first:

```
$ cargo run -- alice      # terminal 1
$ cargo run -- bob        # terminal 2
```

Both parties use TCP port 27183 on `localhost`. If you see "address already in use" or a hang, a party from an earlier run is probably still alive; kill it and try again.

You can type-check, lint, build, and run basic tests with the `Makefile`:

```
$ make check              # type-check, check format, and lint
$ make build              # build
$ make test               # build, then test (20 trials)
```

Submitting In the crate directory, run

```
$ make zip
```

which cleans the build artifacts and writes `../coin_flip-solution.zip`, a zip of the whole crate, one directory up. Submit that zip file.

Grading "`make test`" just tests correctness, but your protocol will also be hand-graded for security. Be sure to implement all of Blum's protocol!

Optional Feedback [0 points]. Please answer the following questions to help us design future problem sets. You do not need to answer these questions, and if you would prefer to answer anonymously, please use this form. However, we do encourage you to provide us feedback on how to improve the course experience.

(a) What was your favorite problem on this problem set? Why?

(b) What was your least favorite problem on this problem set? Why?

(c) Do you have any other feedback for this problem set?

(d) Do you have any other feedback on the course so far?