

Homework 3

Due: 6pm, Friday, 2 October 2026 (submit via Gradescope)

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

<https://cs8803pc.cc.gatech.edu/hw/template.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: Conceptual questions on ZK and succinctness [12 points]. For each of the following statements, say whether it is TRUE or FALSE. Write **at most three sentences** to justify your answer. One sentence likely suffices.

- If an interactive proof system has soundness error greater than $1/3$, then it cannot be a proof of knowledge with knowledge error less than $1/3$.
- In class, we showed how to use the Fiat-Shamir heuristic to construct non-interactive zero-knowledge proofs *without* the random oracle model.
- In the standard model (without random oracles), there exist non-interactive zero-knowledge proofs for languages in BPP.
- Consider a hash function $H : \mathcal{W} \rightarrow \mathcal{X}$, and the NP-relation $\mathcal{R}_n$ for knowledge of n pre-images of H . Formally, $\mathcal{R}_n$ has instance space $\mathcal{X}^n$, witness space $\mathcal{W}^n$, and is defined by $\{(x \in \mathcal{X}^n, w \in \mathcal{W}^n) : H(w_1) = x_1 \wedge H(w_2) = x_2 \wedge \dots \wedge H(w_n) = x_n\}$. A SNARG for $\mathcal{R}_n$ must have verification time sublinear in n .

Problem 2: A ZK-PoK for equal discrete logarithms [15 points]. Let $\mathbb{G}$ be a group of prime order p . Let ϕ be the following relation:

$$\phi(x = (g_1, h_1, g_2, h_2) \in \mathbb{G}^4, w \in \mathbb{Z}_p) \triangleq h_1 = g_1^w \wedge h_2 = g_2^w$$

That is, ϕ is a relation for the equality of two discrete logarithms. Give a ZK-PoK (P, V) for ϕ and prove that it has the following properties:

- perfect completeness
- knowledge soundness for any prover P^* satisfying $\Pr[\langle P^*, V \rangle(x) = 1] = 1$
- perfect honest-verifier zero knowledge

[Hint: Start by duplicating Schnorr's protocol (in parallel). Now, try to share as much as possible between the two duplicates. Many parts of the protocol (but not all) can be shared since the logarithms are the same.]

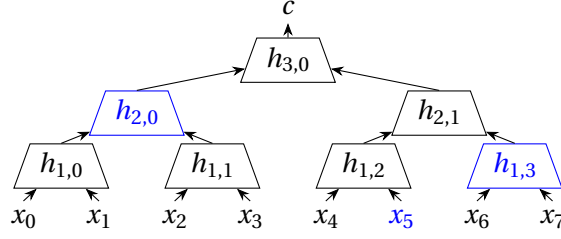


Figure 1: A Merkle tree

Problem 3: Merkle trees [15 points]. In this problem we consider *vector commitments*: commitments to vectors which can be opened to one index at a time.

The classic construction vector commitment scheme is the *Merkle tree*, which is parameterized by a hash function $H : \mathcal{X} \times \mathcal{X} \rightarrow \mathcal{X}$. The core of this construction is a hash tree, as shown in Figure 1. Each internal node in the tree represents an evaluation of the hash function over the child nodes. The Commit procedure evaluates the hash tree, returning the root at the commitment. The IdxOpen procedure produces the siblings along a path (e.g., the blue nodes, for x_4), and the IdxVerify checks the hash evaluations along the path.

More precisely, the Merkle tree vector commitment scheme is defined by three algorithms:

- $\text{Commit}(d \in \mathbb{N}, x \in \mathcal{X}^{2^d}) \rightarrow c \in \mathcal{X}$:
 for $i \in \{0, 1, \dots, 2^d - 1\}$: $h_{0,i} \leftarrow x_i$
 for $\ell \in \{1, 2, \dots, d\}$, for $i \in \{0, 1, \dots, 2^{d-\ell} - 1\}$: $h_{\ell,i} \leftarrow H(h_{\ell-1,2i}, h_{\ell-1,2i+1})$
 return $h_{d,0}$
- $\text{IdxOpen}(d \in \mathbb{N}, i \in \{0, \dots, 2^d - 1\}, x \in \mathcal{X}^{2^d}) \rightarrow p \in \mathcal{X}^d$:
 compute $h_{\ell,i}$ as in Commit
 for $\ell \in \{0, \dots, d-1\}$: $p_\ell \leftarrow h_{\ell, ((i \gg \ell) \oplus 1)}$ ¹
 return $(p_0, \dots, p_{d-1})$
- $\text{IdxVerify}(d \in \mathbb{N}, i \in \{0, \dots, 2^d - 1\}, x \in \mathcal{X}, c \in \mathcal{X}, p \in \mathcal{X}^d) \rightarrow \{0, 1\}$:
 $h_0 \leftarrow x$
 for $\ell \in \{1, \dots, d\}$:
 $j \leftarrow i \gg (\ell - 1)$
 if j is odd: $h_\ell \leftarrow H(p_{\ell-1}, h_{\ell-1})$
 else: $h_\ell \leftarrow H(h_{\ell-1}, p_{\ell-1})$
 return $h_d \stackrel{?}{=} c$

¹Here, $\gg$ is logical right shift and $\oplus$ is bitwise XOR, as in C. That is, $x \gg y$ denotes $\lfloor x/2^y \rfloor$ and $x \oplus y$ denotes the integer with binary representation equal to the bitwise XOR of the binary representations of x and y .

A vector commitment scheme is *index binding* if for all $d \in \mathbb{N}$ and for all efficient adversaries $\mathcal{A}$,

$$\Pr \{ \text{IdxVerify}(d, i, x, c, p) = 1 \wedge \text{IdxVerify}(d, i, x', c, p') = 1 \wedge x \neq x' : (c, i, x, p, x', p') \leftarrow \mathcal{A}(d, \lambda) \} = \text{negl}(\lambda)$$

where λ is the security parameter of H . A hash function H is *collision resistant* if for all efficient adversaries $\mathcal{A}$,

$$\Pr \{ H(x, y) = H(x', y') \wedge (x, y) \neq (x', y') : (x, y, x', y') \leftarrow \mathcal{A}(\lambda) \} = \text{negl}(\lambda)$$

- (a) Prove that Merkle trees are index binding if H is collision-resistant.
- (b) Now, we model H as a random oracle, that all algorithms have oracle access to: $\text{Commit}^H, \text{IdxOpen}^H, \text{IdxVerify}^H$. This includes any adversary $\mathcal{A}^H$. Fix $\mathcal{X} = \{0, 1\}^\lambda$.

Prove that Merkle trees are index binding in the ROM. You can no longer *assume* that H is collision-resistant.

[**Hint:** You can prove this directly, or by proving collision resistance and using part (a).]

[**Hint:** Either way, you will not need a reduction for this proof.]

- (c) Now, show that **Commit** is **not** hiding in the ROM.
- (d) Now, explain how to modify **Commit**, **IdxOpen**, and **IdxVerify** (in the ROM) to preserve index-binding and also achieve hiding. Call your new procedures Commit' , $\text{IdxOpen}'$, and $\text{IdxVerify}'$. You will need to add another argument to at least some of the algorithms. You need not *prove* that your modification is index-binding or hiding, you can just explain informally. $\text{IdxOpen}'$ does not need to have *any* privacy properties.

Problem 4: A Sigma protocol for circuit satisfiability [18 points]. Now, you will implement a sigma protocol for circuit satisfiability. Actually, you'll implement a number of necessary sub-protocols: Schnorr's protocol, protocols for the value in a Pedersen commitment, an AND protocol for a row in a truth table for a logic gate, and an OR protocol for a whole truth table.

Toolchain Just as for prior problems, you'll need Rust (via rustup).

How to implement a sigma protocol We will represent sigma protocols with the `SigmaProt` trait. Learn about traits in the [Rust Book](#) or [by example](#). A `SigmaProt` has the following components:

- `SigmaProt/Self` (main type): think of this as the instance x .
- `Witness` (associated type): the witness w .
- `Commitment` (associated type): the prover's first message
- `ProverState` (associated type): state used to generate the first message, which must be saved to respond
- `Response` (associated type): the prover's response
- `prover_commit` (function): generate a commitment `cmt` and state `st`, from a witness `w` and instance `self`

- `prover_respond` (function): generate a response `r` from a witness `w`, instance `self`, challenge `c`, and state `st`
- `verifier_check` (function): check a transcript: instance `self`, commitment `cmt`, challenge `c`, and response `r`.
- `simulate` (function): simulate a transcript for a fixed instance `self` and challenge `c`.

Read `schnorr.rs` for an example of how these types are defined for a protocol that you are familiar with.

What to implement Start from `sigma_sat-starter.zip`, which unpacks into a Rust crate in a directory named `sigma_sat`.

1. **Schnorr's protocol:** First, you'll [implement Schnorr's protocol in `schnorr.rs`](#). For this protocol, and all others, the associated types are defined for you and you must implement the *functions*.

After implementing, you can run “`cargo test -- test::schnorr`” to test it.

2. **Pedersen bit value protocols:** Now, you'll implement a protocol to show that a public Pedersen commitment is to a public value. Since this protocol is just a special case of Schnorr: Schnorr with a particular instance, you don't need to implement `SigmaProt` for a new type. Instead, you'll [implement a function `bit\_value\_sigma` in `bit\_value.rs`](#) to return a `Schnorr` to show that `cmt` is a commitment to value.

After implementing, you can run “`cargo test -- test::bit_value`” to test it.

3. **Truth table row:** Now, you'll [implement `TruthTableRowSigma` in `row.rs`](#): a sigma protocol to show that three commitments are to three values—the values being the row of a truth table for a binary, boolean logic gate.

This is an example of an AND sigma protocol.

After implementing, you can run “`cargo test -- test::row`” to test it.

4. **Truth table:** Now, you'll [implement `TruthTableSigma` in `table.rs`](#): a sigma protocol to show that three commitments are to *some* row of a given truth table.

This is an example of an OR sigma protocol.

After implementing, you can run “`cargo test -- test::table`” to test it.

After that, the next step would be to implement a full sigma protocol for circuit satisfiability, using the approach from class. (That is, commit to every wire value, and do a sigma protocol for the output being 1 and every gate being evaluated consistently with its truth table.) This is simple in theory, but a little involved, so it's done for you.

To test everything, end-to-end, run “`cargo test`”.

A sigma protocol for AES Now, you can write sigma protocols about circuits! There's a circuit for AES (at the standard $\lambda = 128$ security level) in `circuits/aes_128.txt`. Run “`make test-aes`” to create and verify an AES proof, and print some statistics about it. It should take about 5 minutes, which is much better than the 100+ **years** that you estimated it would take to prove AES via graph 3 coloring (last assignment).

[Report the proving and verifying time for your AES proof \(in \$\LaTeX\$ \).](#)

What factors make your sigma protocol so much faster?

Give two reasons your sigma protocol is faster than your 3-color-based protocol (in $\LaTeX$).

Extra credit [2 points] Find a way to change something in your sigma protocol to cut the proving time in half, without giving up much security or verifying time. (Most improvements will probably half the verifying time too.) Explain your change here, along with the proving time before and after. Your change must still be secure, but you don't have to prove it.

Problem 5: Time Spent [1 point for answering]. How long did you spend on this problem set? This is for calibration purposes, and the response you provide will not affect your score.

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 the online anonymous 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?