CS 5433 SP24: Homework 1

Instructor: Ari Juels
TAs: Dani Vilardell, Carolina Ortega
Due: 26 February 2025, 11:59 PM ET

Policies and Submission

Submit your written solutions and code to CMSX.

Integrity. For all assignments, you may make use of published materials, but you must acknowledge all sources, in accordance with the Cornell Code of Academic Integrity. Additionally, you must ensure that you understand the material you are submitting; you must be able to explain your solutions to the course instructor or TA if requested. You must complete this homework assignment *on your own*. You are permitted to use LLMs (e.g., ChatGPT), provided you indicate in code comments where you made use of them.

Compatibility. Your submitted code must be compatible with `python3.10`. The directory structure of your submission must follow the specification shown at the end of this document.

Problem 1: The Original Proof-of-Work Digital Currency

(30 points). In this problem, we'll explore the first proof-of-work-based digital currency we're aware of, called *Micromint*. It was devised by Ron Rivest and Adi Shamir and first published in 1996.

In this scheme, a coin is represented by a k-way *hash collision*. An example of a 4-way hash collision is shown in Figure 1. Finding such collisions is expensive. Creating coins therefore requires a substantial initial investment, but after a time, collisions quickly cascade. In other words, during the minting process, which involves hashing randomly generated preimages, it is the case that after the first collision is found, other coins accumulate soon afterward. (Exercise for yourself: why is this the case?) This situation replicates the economics of a real-world mint: The up-front manufacturing costs for currencies is high, but the incremental cost for producing units of currency is low.

Random preimages	SHA-256 hash of preimage
`1234a045de68a2a3`	**`1513a481`**`06930d57acd09762d816128351e0026d9ebb2c07387d59e9466d0188`
`123438b4f5eca470`	**`1513a481`**`765f195fb8a77e22971d2782b08aeb7dcafcfa88272a0fa0ca12d574`
`12347815572d6366`	**`1513a481`**`834fee17905090e1cc8115db9c1d4d0a766be18f95b5126474c6df53`
`12342fb88044dc3d`	**`1513a481`**`86bc4d5d60a27a17e582eac38162112c042cb0fa26eb9ea6c80a7d36`

Figure 1: A 4-way, SHA-256 hash collision of the leading 32 bits. Notice that the hashes of the preimages have the same first 32 bits; that's what makes this particular set of preimages a hash collision.

In this problem, we use the notation $[x]_i$ to denote the first i bits of a bitstring x (we mean the most significant bits).

Each coin you produce will include a "watermark." Specifically, each preimage includes a sixteen-bit value $w = [H(\mathsf{nid})]_{16}$, where nid is the ASCII encoding of your NetID. See Figure 2 for an example. The purpose of the watermark is to ensure that coins are tied to the mint that produced them.

Net ID:	`dg750`
H(Net ID):	**`1234`**`3810e62bd6ad7dad8ea0064cb3423928355f1c532c10876c45ec8794bbf6`
	The first 16 bits of the hash of this Net ID are **1234**.

Figure 2: The watermark calculation.

Coin format: A coin $C = (c_1, c_2, \ldots, c_k)$ such that $[H(c_i)]_n = y$ for all $i \in [1, \ldots, k]$ and for some value y (you can choose y). Each preimage c_i must be 8 bytes (i.e. 64 bits) long and have w as its first 2 bytes (i.e. 16 bits), where w is as described above. In other words, for each $i \in [1, \ldots, k]$, it must be that $[c_i]_{16} = w$. Note that computing $H(c_i)$ using the Python's hash library requires representing c_i in bytes (make sure you input **8 bytes** to your hash function). Remaining parameters are set as follows: $k = 4, n = 28$ and the hash function H is `SHA-256`. We do not expect generating collisions to take more than a few minutes if implemented correctly. Figure 1 is a valid example of such a collision ($k = 4$) for Net ID `dg750`.

Problem parts: For this problem, you will need to submit three files: `coin.txt`, `forged-watermark.txt`, and the code that you used to generate the solutions in `generate.py`.

A) *Coins:* In a file `coin.txt`, provide one valid coin C for your mint. The file must have exactly k lines with the ith line storing c_i in hexadecimal form (without the leading "0x").

B) *Forging watermarks:* For your coin C, forge an alternative netid nid^*. That is, present a netid $\mathsf{nid}^* \neq \mathsf{nid}$ such that $[H(\mathsf{nid}^*)]_{16} = w$, where w is the original watermark that you used above (again, where we hash the ASCII encoding of nid^*). Your forged netid nid^* must take the form $\ell^i d^j$, where ℓ is a lowercase letter, d is a digit, $i \in \{2, 3\}$, and $j \in \{1, \ldots, 10\}$. An example is provided below. Please specify the forged netid nid^* in a file `forged-watermark.txt`.

Example: To help with testing and also explain the format, we provide a valid coin. Assuming nid = `af535`, the watermark is w = `1000011001111010` and the coin is C =(`867a95c2a8781d95`, `867a79c683c4b9de`, `867a18839dcbd23f`, `867aee195b47b3d2`). This same coin is also provided in `example_coin_af535.txt`. (You must submit your own coin for your own netid!) A forged watermark is nid^* = `qn00061`. In addition, to help you debug and test your answers, in `verify_coin.py` we have provided you with a function to test if your coin is valid, and if your watermark passes the test. You can run it on the provided example coin above, using

```
python verify_coin.py af535 example_coin_for_af535.txt
```

You will **not** get full credit in this problem if your own coin does not pass this same test!

Note: Micromint is a centralized scheme—a requirement to prevent double-spending. It is not a full-blown cryptocurrency.

Problem 2: Merkle Trees

(30 points). As discussed in class, a Merkle tree is a way to authenticate a large number of digital objects (leaves of the tree) using just a small digest (sometimes called a "commitment") that consists of the root of the tree. In this exercise, you'll build a Merkle tree and create an interface that allows users to query a leaf and receive a proof of inclusion of the leaf in the tree. Note that the proof size and time to generate the proof must be $\mathcal{O}(\log n)$ where n is the number of objects. You will also implement the verification algorithm to authenticate the objects, which again must run in $\mathcal{O}(\log n)$ time.

For the purposes of this question, the Merkle tree you will build must be a complete binary tree, and you will use `SHA-256` as the hash function. Note that in a complete binary tree, every level must be completely filled with nodes, with the exception of the bottom-most level (the leaves) — it is up to you whether to fill all of the leaves or not. Complete and submit the `merkle.py` file, please stick to the given function signatures and do not modify them in any way. In particular, do not change the input/output types for the functions (but you can add your own helper functions, etc).

Problem parts:

1. Implement the `Prover.build_merkle_tree` method that takes in a list of objects represented as strings, builds a Merkle tree and returns a corresponding commitment. The commitment must be a string of hexadecimal digits (as in hexdigest) of length 64. Complete the `Prover.get_leaf` method that returns the object at the particular leaf index. Note that index starts from 0, left to right. Return `None` if there is no object at the given index.

   Hint: Complete binary trees can be implemented as arrays (multi-dimensional arrays also an option).

2. Complete the `Prover.generate_proof` method that takes as input a leaf index and returns the proof string. Return `None` if there is no object at the given index.

3. Complete the `verify` function that takes as input an object string, a proof string, and a commitment string, and returns True if the proof is valid for the given object with respect to the commitment, else returns False.

   Note: Your `verify` function must be compatible with the methods implemented in the previous parts, but should not share any state with the `Prover` class. Except for any import statements, write all your code in the `Prover` class and the `verify` function.

You can use the provided `test.py` to sanity check your code. Although it does not cover all of the test cases, you can get a sense of how we use the `Prover` class and `verify` function in the final test. We stress that `test.py` is only a sanity check, and does not catch the majority of problems. We recommend that you test your own code (e.g. by writing more test cases).

This exercise will serve as a stepping stone for Problem 3.

Problem 3: Signing via Hashing

(35 points). Hash functions are useful for many things, among them building digital signature schemes. (The Merkle signature scheme is the best known such construction.) In this exercise, you'll explore a hash-based signature scheme that is a variant on the Merkle scheme. Our scheme makes use of a Merkle tree. The basic idea is that a message is mapped to a collection of leaves of the tree. A signature consists of preimages of those leaves, along with corresponding paths from the leaves to the root such that a verifier (by means of the algorithm Verify) can check that these preimages are indeed correct.

The Merkle tree in this exercise is of depth $d + 1$. The root of the tree is the public key pk. There are 2^d leaves. For each leaf $i \in \{0, \ldots, 2^d - 1\}$, we associate a value s_i. Each s_i is generated by first sampling a keypair (s_i^{-1}, s_i), where $s_i^{-1} \in \{0,1\}^\ell$ is a secret string and sampled uniformly at random, and $s_i \in \{0,1\}^\ell$ is the image of s_i^{-1} under some keypair generation algorithm. (We will assume that the keypair generation algorithm is given to us.) The private key sk of the signature scheme is the set of preimages $\{s_i^{-1}\}$ of all leaves.

Let $H : \{0,1\}^* \to \{0,1\}^\ell$ be a hash function (SHA-256 for this exercise) such that $\ell \geq d$. Let k be an additional "difficulty-of-forgery" parameter that we will choose later. To sign a message m, the signer performs the following procedure for each $j \in \{1, 2, \ldots, k\}$: The signer computes $z_j = H(j \parallel m) \bmod 2^d$, where j is represented using 256 bits in big-endian binary encoding, and $\parallel$ denotes concatenation. (Each z_j value lies in $\{0, \ldots, 2^d - 1\}$ and represents the index of a leaf of the Merkle tree.) The signer sets $\sigma_j = s_{z_j}^{-1}$ and the resulting signature sig on m is $\sigma = \sigma_1 \parallel \ldots \parallel \sigma_k$, along with "**S**ibling **P**aths" from each corresponding leaf to the root of the tree. (See below for more details.)

The parameters (d, k), as we shall see, determine how secure the signature scheme is. (We'll use a fixed value $\ell = 256$ for this exercise.) Intuitively, you can think of the leaf index j of a path SP_j as a compact (hashed) representation of the message m. Releasing a pre-image for the leaf of SP_j signs this compact representation. Larger k means a larger number of compact representations of m. The result is a stronger, i.e., harder-to-forge, resulting signature on m as a whole. The larger d is, the larger the number of compact representations there are in the signature on m and, similarly, the stronger this signature is. The general outline of this scheme is also depicted in Figure 3.

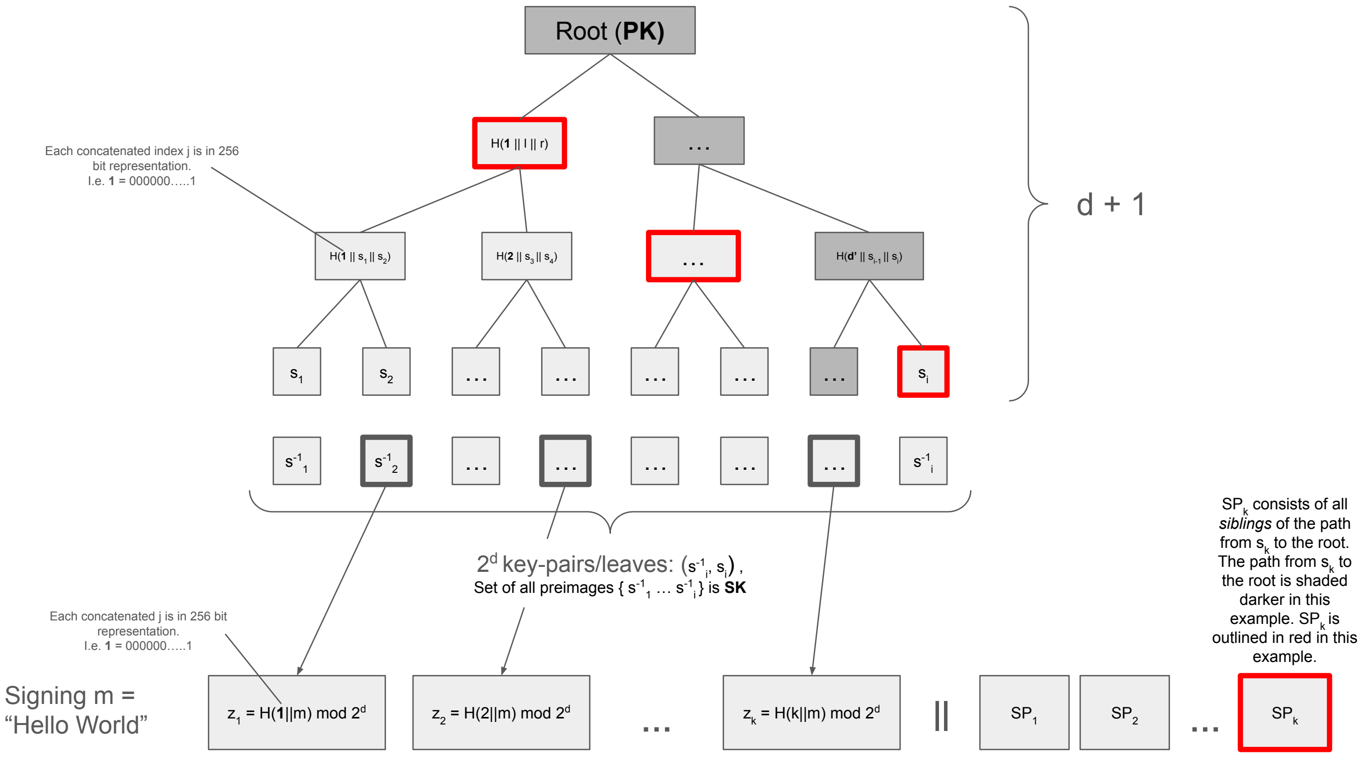

Figure 3: Signature Scheme for Problem 3.

Problem parts:

1. *Signature scheme construction:* Implement the algorithms (KeyGen, Path, Sign) for this signature scheme so that it runs for any $d, k \in [1, 20]$. Specifically, you should complete the following three functions in `signature.py` in the order that follows. Here are the necessary details:

   - KeyGen: This function should as a first step generate 2^d preimage / image pairs (s_i^{-1}, s_i). It generates them from a secret (pseudorandom) seed ρ (provided as input to your code, to make it easier for us to verify your results). We are providing a function to generate the full set of such pairs: KeyPairGen in `signature.py`. It takes as input a parameter d and the seed ρ, which takes the form of an integer, and returns 2^d preimage / image pairs.[1]
     Given leaf values, KeyGen should build a Merkle tree over the set of values $\{s_i\}$, i.e., these values constitute the leaves of the tree. As in Problem 2, you will use `SHA-256` as the hash function. For a non-leaf node A in the Merkle tree at depth $d' \in [0, 1, \ldots, d]$, its value will be calculated as SHA-256$(i \parallel l \parallel r)$, where $i \in [0, 1, \ldots, 2^{d'})$ is the 256-bit binary representation of the index of A among the nodes at depth d', l, r are 64-byte hexadecimal strings representing the values of node A's left child and right child respectively.
     Your code for KeyGen should take as input the parameter d and a random seed r. It will output the hexadecimal string pk.
   - Path: This function returns the path SP_j of a given leaf index z_j. We now precisely define SP_j. The path in the Merkle tree from s_{z_j} to the root has $d + 1$ nodes, call them $A_{j0}, ..., A_{jd}$, with $A_{j0} = s_{z_j}$ being the leaf and $A_{jd} = pk$ being the root. We know that A_{ji} is a child of $A_{j(i+1)}$. For Verify to calculate the next node $A_{j(i+1)}$ recursively, it must know the other child of $A_{j(i+1)}$, which is the sibling node of A_{ji}. We denote this node by sib_{ji}, so that value of $A_{j(i+1)} = H(b \parallel A_{ji} \parallel sib_{ji})$ where b is the index of $A_{j(i+1)}$. SP_j is now defined as $SP_j = sib_{j0} \parallel \ldots \parallel sib_{j(d-1)}$.

[1] Note that KeyPairGen is *not cryptographically secure* and neither it nor its components should be used to build secure applications. We've designed it for this exercise just to keep things simple.

Your code should take the input $cur = z_j$, and output SP_j.

- Sign: Continue to use SHA-256 as the hash function here. The function Sign should take as input an arbitrary string m, and a signature security parameter k. It will generate and output a signature $sig = \sigma \parallel paths$. Here σ is as described above (with each σ_j in hexadecimal notation), while $paths = SP_1 \parallel \ldots \parallel SP_k$ denotes the sequence of nodes that are needed to verify (using the public key) that each s_{z_j} is a leaf of the tree.

You can use the provided `test.py` to test your code. Again, the provided checks are just a sanity check and you should test your own code.
Tip: You can use `format(j, "b").zfill(256)` in `python` to convert an integer `j` to a 256-bit binary string.

2. *Signature forgery:* For poor choices of (d, k), an adversary that is given just one signature can forge a second one. Run your scheme with $d = 10$, $k = 2$ and $r = 2023$. Sign a message m of your choice, yielding signature σ. Find a message $m' \neq m$ such that σ is a valid signature for m'. (*Extra credit*: Provide m, m' that are grammatically correct English sentences. Mention in `double.txt` whether you have found one.) Output your m and m' in `forgery.txt`, each in one line.

3. *Double signatures:* Suppose $k = 1$, and a signer has signed 200 messages $m_1, \ldots, m_{200}$. There is some probability that there exists among these messages a pair m_i, m_j, with $m_i \neq m_j$, such that both messages have identical signatures. How large does d have to be to ensure that this probability is less than 50%? (You may assume a predetermined set of messages. The probability here is then over invocations of KeyGen.) Explain your reasoning in a text file `double.txt`, and include `double.txt` in your submission (can also be .docx, .pdf, or an image).

Problem 4: Survey

(5 points). How long did it take you to complete this assignment (in hours)? Do you feel that this homework was too easy, too challenging, or just right? Please submit your answers in `survey.txt` (a one liner is enough).

Submission Instructions

Directory Structure:

```
hw1-<NetID>
├── p1
│   ├── coin.txt
│   ├── forged-watermark.txt
│   └── generate.py
├── p2
│   └── merkle.py
├── p3
│   ├── signature.py
│   ├── forgery.txt
│   └── double.txt
└── p4
    └── survey.txt
```

Your submission should be a zip'd directory, following the above structure exactly. Please upload the .zip file on CMSX.