1.

**Authenticated Data Structures.** You are designing SecureBox, an authenticated online file storage

system. For simplicity, there is only a single folder. Users must be able to add, edit, delete, and

retrieve files, and to list the folder contents. When a user retrieves a file, SecureBox must provide a

proof that the file hasn’t been tampered with since its last update. If a file with the given name

doesn’t exist, the server must report that — again with a proof.

We want to minimize the size of these proofs, the time complexity of verifying them, and the size of

the digest that the user must store between operations. (Naturally, to be able to verify proofs, users

must at all times store some nonzero amount of state derived from the folder contents. Other than

this digest the user has no memory of the contents of the files she added.)

Here’s a naive approach. The user’s digest is a hash of the entire folder contents, and proofs are

copies of the entire folder contents. This results in a small digest but large proofs and long verification

times. Besides, before executing add/delete/edit operations, the user must retrieve the entire folder

so that she can recompute the digest.

Alternatively, the digest could consist of a separate hash for each file, and each file would be its own

proof. The downside of this approach is that it requires digest space that is linear in the number of

files in the system.

Can you devise a protocol where proof size, verification time, and digest size are all sublinear? You

might need a sub‐protocol that involves some amount of two‐way communication for the user to be

able to update her digest when she executes and add, delete, or edit.

Hint: use the Merkle tree idea from Section 1.2.

2.

**Birthday Attack. ​**

Let H be an ideal hash function that produces an n‐bit output. By ideal, we mean

that as far as we can tell, each hash value is independent and uniformly distributed in {0,1}

n

. Trivially,

49we can go through 2

n

+ 1 different values and we are guaranteed to find a collision. If we're

constrained for space, we can just store 1 input‐output pair and keep trying new inputs until we hit

the same output again. This has time complexity O(2

n

), but has O(1) space complexity. Alternatively,

we could compute the hashes of about O(2

n/2

) different inputs and store all the input‐output pairs. As

we saw in the text, there’s a good chance that some two of those outputs would collide (the “birthday

paradox”). This shows that we can achieve a time‐space trade‐off: O(2

n/2

) time and O(2

n/2

) space.

1. (Easy) Show that the time‐space trade‐off is parameterizable: we can achieve any space

complexity between O(1) and O(2

n/2

) with a corresponding decrease in time complexity.

2. (Very hard) Is there an attack for which the product of time and space complexity is o(2

n

)?

[Recall the

little oh notation

.]

3.

**Hash function properties** (again). Let H be a hash function that is both hiding and puzzle‐friendly.

Consider G(z) = H(z) ǁ z

last where z

last represents the last bit of z. Show that G is puzzle‐friendly but not

hiding.

4.

**Randomness​**

. In ScroogeCoin, suppose Mallory tries generating (sk, pk) pairs until her secret key

matches someone else’s. What will she be able to do? How long will it take before she succeeds, on

average? What if Alice’s random number generator has a bug and her key generation procedure

produces only 1,000 distinct pairs?

1. To design a protocol where proof size, verification time, and digest size are all sublinear, we can use the Merkle tree idea. In this protocol, the digest is the hash of the root of the Merkle tree. Each leaf node in the Merkle tree corresponds to a file in the folder, and its value is the hash of the file content. The intermediate nodes are the hash of the concatenation of their child nodes.

When a user retrieves a file, SecureBox provides a proof by sending the hash values of the Merkle tree nodes on the path from the leaf node to the root. The user can verify the proof by checking that the computed hash of the file matches the hash value of the leaf node and that the hash values of the Merkle tree nodes on the path from the leaf node to the root match the hash values received from the server.

When a user adds, deletes, or edits a file, the server sends the hash values of the Merkle tree nodes that need to be updated, and the user updates her digest accordingly. To minimize the size of the proof, the server can send only the hash values that are different from the user's digest.

This protocol has sublinear proof size, verification time, and digest size. The proof size is logarithmic in the number of files, and the verification time and digest size are proportional to the depth of the Merkle tree, which is also logarithmic in the number of files.

1. The time-space trade-off for finding collisions in an n-bit hash function is parameterizable. We can achieve any space complexity between O(1) and O(2^(n/2)) with a corresponding decrease in time complexity.

To achieve a particular space complexity, we can compute the hashes of about O(2^(n/2 - k)) different inputs, where k is the number of bits we want to use for storage. We store all the input-output pairs and use them to find collisions. The time complexity is O(2^(n/2 - k)), and the space complexity is O(2^k).

1. To show that G(z) is puzzle-friendly but not hiding, we need to show that:

* Puzzle-friendliness: Given G(z) and a random string r, it is hard to find a z' such that G(z') = G(z) and H(z') = H(z) without computing H(z) and the Merkle path for z in the Merkle tree rooted at H(z).
* Non-hiding: Given G(z), it is easy to find z' such that G(z') = G(z) by trying all possible values of z' that have the same last bit as z.

To show puzzle-friendliness, assume that we have a z and r such that G(z) = G(z') and H(z') = H(z) for some z'. We want to find z' without computing H(z) and the Merkle path for z.

Since G(z) = G(z'), we have H(z) ǁ z\_last = H(z') ǁ z'\_last. Therefore, H(z) = H(z') and z\_last = z'\_last. We can now construct a new string z'' by concatenating the first n-1 bits of z' and the last bit of z. We have H(z'') = H(z), and the Merkle path for z' in the Merkle tree rooted at H(z') is the same as the Merkle path for z'' in the Merkle tree rooted at H(z). Therefore, we can use the Merkle path for z in the Merkle tree rooted at H(z) to compute H(z') and the Merkle path for z' in

1. If Mallory generates (sk, pk) pairs until her secret key matches someone else's, then she will effectively be trying to perform a birthday attack on the secret key space of ScroogeCoin. Given that ScroogeCoin uses 256-bit secret keys, the probability of Mallory generating the same secret key as someone else in the network is extremely small. Specifically, the probability of Mallory generating a key collision with just one other person in the network is approximately 2^(-128), which is a very small number. Therefore, it is highly unlikely that Mallory will be able to generate the same secret key as someone else in the network.

Assuming that Alice's random number generator is functioning correctly, the probability of generating the same (sk, pk) pair twice is negligible. Specifically, if Alice generates 1,000 distinct pairs, then the probability of generating the same pair twice is approximately 1/((2^256)^1000), which is essentially zero. However, if Alice's random number generator has a bug and only produces 1,000 distinct pairs, then an attacker could simply generate all 1,000 pairs and check them against the network's public keys until a match is found. The expected number of trials required for this attack is simply the number of distinct pairs divided by the number of unique public keys in the network. If the network has, for example, 10,000 unique public keys, then the expected number of trials required for a successful attack is 1,000/10,000 = 0.1 trials on average. In practice, the actual number of trials required could be higher or lower, depending on the distribution of the network's public keys.