Part 1 Sample Solutions

There are many ways to implement Part 1. Here, we explain two secure designs.

Design 1

One approach is to maintain a directory of ids and keys on the server. The client stores each file under a random id. A directory listing is kept on the server (encrypted) that maps the filename to the corresponding ID and the keys used to encrypt and MAC that file.

The client needs to create and securely store the keys it will use to maintain the directory on the StorageServer. During client initialization, the client generates two symmetric keys, each 16 bytes long, using `get_random_bytes()`. The client gets its public key from the PublicKeyServer, encrypts the symmetric keys using its public key, and then signs the resulting ciphertext using its private key. It then puts the ciphertext and signature to the StorageServer under the id `<username>/dir_keys`. In other words, we store $E_{K_A}(k_e, k_a), \text{Sign}_{K_A^{-1}}(E_{K_A}(k_e, k_a))$ on the storage server, where k_e, k_a are symmetric keys for encrypting the directory listing.

The client can retrieve k_e, k_a by getting this data from the storage server, verifying the signature, and decrypting the ciphertext.

Apart from `<username>/dir_keys`, everything else on the storage server will be encrypted using using authenticated encryption, and specifically, Encrypt-then-Mac. In particular, if we want to store data D at id I, we'll store

$$AE_{k_1,k_2}(D, I) = E_{k_1}(D)||\text{MAC}_{k_2}(I, E_{k_1}(D)).$$

Here E_{k_1} represents AES-CBC encryption with AES key k_1 and with a random 16-byte IV (generated fresh with `get_random_bytes()` each time). The IV is prepended to the ciphertext. The IV+ciphertext is MAC'd using SHA256-HMAC under key k_2, and the MAC tag is appended. Including the id in the input to the MAC ensures that an adversary cannot swap around encrypted values between different id's.

The directory listing contains a mapping that maps each filename to (r, k_1, k_2), where r is a 16-byte random id where the file's contents are stored and k_1, k_2 are keys for encrypting the contents. The directory listing is stored at `<username>/directory`, encrypted under keys k_e, k_a (defined above) using authenticated encryption.

A file is stored at `<username>/files/`r (where r is the random id retrieved from the directory listing), encrypted under keys k_1, k_2 (retrieved from the directory listing) using authenticated encryption. Each time the client uploads a new file, it picks new random 16-byte values

r, k_1, k_2, adds an entry to the directory listing, encrypts the file contents, and stores them on the storage server.

We use the utility functions `to_json_string` and `from_json_string` as necessary to convert to and from strings (for encryption, putting to the StorageServer, etc.).

The client catches all exceptions thrown by `from_json_string`, and re-raises them as `IntegrityError`. Any error here means that the result we got from the server wasn't valid JSON, which implies that something was modified by the server so we should raise an `IntegrityError` to signal this.

Design 2

The directory approach takes $O(n)$ space but also $O(n)$ time and bandwidth per update. While you don't need to worry about performance, in practice one might want something with a lower cost per update. Design 2 is an alternative solution that achieves $O(1)$ updates.

Design 2 uses four keys: the public key K_A given to us, a symmetric encryption key k_e, a symmetric MAC key k_a, and a symmetric key k_n for name confidentiality. The symmetric keys k_e, k_a, k_n are stored on the server, encrypted and signed using the public key as in Design 1. All other data is stored encrypted using Encrypt-then-MAC, using k_e and k_a, respectively.

On the server we maintain three types of data: client information nodes, data nodes, and metadata nodes.

Client information nodes store k_e, k_a, k_n in encrypted form. `information/<username>` stores $E_{K_A}(k_e, k_a, k_n)$, the encryption of the symmetric keys under our public key. Also, `information/<username>/signature` stores a signature computed on the previous ciphertext. We always check that the signature verifies before trying to decrypt the ciphertext.

A *data node* stores the encrypted contents of a file, namely $E_{k_e}(v)$ where v is the contents of the file. The tricky part is: where do we store this on the storage server? In particular, what id can we use, without compromising the confidentiality of the filename? In this design, we use the id `<username>/`r where $r = F_{k_n}(\text{filename})$. The choice of the function F is discussed below, but one reasonable choice is $F_{k_n}(n) = \text{SHA256-HMAC}_{k_n}(n)$.

The *metadata nodes* contain the MAC of the information stored in the corresponding data node. In particular, the metadata node for a file contains $\text{MAC}_{k_a}(C, n)$ where $C = E_{k_e}(v)$ is the ciphertext stored in the corresponding data node, and n is the name of the file. We use `<username>/metadata/`r as the id for the metadata node. Including the filename in the input to the MAC prevents a malicious storage server from swapping files between ids (see the Test Case Explanations below for more on this attack).

Encryption. We encrypt all data with AES in CBC mode. The symmetric keys k_e, k_a, k_n are chosen randomly with `get_random_bytes()` during client initialization and stored in the client information node. The IVs are always chosen fresh each time with `get_random_bytes()`,

and then prepended to the ciphertext. We use SHA256-HMAC as our MAC. We always verify the MAC before decrypting the messages.

Choice of F. All that remains is to choose the function F. It needs to satisfy three properties: (1) it must be deterministic (the same filename always yields the same id); (2) it must depend on a key (otherwise dictionary attacks will reveal the filename, for some files); and, (3) the value $F_k(n)$ should not reveal anything about the filename n. To achieve property (3), our approach will be to try to essentially mimic Design 1: we'll try to ensure that $F_k(n)$ looks like a random id, that is random and independent for each different n.

There are several ways to achieve these properties. Here are a few constructions that work:

- $F_k(n) = \text{SHA256-HMAC}_k(n)$. In general, there's no guarantee that a MAC will necessarily provide confidentiality for its input—but some MACs do happen to provide that property. HMAC is one. In particular, HMAC is believed to be pseudorandom function (PRF), which means it behaves like a random function (a function that for each input, outputs a random bit-string that's uniformly distributed and independent of its output for all other inputs). Thus, $F_k(n)$ is like a random string for each filename n, so this effectively has the same properties as Design 1, where we pick a truly random id for each filename.
- $F_k(n) = H(n||k)$. This also seems to achieve similar properties, if H is a secure cryptographic hash function (e.g., SHA256).
- $F_k(n) = \text{AES-ECB}(H(n))$, where H is a cryptographic hash. Why is this secure? Intuitively, because H is collision-free, each file will have a different value of $H(n)$. And, because AES is a pseudorandom permutation (see the lecture notes), if each input to the AES block cipher is different, then all of the outputs look like different random values, so they reveal nothing about the name n. Or, at a very intuitive level, the problem with ECB arises when you have two plaintext blocks (possibly in two different messages) that are equal; ECB leaks this fact. However, if the plaintext blocks come from a cryptographic hash function, the probability that two plaintext blocks are equal is extremely low.
- $F_k(n) = \text{AES-CBC}(H(n))$, where we use a constant (say, all-zeros) IV with AES-CBC mode, also works, for similar reasons.

These schemes are tricky. How could you have come up with them? The basic approach is to enumerate the properties we need, then try possible combinations of the building blocks we've given you to look for one that has all the necessary properties.

However, we think Design 1 is more realistic for you to come up with, given the tools we've taught you at this point. We wouldn't necessarily expect you to be able to come up with Design 2. We just wanted to show you another design that would work.

Some choices of F that don't work:

- $F_k(n) = H(n)$ is not secure: because it is an unkeyed function, anyone can compute

the function. This allows dictionary attacks: if the attacker has a set of candidates for the filename, the attacker can hash all of those candidate names and see which hash digest matches the id stored on the file server, thereby learning the filename. Because human-chosen filenames will often be guessable (low entropy), an attacker can probably learn many filenames in this way.

- $F_k(n) = \text{AES-ECB}_k(n)$. This is not secure: it doesn't protect the confidentiality of the filename n. For instance, storing something under the filename $n_1 =$ `cs161 projects are super easy` and something else under the filename $n_2 =$ `cs161 projects are super hard` would cause the corresponding id's to match in their first 16 bytes, which reveals that the names n_1, n_2 start with the same 16 bytes.
- $F_k(n) = \text{AES-CBC}_k(n)$, with a constant (e.g., all-zeros) IV. This has a similar problem as AES-ECB mode: if two filenames that start with the same 16 bytes, an adversary will be able to detect this, because the corresponding ciphertexts will start with the same 16 bytes.
- $F_k(n) = \text{RSA-Encrypt}(n)$ doesn't work, because encryption is non-deterministic.
- $F_k(n) = \text{Sign}(n)$ doesn't work, because the existence of the Verify() function allows dictionary attacks. An attacker who suspects the name might be n_0 can check his guess by seeing whether $\text{Verify}_{K_A}(n, id)$ is true or not; if Verify returns true, then the attacker knows his guess was correct. Thus, if an attacker can narrow down the set of possible filenames to a few million possibilities, the attacker can identify which one of those candidates is correct.