Part 3 Sample Design

We augment our design from Part 2 to allow efficient updates. Our efficient update scheme is inspired by Merkle trees. As before, when we encrypt something we always perform authenticated encryption by first encrypting the value and then computing a MAC on the ciphertext. The MAC includes the ID at which the ciphertext is stored.

To initialize a client, we create three master keys (as in the sample design for Part 2). We store these keys on the server encrypted and signed using our public/private key pair.

To create a file initially, a client creates two *new* keys (k'_e, k'_a). It then stores the following two files on the storage server:

- The *data node* points to the root of a tree data-structure (described later) that stores the file contents and enables efficient updates.
- The *key node* contains the two keys k'_e and k'_a along with a pointer to the data node, all encrypted and authenticated under the user's symmetric keys (as in the solutions for Part 2).

To perform efficient updates, we use a tree-based approach. When a file is initially uploaded, we divide the file into chunks of size 512 bytes each, and then create a binary tree over the chunks. Each node of the tree contains several pieces of information:

1. Pointers to the left and right sub-trees, or None to indicate a leaf.
2. A cryptographic hash of all the chunks below this node.
3. The total length of all the chunks below this node.

We encrypt and authenticate each node of the tree using the keys k'_e and k'_a, and store them at random IDs on the server. Before encrypting each node, we first convert it into a dictionary that contains the values above, and then serialize it using `util.to_json_string()`.

To update a file, we first fetch the root node and check whether the update would result in a larger file than currently stored on the server. If so, we simply re-upload the entire file; else, we apply the following procedure for efficiently updating the file.

We describe our update procedure recursively. We first compare the hash of the new file with the hash stored at the root node. If the hashes are equal, then we have no more work to do. If they differ, we update the node with the new hash value, fetch the node's left and right

children, and then split our file into two pieces according to the lengths of the sub-trees at the two children. Next, we recursively call update on the children nodes. When we reach a leaf node, we replace the corresponding chunk with the new data if the hash does not match.

To optimize the performance of our scheme, we store a copy of the tree locally whenever a file is uploaded or downloaded. Before updating a file, we compare the root node of the tree on the server with the root of the local copy, if it exists. If the roots are the same, then we compare the file against the local tree instead of fetching the nodes from the server. Otherwise, we compare the file with the tree on the server, and update the local tree accordingly.

Our scheme is efficient even in the case where Alice uploads a file F, shares it with Bob, and Bob makes a small update, even if Bob has not downloaded the file ahead of time.

To download a file, we walk the tree post-order. At each leaf, we verify the integrity of the encrypted chunk, decrypt it, and then return the chunk if the hash at the leaf is valid. At each internal node, we similarly verify the integrity of the node, decrypt it, and then return the chunks in the left and right sub-trees if the hash at the internal node is valid.

Part 1 Tests

CheckDifferentialAttacks. Check if there are any values stored on the server that leak any bits of the message. We do this using a technique similar to differential cryptanalysis, which will catch any submissions that store the plaintext in a non-standard manner (e.g., compressed, pickled, ROT13d, etc).

CheckEncryptionMACDifferentKey. MAC keys and encryption keys should be different, and this test case checks just that.

CheckKeyNotStoredPlaintext. The point of creating an encryption key is to hide the content of a message. This test verifies that the encryption keys are never stored, in plaintext, on the server.

CheckPaddingOracle. A padding oracle attack is a chosen-ciphertext attack: if the attacker can convince the client to decrypt (tampered) ciphertexts chosen by the attacker, then a padding oracle attack allows the attacker to decrypt an encrypted message. In this attack, the attacker modifies the ciphertext in some cleverly chosen fashion, asks the client to decrypt it, and then observes the decryption process caused an invalid-padding error. If the attacker can observe whether such an error occurred, then this leaks partial information; after repeating this many times, an attacker can piece together all of these clues to deduce what the original message must have been. These attacks are very powerful and are subtle if you don't know about them.

Failure of this test case typically indicates that you used encryption without authentication, or that you decrypted some ciphertext without checking the MAC or before verifying the validity of the MAC. Defend against padding oracle attacks by always verifying the MAC *before* decrypting message contents.

CheckPlainttextMAC. MACs provide integrity, but they don't promise to provide confidentiality for their input. Therefore, you should generally avoid including plaintext data directly as part of the input to the MAC, if the MAC is not encrypted, as this might leak plaintext bits. Also, when computing a MAC solely on the file contents, the same file contents will always yield the same MAC.

CreateFakeKey. It is important to bind the name of a file to its contents. Otherwise, if a user creates files A and B, a malicious server could swap the names and have downloads for A return the content of B and vice versa.

RandomSwapValues. This test randomly permutes the IDs and values stored on the storage server. For instance, if you have stored {a:b, c:d} on the server, this test might change it to {a:d, c:b}. Then, it will try downloading files. If download() returns incorrect data, then the test case fails. Failing this test case tends to indicate that you are missing a MAC, or you have not bound the MAC'ed value to the filename it is associated with. The fix is generally to include the filename or ID in the input to the MAC, as we did in Design 1 and Design 2.

EncryptionHasNoncesBlackBox and EncryptionHasNonces. Verify that encrypted messages are correctly randomized, using IVs or nonces.

CBCIVsNotReused. Verifies that you don't reuse the same IV more than once for CBC mode encryption per key. Such reuse is insecure and can leak partial information about the plaintexts: the same contents will encrypt to the same ciphertext, and if two plaintexts start with the same 16 bytes, then an adversary can detect this (by noticing that the ciphertexts start with the same 16 bytes). Use a fresh, random IV for each CBC encryption you do. Generate the IV using `get_random_bytes()`.

KeysAreChosenFresh. Keys should be chosen randomly using `get_random_bytes()` and should not be generated via any other means. In particular, hashing the private key is poor practice and makes rotating keys difficult if the private key were ever to be compromised.

NameIsNotFromECB. IDs should not be generated by encrypting the filename with ECB mode or CBC/CTR mode with constant IV, as this leaks partial information. In particular, if two filenames start with the same 16 bytes, then the adversary can detect this (by noticing that the ciphertexts start with the same 16 bytes). Also, these modes allow dictionary attacks: an adversary who can convince you to upload files under a name of the attacker's choice will be able to exploit the properties of ECB/CBC/CTR to figure out the filename of your secret file, if the attacker can identify a list of candidate names for the secret file. Finally, CTR mode with a constant IV is especially bad: it is vulnerable to the attacks on pad reuse with the one-time pad.

NameIsNotFromHash. Verifies that IDs are not generated by hashing the filename. That has the same problems outlined above.

SmallCTRUsed. When using CTR mode, the counter should be at least 64 bits, to prevent repetition of the counter value.

CTRUsesUniqueValues. When using CTR mode, the counter values used should be unique. Give either a unique prefix, suffix, or initial value.

StringNotPlaintext. The filename should not directly appear in plaintext on the server. Similarly, the contents of the file should not appear in plaintext on the server.

StringNotPlaintextBoth. You fail this test if both the name and the contents of the file appear in plaintext on the server.

CanEncryptionBeOracle. This test checks that you are not doing anything funky with ciphertexts. We don't take off any points for it, but if you have passed all the functionality tests, it is a warning to double-check your design as the approach seems a bit suspicious. The test checks that it is possible to replace the specifics of `symmetric_encrypt()` with a function that returns random bytes. If your test fails, it means either (a) you fail one of the functionality tests, or (b) when we replace `symmetric_encrypt()` in this way, you fail a functionality test. If you pass all the functionality tests but fail this test, review your design to see what you are doing with the crypto. If you failed some of the functionality tests, you can ignore this test.

BrokenTestBadCrypto and WeakTestBadCrypto. You fail this test if you use insecure cryptographic algorithms. DES and RC4 are broken; you should not use those algorithms. Encrypted messages stored with these algorithms can be decrypted without the key. MD2, MD5, and SHA-1 are cryptographically weak hash functions and should not be used. They shouldn't be used for HMAC-based schemes, either: though no attack is known that can forge MAC tags for these schemes, their use to build a MAC is discouraged. Instead, use strong algorithms like AES, SHA256, and SHA256-HMAC.

NotIntegrityError. Something other than an `IntegrityError` was raised during testing. Our tests change data to unexpected values, and your code must handle those values by the spec.

CheckIntegrityPart1 Verifies that data stored on the server is integrity-protected (i.e., either signed or MAC'd).

Part 2 Tests

KeysAreNotReused This test ensures that when different files are shared with different people, they do not see the same keys. That is, Alice uploads files F and G and shares F with Bob and G with Carol. If Bob and Carol share any keys then you fail this test case.

StayAfterRevoke The following four tests will have similar formats. In each, an adversary will pretend to have valid access to the file. If, when another user is revoked, the adversary tricks the file's owner into giving them access, fail.

This test ensures that a revoked user cannot use known values to trick the file's owner into thinking the revoked user has access to the file, so that the revoked user gains access to the file when a different user is subsequently revoked.

LieAboutBeingShared An adversary attempts to use public information to convince a file's owner that the file was shared with the adversary, so that the adversary gains access to the file when another user is revoked.

LieAboutTreeStructure An adversary attempts to change the tree-structure of the file. A uploads a file, and shares A→B, A→C, B→D. Now D attempts to change the tree structure so that it becomes A→B, A→C, C→D. That way, when B is revoked, D still gets access.

ShareTiedToFile An adversary who received a valid share for file A attempts to convice the file owner that they have access to file B, so that the adversary gains access to file B when another user with access to B is revoked.

NoReadAfterRevoke This test verifies that it is not possible for a client to remember encryption keys so that after they are revoked, they can still read the modified file on the server by re-using the old keys they remembered.

OutOfOrderSharing This test checks that a client cannot access files it should not be able to access if it receives shares out of order.

NoReceiveAfterRevoke This test checks that if sharing is revoked before a client receives it, the client cannot receive it anymore.

NoRollbackPartialPuts This test makes sure that updates are always consistent even if an adversary tries to block some writes from occurring. That is, if Alice uploads (F, V) and then (F, V') where V' modifies many portions of F, this test checks if it is possible to stop some writes so that the result of downloading F returns some contents intermediate between V and V'.

NoRollbackPartialPuts2 Simply another way of testing NoRollbackPartialPuts.

NoSharedKeysAfterRevoke After a user's access to a file has been revoked, the user should no longer share any keys in use with the parent. That is, if Alice shares a file with Bob, then after Alice revokes Bob's access, Alice should never try to use a key that Bob knows, and Bob should never try to use a key that Alice knows.

RevokedDataIsReEncrypted Once a user's access to a file has been revoked, the original file must be re-encrypted under a new key. This test verifies that this occurrs and all modifications to the file are done under the new encryption key.

RevokedDataIsReEncrypted2 A different way of testing RevokedDataIsReEncrypted.

ShareMessageDoesntLeakKeys Share messages should be encrypted and keys should not be sent in plaintext in these share messages.

ShareMessagesAreSigned When Alice shares a file with Bob, it should not be possible for Mallory to switch out the message with one of her own so that Bob instead receives a file owned by Malory instead of Alice.

ShareWithSameName If Alice shares a file she calls F with Carol (who calls it F1), and then Bob shares a file he calls F with Carol (who calls it F2), then it should still be possible for Alice and Bob to revoke Carol's access of their own files without breaking Carol's client.

SharingMessageCantBeReplayed This test verifies that a user can't replay a share message to herself after having been revoked in order to re-grant herself access. That is, if Alice shares a file with Bob with share message S, and then later revokes Bob's access, it should not be possible for Bob to call `receive_share` on himself with S again and re-grant himself access.

SharingMessageCantBeReplayed2 Another way of testing SharingMessageCantBeReplayed.

SharingMessageIsEncrypted This test verifies that sharing a file from Alice to Bob does not allow a man in the middle to learn the name of the file being shared.

SilentDropPuts Another way of testing NoRollbackPartialPuts.

CheckIntegrityPart2 Verifies that data stored on the server is integrity-protected (i.e., either signed or MAC'd).

Part 3 Tests

ApplyEditsToNewFile If your Q3 solution involves a log to make updates efficient, this test case checks whether it is possible to add fake entries to the log, leading to invalid files being uploaded. That is, we upload on the server (F, V) and (G, W), then we update F to V' with an update of D; this test checks if it is possible to apply D to G to obtain some W'.

EfficientPutHasHashes This test tries to make sure that it is not possible to remove selective edits from occurring during efficient updates. That is, we first upload (F, V) and then (F, V') and finally (F, V''). We then randomly discard edits made during the two updates to see if it is possible that downloading F will result in a file that is not one of V, V', or V''.

PairwiseSwapValues Similar to RandomSwapValues in part one, but this time places a single multi-chunk file on the server, and attempts to swap values between different IDs on the server. Example Failure Condition: If the client uses a Merkle Tree, and doesn't ensure the integrity of the top node.

NoRollbackPartialPuts This test makes sure that updates are always consistent even if an adversary tries to block some writes from occurring. That is, if Alice uploads (F, V) and then (F, V') where V' modifies many portions of F, this test checks if it is possible to stop some writes so that the result of downloading F returns some contents intermediate between V and V'.

NoRollbackPartialPuts2 Simply another way of testing NoRollbackPartialPuts.

SilentDropPuts Another way of testing NoRollbackPartialPuts.