Dropbox Gearup

Spring 2023

Note: some requirements and due dates will differ for Spring 2024

Goals

Idea: design an end-to-end encrypted file sharing service

Learn how to design a secure system using the cryptography and security tools we've learned so far!

- Thinking about how to design a system securely
- Iterate on your design after receiving feedback
- Think about attacking your design based on a threat model

Goals

Goal: client for end-to-end encrypted file sharing service

What you have

- Crypto library
- Some insecure data storage
- Threat model (what kinds of attacks to defend against)

What you'll build

- Client API for storing data securely on insecure data storage

You get to figure out how to use the provided crypto operations to accomplish this goal!

How you'll do this

- Now: Design document
  - Think carefully about how you'll implement the requirements
  - How you'll store data, how you'll use crypto to secure it
  - ~4 pages + diagrams
  - See handout for details
  - ⇒ Meet with TAs afterward for feedback
- Implementation (Due Monday, May 8)
  - Submit your code + final design document

Remember: the big part is about your design!

How you'll do this

- Now: Design document
  - Think carefully about how you'll implement the requirements
  - How you'll store data, how you'll use crypto to secure it
  - ~4 pages + diagrams
  - See handout for details

  ⇒ Meet with TAs afterward for feedback

- Implementation ~~(Due Monday, May 8)~~
  - Submit your code + final design document

What the client looks like

```
# Make a user
client.create_user("usr", "pswd")

#             . . .

# Log in
u = client.authenticate_user("usr", "pswd")  # Returns a User object

# Make some data to upload
data_to_upload = b'testing data'

# Upload it
u.upload_file("file1", data_to_be_uploaded)

# Download it again
downloaded_data = u.download_file("file1")
assert downloaded_data == data_to_be_uploaded
```

YOU IMPLEMENT THESE.

The Client API: what you'll implement

Your implementation: some functions that implement the client

- User operations: `create_user`, `authenticate_user`
- File operations: `upload_file`, `download_file`, `append_file`
- Sharing operations: `share_file`, `receive_file`, `revoke_file`

Your goal: implement client while preserving *confidentiality* and *integrity* *in an insecure environment*

So what's the environment?

The Wiki

The definitive source for everything all specifications is the wiki:

https://brown-csci1660.github.io/dropbox-wiki

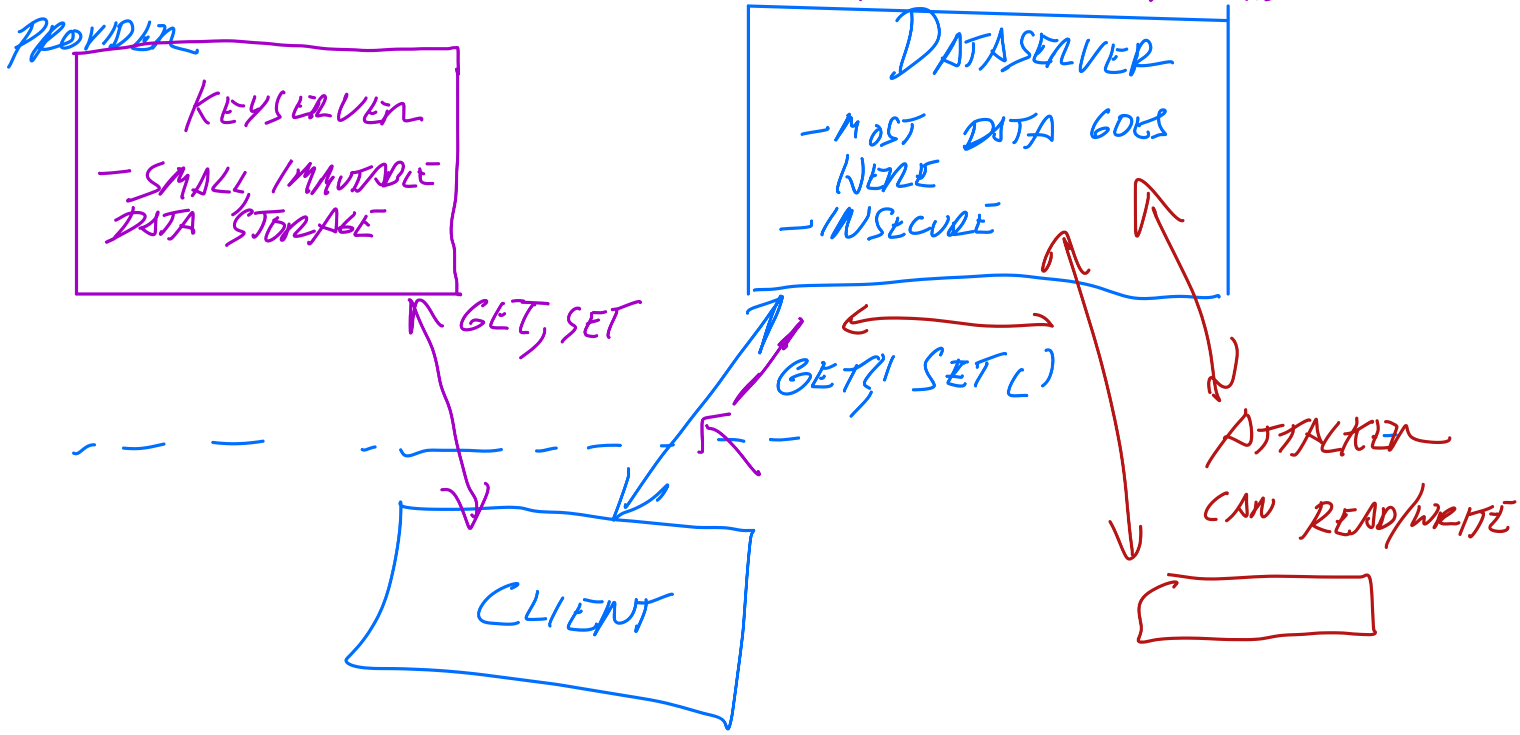
Look here for:

- Descriptions of each API function and requirements
- Detailed description of threat model/environment
- Documentation for all support code

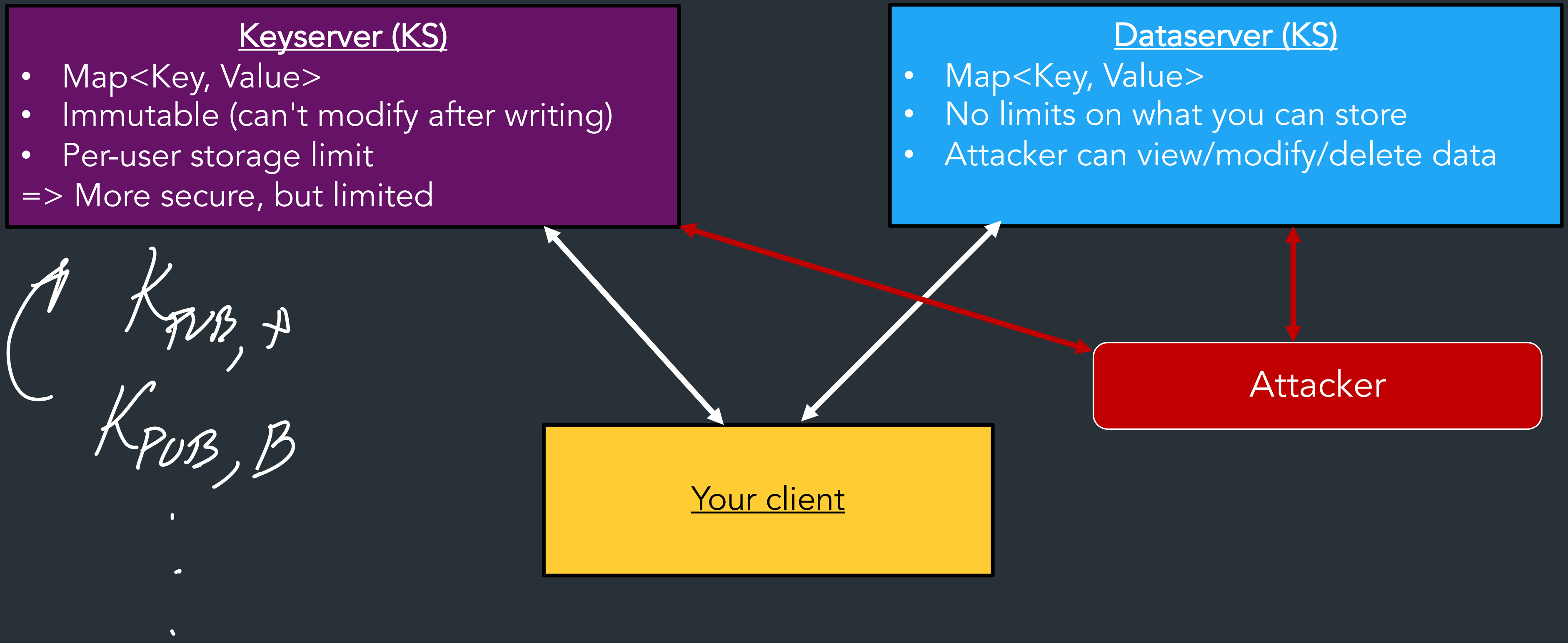
For implementation notes and container setup, see the setup guide:
https://hackmd.io/@cs1660/dropbox-setup-guide

THE SETUP (INITIAL VIEW)

ALL STATE MUST BE IN ONE OF THESE.



System Overview



Dataserver

16 BYTE VALUE (COULD JUST HASH A STRING & TRUNCATE)

- Map<Memloc, Data>
  - Memloc: 16 bytes
  - Data: `bytes`
- Operations: Set(), Get()

- Most data will be stored here
- Attacker has full access
  - What could an attacker read? => Threat to confidentiality
  - What happens if an attacker changes something? => Threat to Integrity

Keyserver

- Public, immutable key-value store
- Map<key_name, data>
  - key_name: any string ("key-alice")
  - Data: ~~[illegible]s~~ ANY PUBLIC KEY
- Operations: Get(key), Set(key, value)

- Designed for storing public keys
- Immutable: upload once, can't modify again (but neither can attacker)
- Number of keys per user must be constant

  => Can't grow with number of files, operations, etc.

Threat model: What the attacker can do

- Read/write/modify anything on Dataserver
- Read on the Keyserver (but not modify)
- Can create users/use client API, just like any normal user
- Knows how your client works
  - Can see your code → DON'T RELY ON OBSCURE FILENAMES, ETC
  - Knows what format in which you'll store data

=> For full details, see the wiki ("Threat model" section)

API Overview

API: User functions

- `create_user(user, pass) -> User`
- `authenticate_user(user, pass) -> User`

Creates/Authenticates user in your system

- Generates or fetches any keys you'll need to implement other operations
- User object: you get to decide what goes in here
- All keys for encryption/integrity/etc will depend on this password (more on this later)
  - Don't worry about the user picking a bad password

API: File operations

BASED ON USER'S STATE (PER-USER KEYS ETC)

- `User.upload_file(filename, data)`
- `User.download_file(filename, data)`
- `User.append_file(filename, data)`

STRING BYTES()

- Upload/download a file securely
- Append to an existing file
  - Performance requirement: data sent must scale only with data being appended (ie, can't download and re-encrypt entire file)

- CS1620/CS2660 students: additional requirement on how files are stored for performance (more on this later) *(Must implement either this or "delegated sharing")*

ADDITIONAL PERF REQUIREMENT (SEE END FOR MORE INFO)

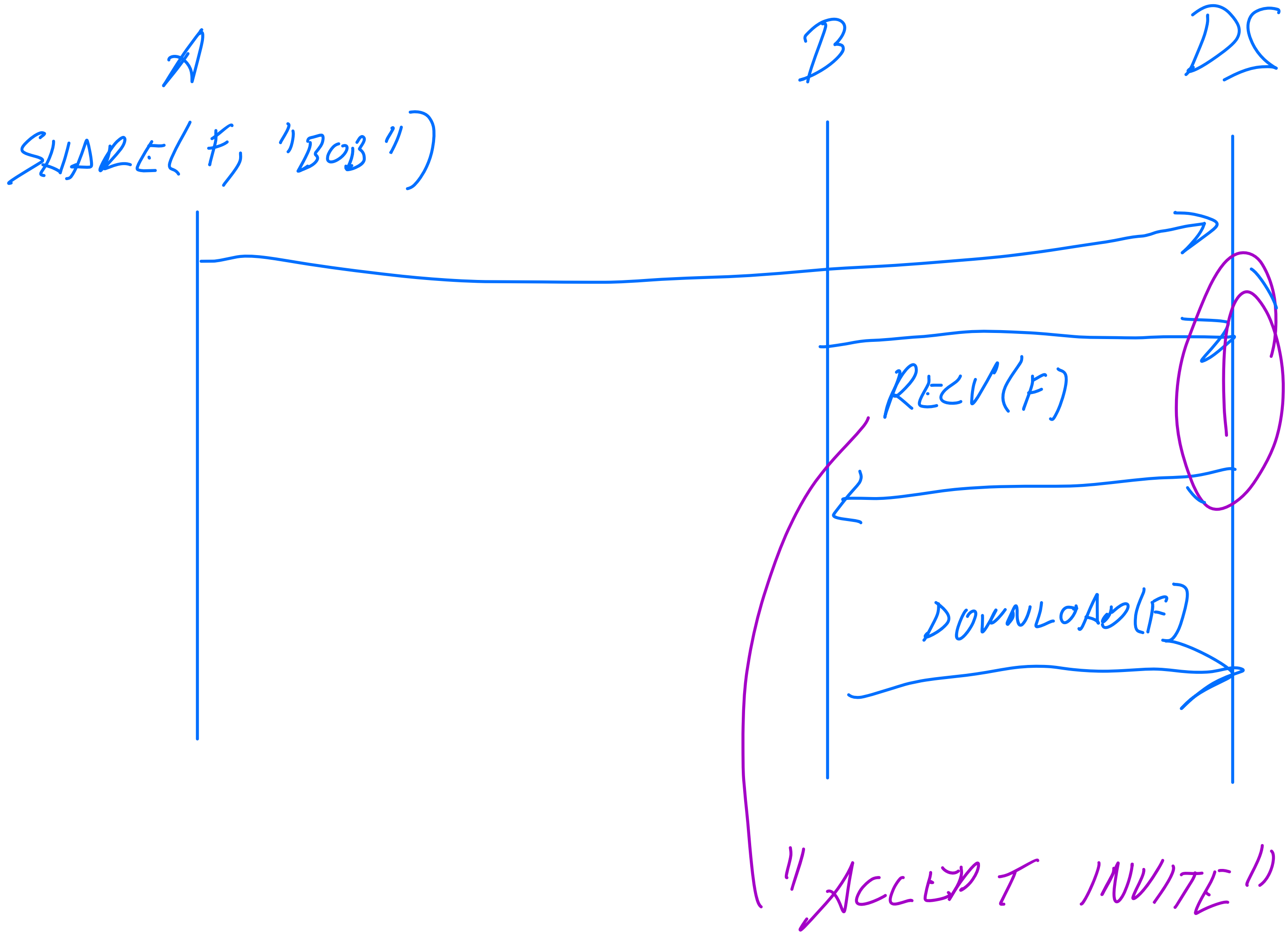
API: Sharing

- `User.share_file(filename, user_to_add)`
- `User.receive_file(filename, file_owner)`
- `User.revoke_file(filename, user)`

ONLY OWNER CAN SHARE/REVOKE

- Owner can share file with any number of users
- Users can do any file operations on file (upload, download, append)
  - All users see same copy of file
- Owner can revoke users
  - When user revoked, they can no longer do any operations on file

Extra CS1620/CS2660 requirement: "delegated sharing": users other than owner can also share (see wiki for details)
=> (CS1620/CS2660 students must implement EITHER this or "efficient updates" (previous slide)


A
B
DS
SHARE(F, "BOB")
RECV(F)
DOWNLOAD(F)
"ACCEPT INVITE"

What you WON'T implement

- Networking (it's all local)
- Writing actual files to disk
- Crypto (we provide a library)

⇒ You can think of the actual implementation as a secure, in-memory key value store

Note: All client state must be on the dataserver/keyserver

=>

```
# Make a user
client.create_user("usr", "pswd")

#           . . .

# Log in
u = client.authenticate_user("usr", "
# Make some data to upload
data_to_upload = b'testing data'

# Upload it
u.upload_file("file1", data_to_be_upl

# Download it again
downloaded_data = u.download_file("fi
assert downloaded_data == data_to_be_
```

Crypto primitives

The crypto library

The support code contains a cryptographic library, which provides the total set of cryptographic primitives you can use

- No external crypto libraries

- CONFIDENTIALITY
- INTEGRITY

<u>What you have</u>

- Asymmetric crypto (Encryption, digital signatures)
- Symmetric crypto (Encryption, HMACs)
- Hashing
- Key derivation functions
- Secure randomness

A big part of your design is deciding <u>how</u> to use these!

Asymmetric Crypto

Encryption

- Gen() -> K_pub, K_priv
- Encrypt(k_pub, data)
- Decrypt(k_priv, data)

CONFIDENTIALITY

Signing

- Gen() -> K_pub, K_priv
- Sign(k_priv, data)
- Verify(k_pub, data)

INTEGRITY

Asymmetric Crypto

Encryption
- Enc(k, m)
- Dec(k, c)

CONFIDENTIALITY

Authentication with symmetric crypto ← INTEGRITY
- Message authentication codes: computed based on hash of message, can verify if you have key
- HMAC(k, m) -> t (MAC)
- HMACEqual(t1, t2) => {0, 1}

THINK OF IT LIKE A KEYED HASH FUNCTION

Design: In general

- In general, use one key per purpose
  - Think about how sharing keys between operations can affect security
  - HashKDF is your friend
- A bit of software engineering can help you!
  - Consider making some helper functions for common operations
- I will post some examples on serialization (look for them!)

Asymmetric vs. Symmetric crypto

ASYMMETRIC

- CAN DISTRIBUTE KPUB
- SLOW
- LIMIT ON SIZE OF MESSAGES
- ANYONE CAN ENCRYPT JUST BY KNOWING KPUB

⇒ MAYBE USEFUL FOR SHARING

SYMMETRIC

- ONE KEY
- FAST VS. ASYMMETRIC
- CAN ENCRYPT ANY SIZE MESSAGE

⇒ GOOD FOR LARGE DATA.

⇒ YOU WILL HAVE MANY

Key derivation

- PBKDF2(password, salt, key_length) -> key_bytes
  - Secure generation of a key based on a password
  - Implemented as many iterations of a hash function (see passwords lecture)

← SYMMETRIC KEY

K_0 "SIGN"

- HashKDF(key, purpose) -> another_key
  - Given one key, generate another deterministically
  - Used to generate more keys!

HASHKDF(K_0, "SIGN")

SESSION 2 ⇒ K_{SIGN}

⇒ CAN USE TO COMPUTE SAME KEY FROM DIFFERENT SESSIONS.

SESSION 1

LOGIN("A", "PASS")

$\Rightarrow K_0$

HASHKDF(K_0, PURPOSE)

$= K_P$

SOME NAME FOR KEY'S PURPOSE (PUBLIC)

SESSION 2

LOGIN("A", "PASS")

$\Rightarrow K_0$

HASHKDF(K_0, PURPOSE)

$= K_P$

Q: WHY CAN'T ENCRYPT ALL FILES W/ SAME KEY?

ALICE: F_1 F_2 F_3

BOB: F_2

WHAT IF ALICE WANTS TO SHARE ONLY F_2 WITH BOB?

HashKDF example

```
base_key = crypto.SecureRandom(16)

derived_key_1 = crypto.HashKDF(base_key, "encryption")
derived_key_2 = crypto.HashKDF(base_key, "mac")

# Derived keys are the same length as the input key:
assert(len(base_key) == len(derived_key_1))
assert(len(base_key) == len(derived_key_2))

derived_key_3 = crypto.HashKDF(base_key, "encryption")

# Using the same base key and purpose results in the same derived key:
assert(derived_key_1 == derived_key_3)
```

Authenticated encryption

Your goal for most things is confidentiality AND integrity

Two operations:

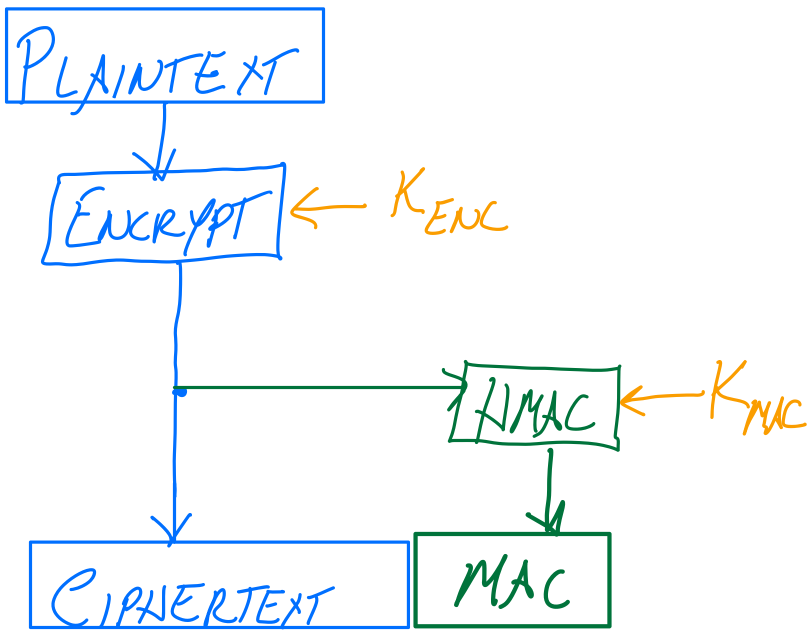
- Encrypt: Confidentiality
- MAC: Authentication

<u>Can combine these operations</u>

- EncryptAndMAC(k, m) => c, mac
- DecryptAndVerify(k, c) => m (or error if c doesn't pass integrity check)

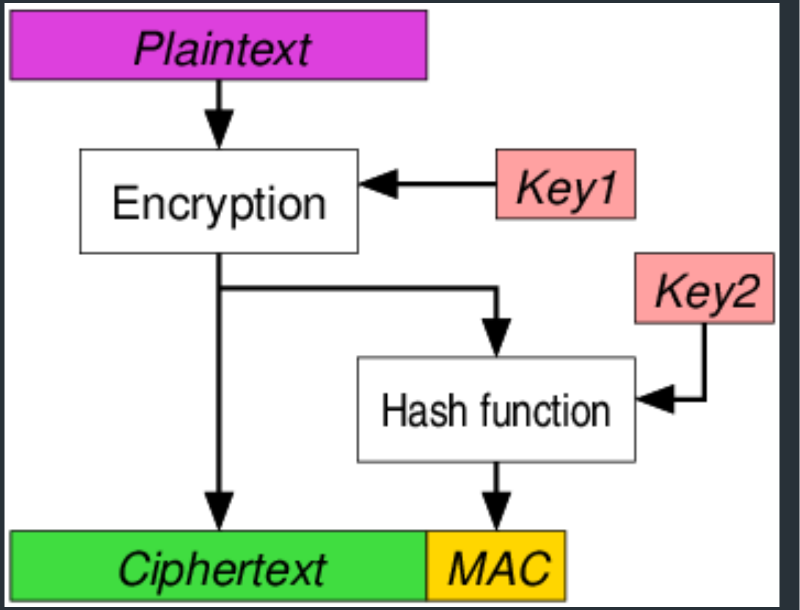
- How to do this is well-studied and has common pitfalls
  - Which do you do first? (Encrypt then MAC, MAC then encrypt, Encrypt THEN MAC, ...)
  - See cryptography lectures for more)
- You should use: Encrypt then MAC

ENCRYPT – THEN – MAC



Authenticated encryption

- You should use: Encrypt then MAC
- Proven to give us the security properties we want, when different keys used for encryption and hashing



Questions?

Setup and Stencil

Container setup & Environment

For this project, we'll use the "Development container" (same as project 1)

- Some slight updates—see setup guide for instructions

- Stencil uses a Python virtual environment
  - See setup guide for instructions
  - Like VSCode? You can use it with the container!

(UPLOAD_FILE)

CS1620/CS2660: Efficient updates

"Efficient" updates

- Broadly, When uploading a new file, bandwidth should scale based on amount of data that was changed
- How you do this is up to you, here's one way...

SHOULDN'T REQUIRE RE-UPLOADING THE WHOLE FILE.

UPLOAD(F_A, DATA$_1$)

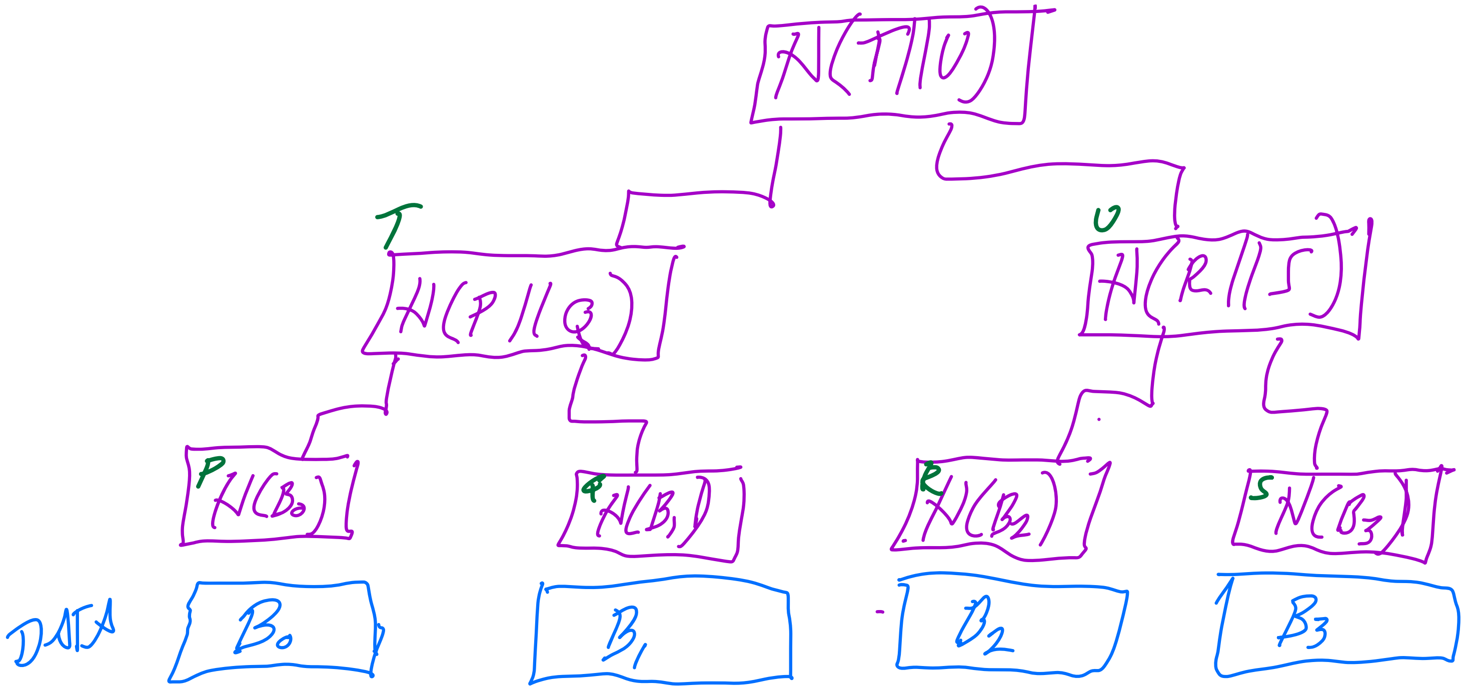
UPLOAD(F_A, DATA$_2$)

⇒ THINK ABOUT DIVIDING FILE INTO BLOCKS

DEAL W/ EACH BLOCK

HOW TO THINK ABOUT INTEGRITY WHEN FILE IS IN MULTIPLE BLOCKS?

ONE WAY: MERKLE TREE (HASH TREE)



For more notes on this, see the "Cloud Security" notes, starting on page 27
(Was extra reading from lecture)