Lecture 9: Deduplication (Part 2)

CSCI4180

Patrick P. C. Lee

Recap: Deduplication

- Given an input data stream (e.g., a large file or an input stream to a file system):
  - **Chunking**: divide the stream into chunks
    - Fixed-size or variable-size chunks (the latter can be done by Rabin fingerprinting)
  - Generate a **fingerprint** for each chunk by cryptographic hash (e.g., SHA-1)
  - For each chunk, check if it has been stored (by checking if the fingerprint has been recorded)
    - If yes, there is no need to store the chunk, but remember the reference (location) of the chunk
    - If no, store the chunk and record the fingerprint
- Space savings are measured by **deduplication ratio**
  - Raw logical data size / physical data size
  - Higher means better (e.g., 10:1 or 10x mean 90% savings)

Recap: Deduplication

Two key components:

- **Fingerprint index**: a key-value store that keeps track of the fingerprint and its reference
- **File recipe**: a manifest file that records the fingerprints and references of all chunks for each file
  - Each file has its own file recipe

Outline

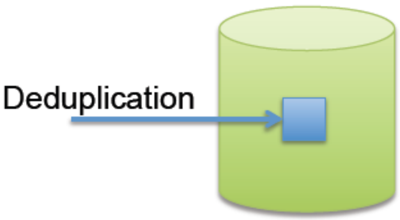
- Indexing techniques

- Read performance

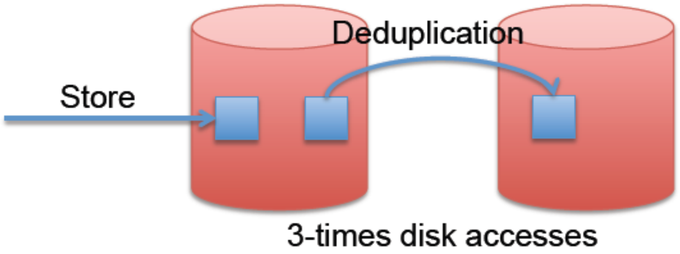
- Security on deduplication and Dropbox

Inline vs. Offline Dedup

- **Inline deduplication**
  - Immediately checks each data chunk for prior appearances
  - Duplicate chunks are never stored



- **Offline (out-of-order) deduplication**
  - First stores all data on secondary storage
  - Searches for duplicates in idle periods
  - No immediate impact on performance
  - Idle periods have to be big enough to perform deduplication



Deduplication

- We focus on inline deduplication
  - Performed during writes

- How to make deduplication very fast?

- Important application: **backup**!
  - Backup workloads can achieve up to 50:1 deduplication ratio [*]

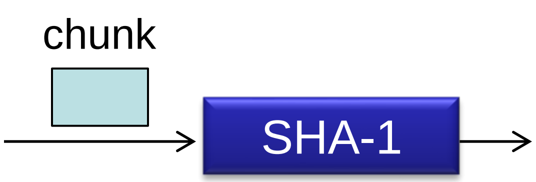
[*] Wallace et al., “Characteristics of Backup Workloads in Production Systems”, USENIX FAST’ 2012

Recap

- Deduplication on a stream of data:
  - Fingerprinting (compare by hash):
    - Generate identifiers of data based on content
  - Chunking:
    - Divide data stream into different chunks
    - Generate fingerprints for chunks
  - **Indexing**:
    - Maintain all fingerprints of existing chunks
    - *To be discussed in this lecture*.

Indexing

- Recall how we detect duplicate chunks?
- Compare by hash
  - Calculate one fingerprint every chunk (e.g., SHA-1)
  - Check whether this fingerprint is already known to the system
- An **index structure** is necessary
  - Keep all mappings of (fingerprint, chunk address, other info)
  - Data structures: hash tables, red-black-tree, etc.



Fingerprint (e.g., SHA-1)	Chunk address	Other info
88bdb5267379e362cb82c8963489761 3f9d098e5	0xFFFF0001	…
86349f088e2ba0b16400b35b789f50eb 1fde340a	0xFFFF0002	
41efd12cddf59989f1ad183f7c971bd4a 6de6a9c	0xFFFF000A	
…	…	…

Index structure

Indexing

- When writing a data chunk of a file:
  - If a chunk is new (cannot be deduplicated)
    - Write the chunk to the disk
    - Add (fingerprint, chunk address) to the index structure
  - If a chunk can be deduplicated
    - Read the chunk address and update the chunk address in the file's indexing structure
- Questions:
  - How many fingerprints to keep?
  - How/where to keep these fingerprints?
  - How to manage the fingerprints to optimize the ***write*** performance?

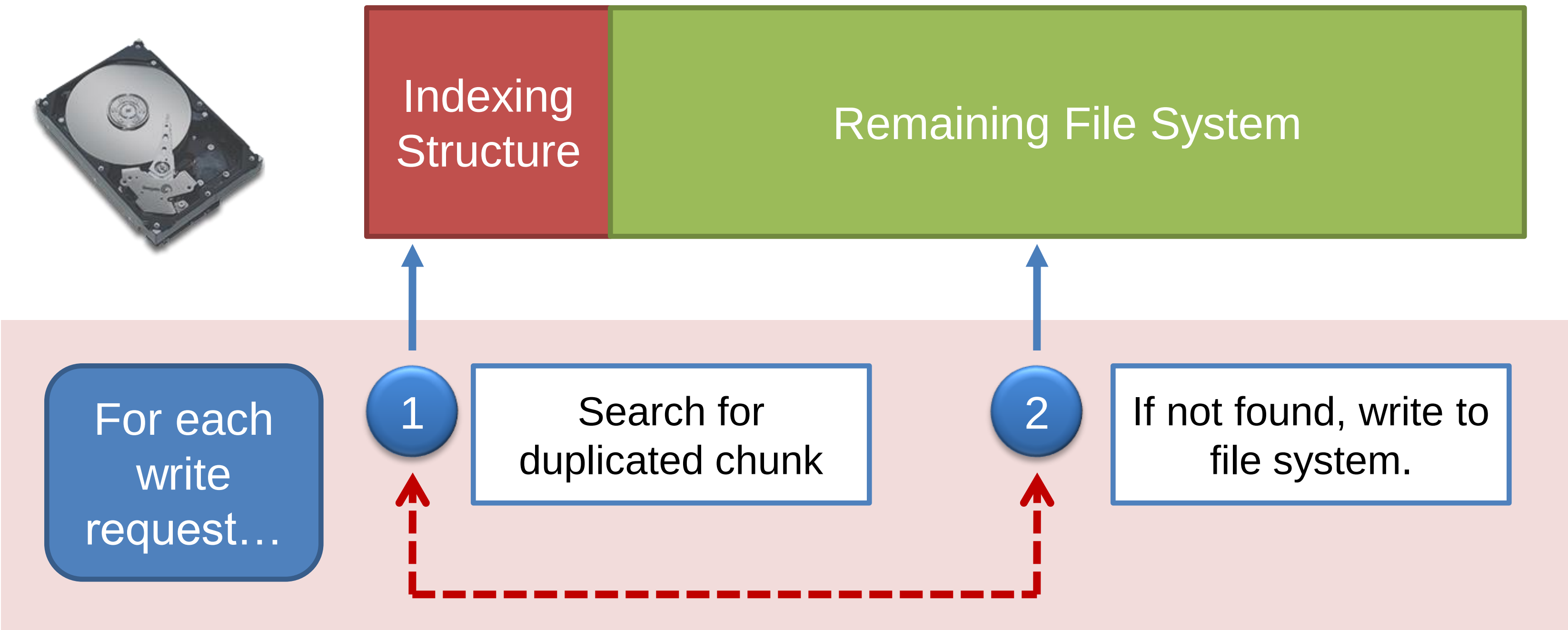
Option 1: Index Structure in RAM

- How about putting whole index structure in RAM?
  - Used in existing dedup file systems (e.g., Sparc ZFS)
- Challenge: need large amount of RAM
- Example: per 1TB of disk content

Chunk Size	4KB
Using MD5 fingerprint	16 bytes per chunk
Number of chunks	1TB / 4KB = 2^{28}
Size of Index	1TB / 4KB x 16 bytes = **4GB**.

Option 2: Index Structure on Disk

- How about putting whole index structure on disk?

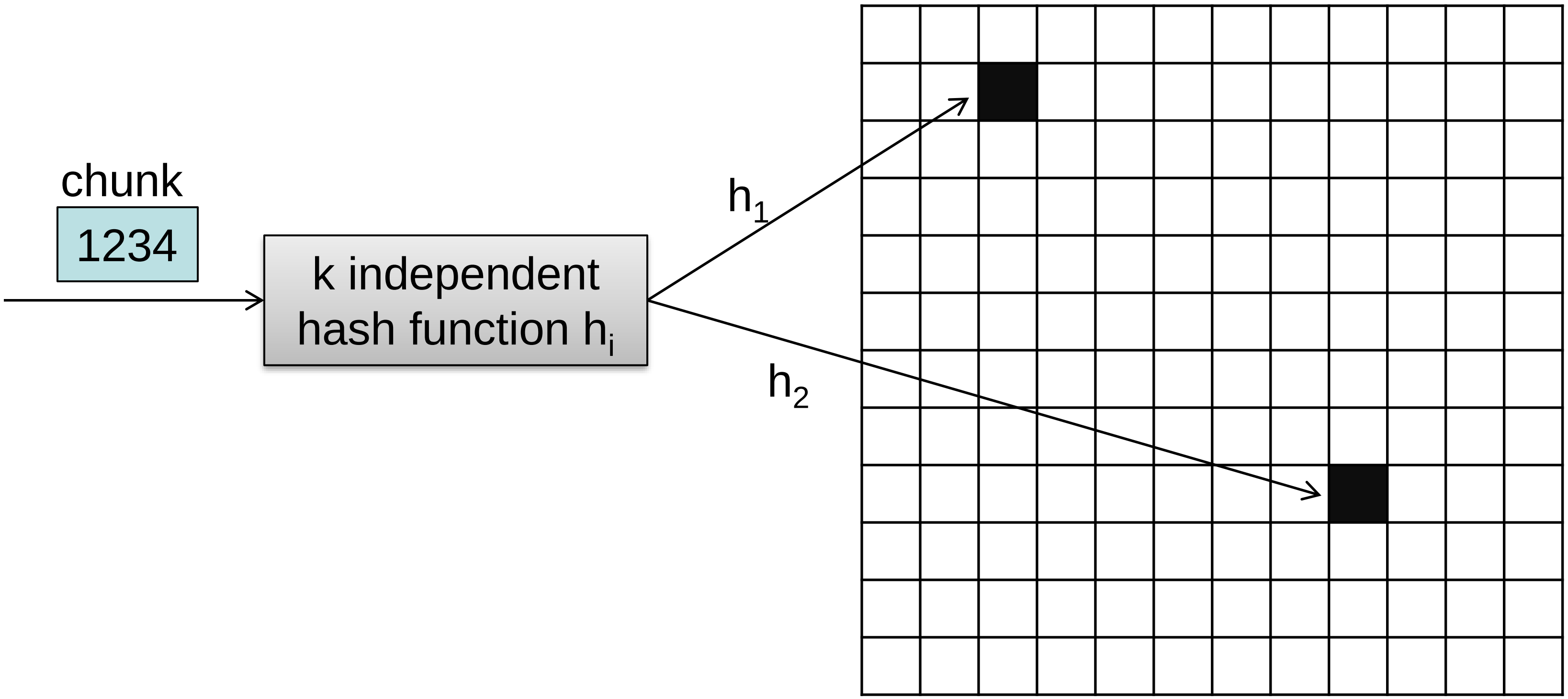


- Challenge: updating each data chunk and its index keeps the disk head moving, which hurts performance.

Bloom Filter

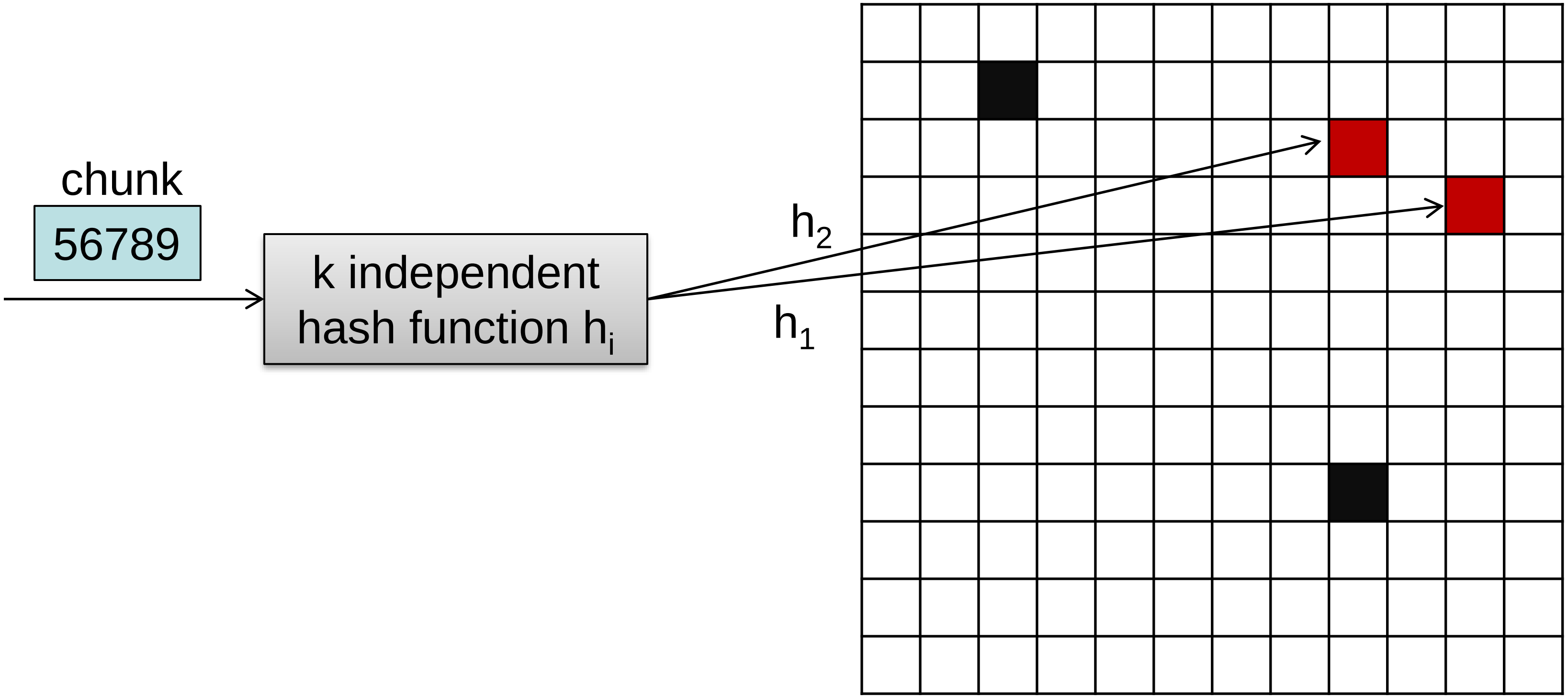
- Probabilistic data structure to decide if a key exists in a set
  - Insert(key), Lookup(key)
    - Lookup(key) = false → item guaranteed not in the set
    - Lookup(key) = true → item probably in the set
  - No Delete(key) operation
- Data structure
  - Bitmap of variable length ***m***
  - ***k*** independent hash functions
- Insert(i)
  - Set bit positions $h_1(i)$, …, $h_k(i)$ to 1

Bloom Filter Insert



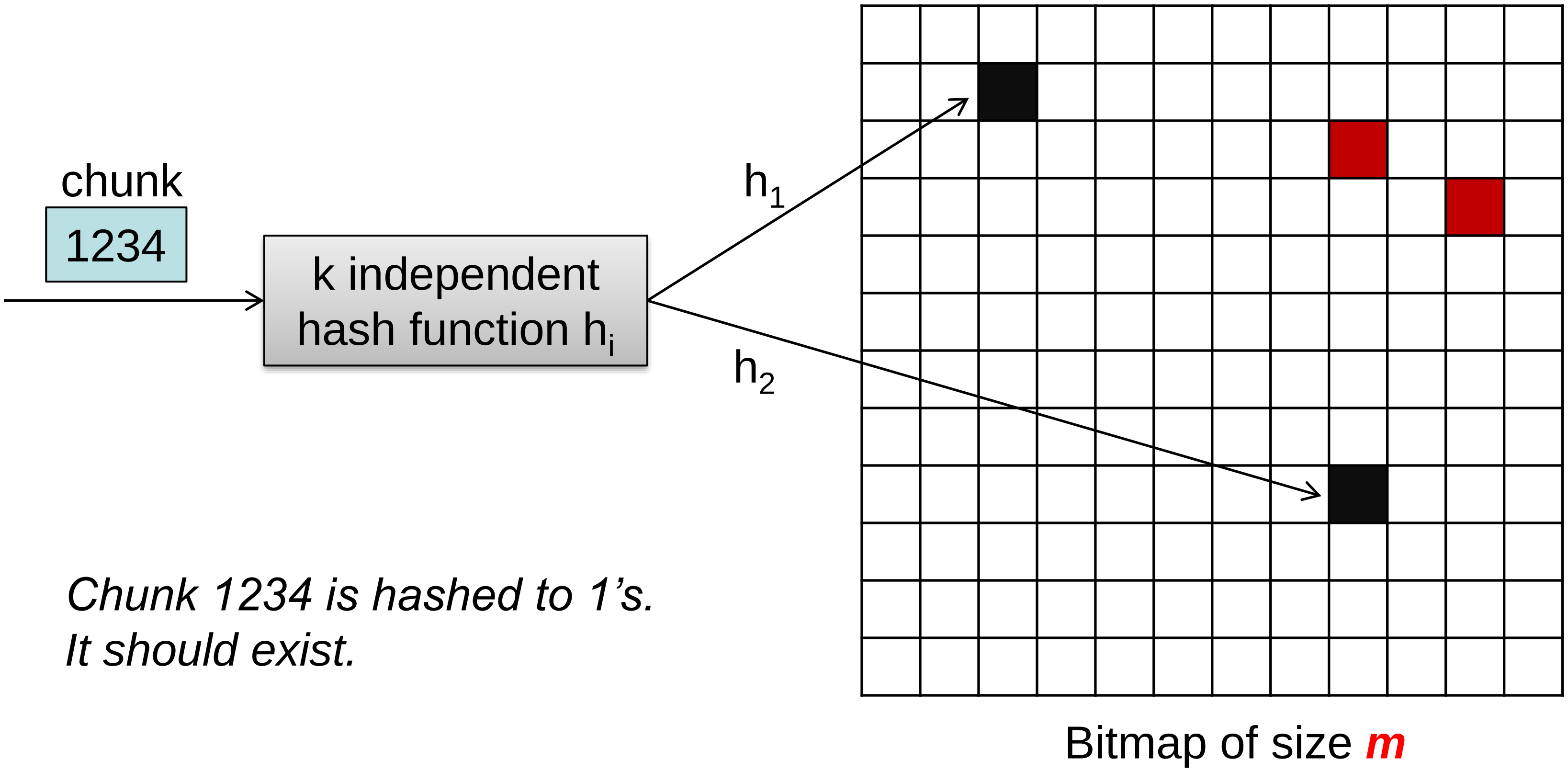
Bitmap of size ***m***

Bloom Filter Insert



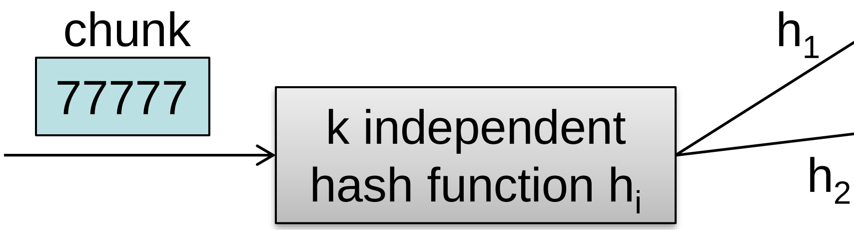
Bitmap of size ***m***

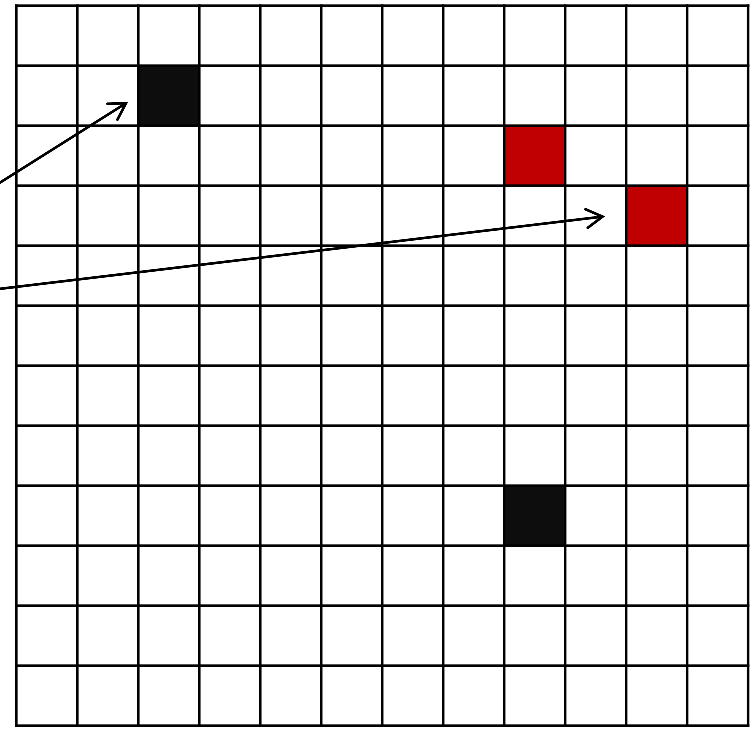
Bloom Filter Lookup



Chunk 1234 is hashed to 1's.
It should exist.

Bloom Filter Lookup





Bitmap of size ***m***

Chunk 77777 is hashed to 1's.
It should exist.
*But in fact it's a **false positive**!!*

Probability of False Positive

- Assume that hash functions are perfectly random and independent of each other
- Probability that a bit is not set after n inserts in a Bloom filter of size m bits using k hash functions:

$$\left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$$

- False positive probability

$$f = \left(1 - e^{-kn/m}\right)^{k}$$

Probability of False Positive

- For a given **m** and **n**, the value of **k** that minimizes the probability **f** is:

$$k = \frac{m}{n} \ln 2$$

- Given fixed **n** and **f**, the minimum **m** is:

$$m = -\frac{n \ln f}{(\ln 2)^2} \quad \text{(in bits)}$$

Probability of False Positive

- Example: 1TB storage, 4KB chunk size
  - Number of chunks: $n = 2^{28}$

False positive prob (f)	Memory size (in MB)
10^{-2}	306.7
10^{-3}	460.1
10^{-4}	613.4
10^{-5}	766.8
10^{-6}	920.2

- Much smaller than keeping all fingerprints in memory
- Independent of the fingerprint size

Bloom Filter in Deduplication

- Two-level index structure:
  - **Bloom filter in memory**. Check if a fingerprint has been (probably) inserted.
  - **Full chunk index mappings on disk**. Check if a fingerprint has been actually inserted
- Bloom filter for fingerprints
  - Lookup(fp) = false → No chunk index lookup needed
  - Lookup(fp) = true → Lookup on disk required

B. Zhu et al., “Avoiding the disk bottleneck in the data domain deduplication file system”, USENIX FAST 2008

Sparse Indexing

- Can we use less memory for indexing than Bloom filter?
- Observation:
  - **Chunk locality**: tendency for chunks in backup data streams to reoccur together
  - Example:
    - if the last time we encountered chunk A, it was surrounded by chunks B, C, and D, then the next time we encounter A (even in a different backup) it is likely that we will also encounter B, C, or D nearby
  - True for many backup applications

Lillibridge et al., "Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality", USENIX FAST 2009

Sparse Indexing

- Introduce a new level of granularity, called **segments**
  - Divide data stream into chunks
  - Each **segment** consists of a number of chunks
  - Segment size in order of 10MB
- Each segment is characterized by hooks, which are fingerprints of chosen chunks
  - Sampling rate defines number of hooks per segment
    - e.g., Sample hashes whose first n bits are zeroes (sampling rate = $1/2^n$)
  - All hooks are part of in-memory index (called **sparse index**)
  - Metadata of a segment, including fingerprints for all chunks, is stored in a **manifest**

Sparse Indexing

- Given an incoming segment, check the hooks in the sparse index, and identify already stored segments that are most similar (call them **champions**)
  - Champions are segments with highest number of hooks in common
  - Load manifests of selected champions into memory and deduplicate chunks
- Idea:
  - Long runs of chunks coming from the same segment
  - 1 (successful) champion lookup delivers fingerprints for many chunks

Extreme Binning

- Another way to reduce memory for indexing
- Observation
  - **File similarity**: two backup streams share files with similar chunks
  - Example:
    - If file A has a set of chunks in one backup window, then in the next backup window, file A still contains most of the chunks
  - *Note the difference with chunk locality (though the idea is similar)!!*
  - True for many backup applications

Bhagwat et al., “Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup”, IEEE MASCOTS 2009

Extreme Binning

- Extreme binning
  - Associate one representative fingerprint to characterize a complete file
  - This fingerprint is the minimal fingerprint
- Broder's Theorem:
  - Consider two sets S_1 and S_2, with $H(S_1)$ and $H(S_2)$ being the corresponding sets of hashes of the elements of S_1 and S_2, respectively, where H is chosen uniformly and at random from a min-wise independent family of permutations. Let min(S) denote the smallest element of the set of integers S. Then:

$$\Pr\left[\min(H(S_1)) = \min(H(S_2))\right] = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

  - The probability is the Jaccard similarity coefficient

Extreme Binning

- All metadata belonging to one file is stored within one **bin**
  - All fingerprints of chunks
  - SHA-1 value of the complete file (whole-file hashing)
    - Same file ➔ chunks of the file are all the same
- Bin is stored in secondary storage
- Primary in-memory index on each client is very small
  - Store all representative fingerprints and the pointers to their bins

Extreme Binning

Primary Index

Representative Chunk ID	SHA-1 Hash	Pointer to bin
045677a29c....	09591b28746.....	•
38a0acc909....	a20ae8a2eeb.....	
...	...	

Bin

Chunk ID	Chunk Size
a07b41fcbd11d...	1570
89cf1bf1c8bfc...	2651
...	...

Structure of the primary index and the bins

Extreme Binning

- To write files, each file is chunked and all fingerprints are calculated
- If representative fingerprint not in primary index
  - Store all chunks
  - Create new bin and store all metadata
- Else
  - If SHA-1 value for complete file isn't identical
    - Load corresponding bin from secondary storage
    - Store all chunks not in bin
    - Update metadata in bin and store bin
    - *Do not* update the whole file hash
      - since the written file may be different from the existing one being dedup'ed

Summary

- Different indexing techniques speed up write performance while using less memory
  - Bloom filter
  - Sparse indexing: exploits chunk locality
  - Extreme binning: exploits file similarity
- Above techniques mainly target for backups
  - Write-intensive, read-rare
- What about the read performance in deduplication storage systems?

Other Notes

- How about deletion from the index structure: **reference counting**.
  - If a fingerprint is first seen by the index structure, set the reference counter of that fingerprint to 1;
  - If a duplicate fingerprint is found, increment the corresponding reference counter by 1.
- For deletion, it means one reference to that fingerprint should be taken away:
  - Decrement the reference counter by one.
  - If the new value of the reference counter is **zero**, safely remove that fingerprint from the index structure.
    - The corresponding physical chunk can also be removed

Outline

- Indexing techniques
- Read performance
- Security on deduplication and Dropbox

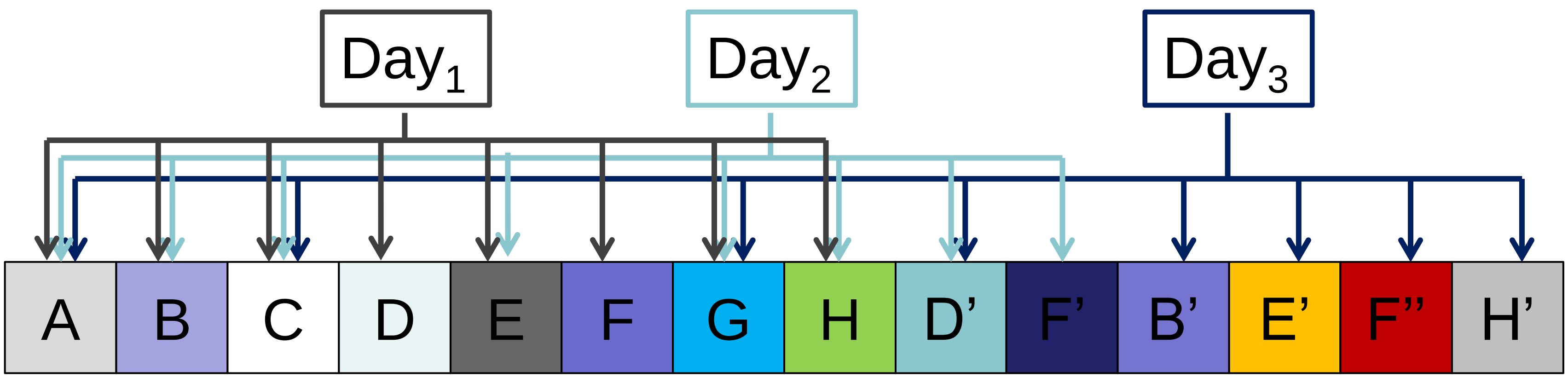
Fragmentation

- Inline deduplication: check if every new chunk can be deduplicated with any existing chunk on the write path:
  - If chunk exists: reference it
  - If new: add to last write position
- Data is no longer sequentially stored
- At restore time, many I/O seeks
  - Modern disks have poor random I/O performance
  - For backups, slow restore implies long system downtime

Fragmentation

- Latest backup → most fragmented

Day_1	A	B	C	D	E	F	G	H
Day_2	A	B	C	D’	E	F’	G	H
Day_3	A	B’	C	D’	E’	F’’	G	H’



Naïve Solutions

- Offline deduplication
  - Remove existing duplicate chunks at the background
  - Extra I/Os
    - Write redundant chunks first
    - Remove redundant chunks
- Chunk rearrangement
  - Similar to how disk fragmentation is resolved
  - But finding a good chunk layout is difficult because chunks are shared across disk
  - Expensive I/O

New Ideas

- Capping
  - Trade off lower deduplication for less fragmentation
  - That is, don’t remove all duplicate chunks, but write more to allow faster read
- Forward assembly area
  - Informed caching
  - Prefetching the data to be read
- Both solutions don’t require rearranging chunks

“Improving Restore Speed for Backup Systems that Use Inline Chunk-Based Deduplication”, FAST 2013

Containers

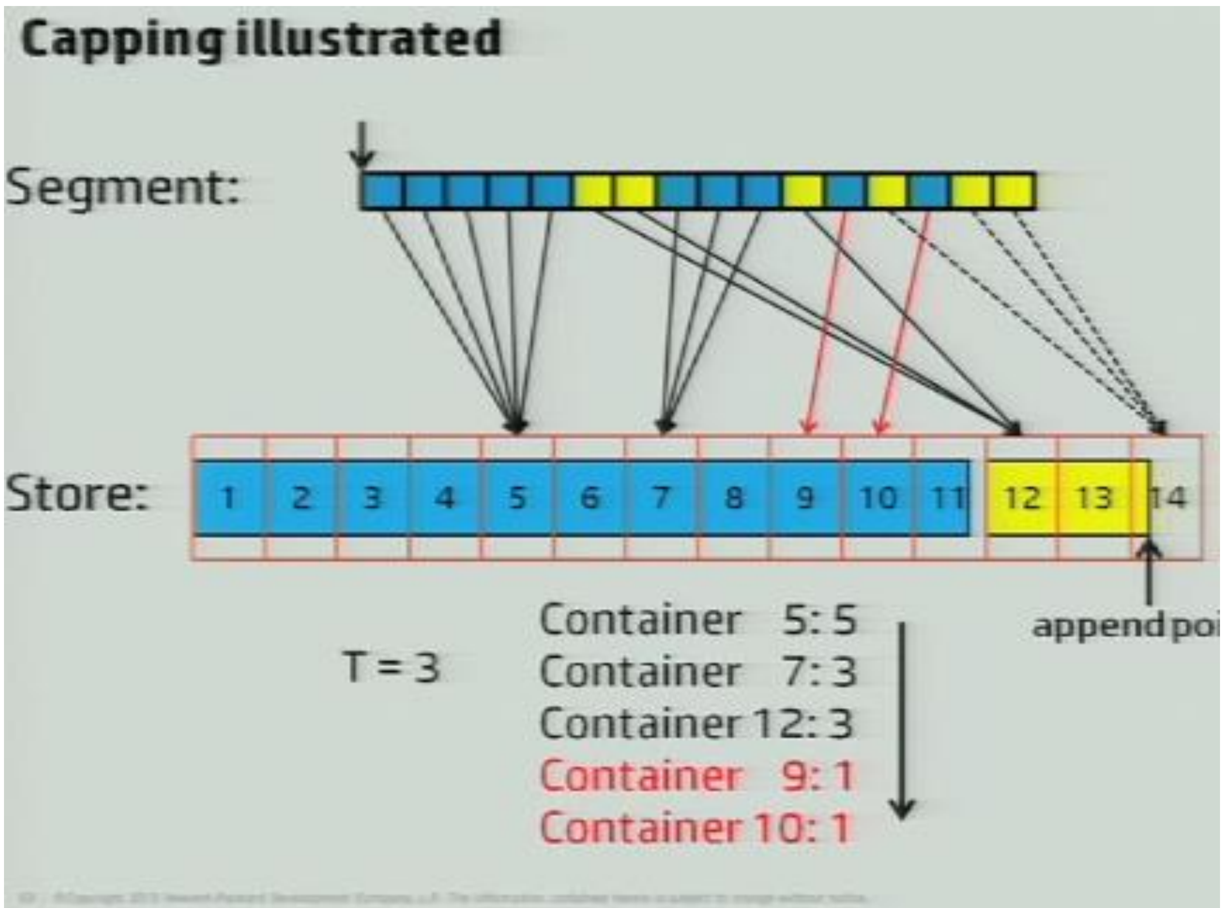
- Deduplication often works on small-size chunks (e.g., 4KB or 8KB)
- Chunks are grouped into **containers**, typically of size 4MB or 8MB
- Containers form the basic units of read/write operations
  - Reading a chunk → reading the whole container
  - Overhead of reading the whole container is small
    - Seek time ~ 10ms
    - Read time for 4MB container on 100MB/s SATA disk ~ 40ms

Capping

- Basics:
  - Container: a file that holds all chunks
  - Break a backup stream into segments
- Idea: limit the number of old containers that a segment can refer to
  - Break a data stream into fixed size segments (20MB)
  - Determine which chunks are already stored and in which containers
  - Choose up to T containers (capping level), ranked by how many chunks they contain, breaking ties to favor the most recent containers
  - Any duplicate chunks, if not referring to the T containers, are treated as new chunks

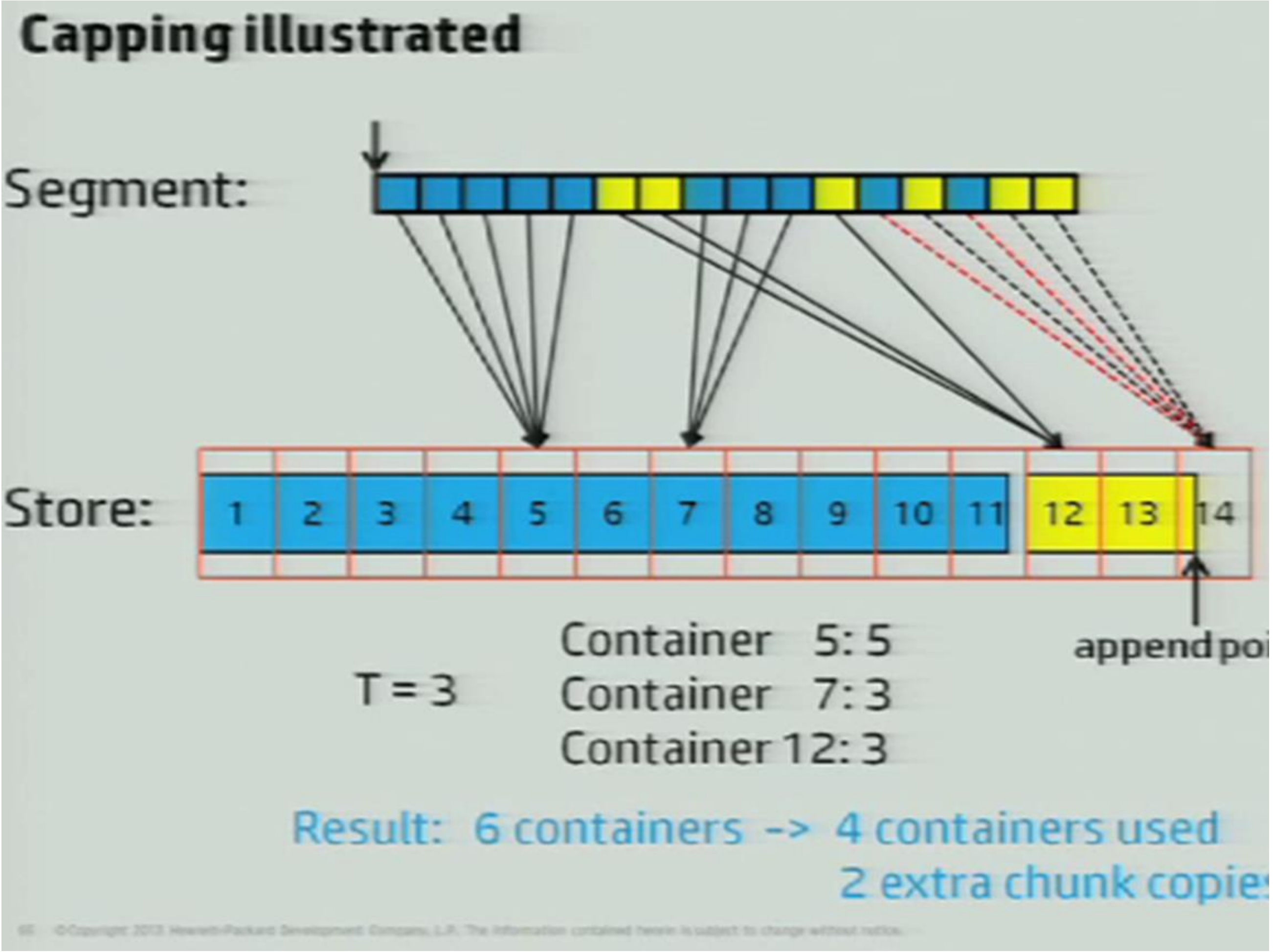
Capping: Example

- Only choose containers 5, 7, and 12



Capping: Example

- Result:



Forward Assembly Area

- Idea:
  - Future knowledge of accesses can be exploited to improve both caching and prefetching
  - Restore is deterministic
- How it works:
  - Prefetch next M bytes into an in-memory buffer
  - Assemble the M bytes in sequence and return it to the client

Summary of Results

- Capping: 2-6X, 8% dedup loss
- Forward assembly area: 2-4X
- Note the metric:
  - restore speed, speed factor = 1/containers read per MB (restored)
  - Open issue: what's the actual read throughput?

Outline

- Indexing techniques
- Read performance
- Security on deduplication and Dropbox

Convergent Encryption

- How to protect the confidentiality of our files on the cloud?
- Naive solution:
  - Encrypt files before uploading to the cloud
  - But encrypted files become scrambled and cannot be deduplicated
- Convergent encryption
  - Encrypt on a per-chunk basis
  - Use hash value as the cryptographic key

ftp://ftp.research.microsoft.com/pub/tr/tr-2002-30.pdf

Convergent Encryption

$E_k(m)$	Symmetric encryption with key k on message m
$D_k(m)$	Symmetric decryption with key k on message m
$H(m)$	Hash value of m
P	Plaintext
C	Ciphertext

- Given a plaintext chunk P
  - Encryption: $C = E_{H(P)}(P)$
  - Decryption: $D_{H(P)}(C)$

Convergent Encryption

- If two chunks (in plain) can be deduplicated, then after convergent encryption, they can still be deduplicated
- The hashes are securely protected
- Outsiders cannot decrypt the content without knowing the hashes

Convergent Encryption

- Is convergent encryption secure?

- Suppose you know that a plaintext chunk P must be an integer from 1 to 100,000
  - For example, P records the salary

- Given a ciphertext chunk C, can you tell what is its original plaintext chunk P?

Convergent Encryption

- Convergent encryption is vulnerable to **offline brute-force dictionary attack**
  - If the message space of a chunk is small (i.e., a chunk is predictable), brute-force dictionary attack is feasible
  - Root cause: the key is directly derived from the chunk
- How to fix?
  - Don’t use the hash as the key
  - Use a key that is deterministically derived from some one-way function for each chunk, but looks random to outsiders

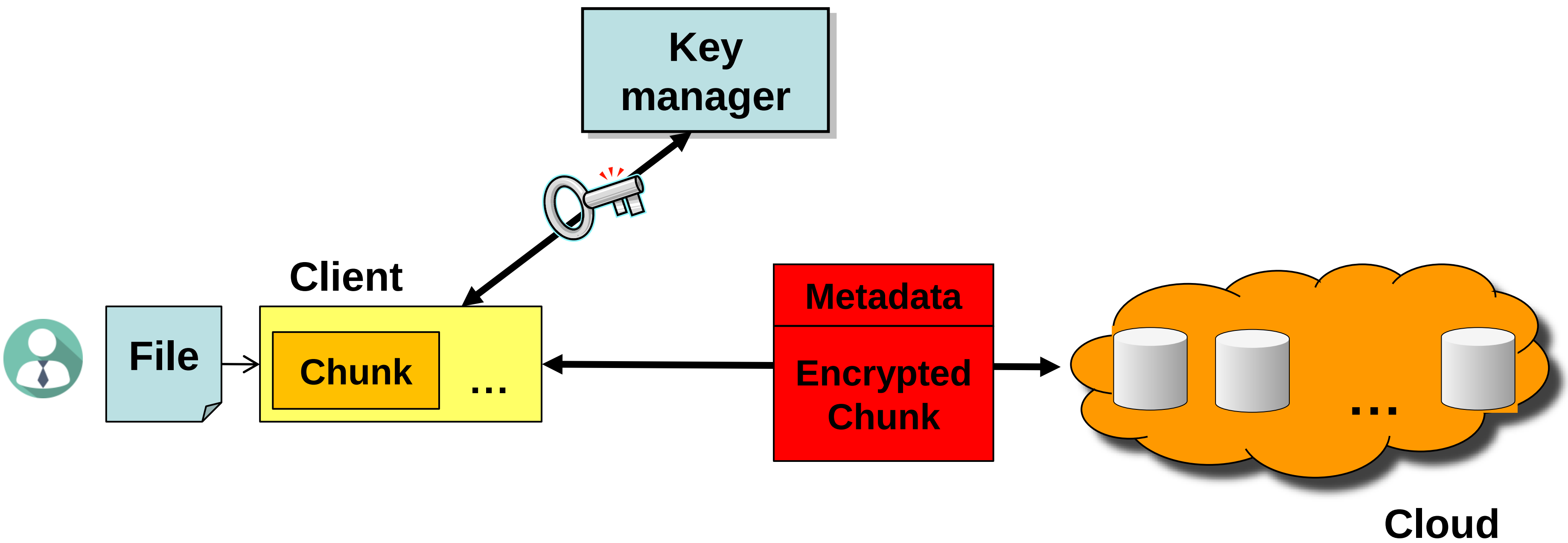
Server-Aided Encryption

- Key idea: use a dedicated key manager
- Key generation:
  - Client sends a fingerprint f to the key manager
  - Key manager generates a key K by:
    - K = OWF(f, s), where OWF(.) is a one-way function and s is a random seed that known to the key manager only
- Extensions:
  - **Rate-limiting** key generation requests
    - To avoid online brute-force attacks
  - **Blinded key generation**: key manager can generate the key without knowing f

DupLESS: Server-Aided Encryption for Deduplicated Storage [Bellare, USENIX Security'13]

Server-Aided Encryption

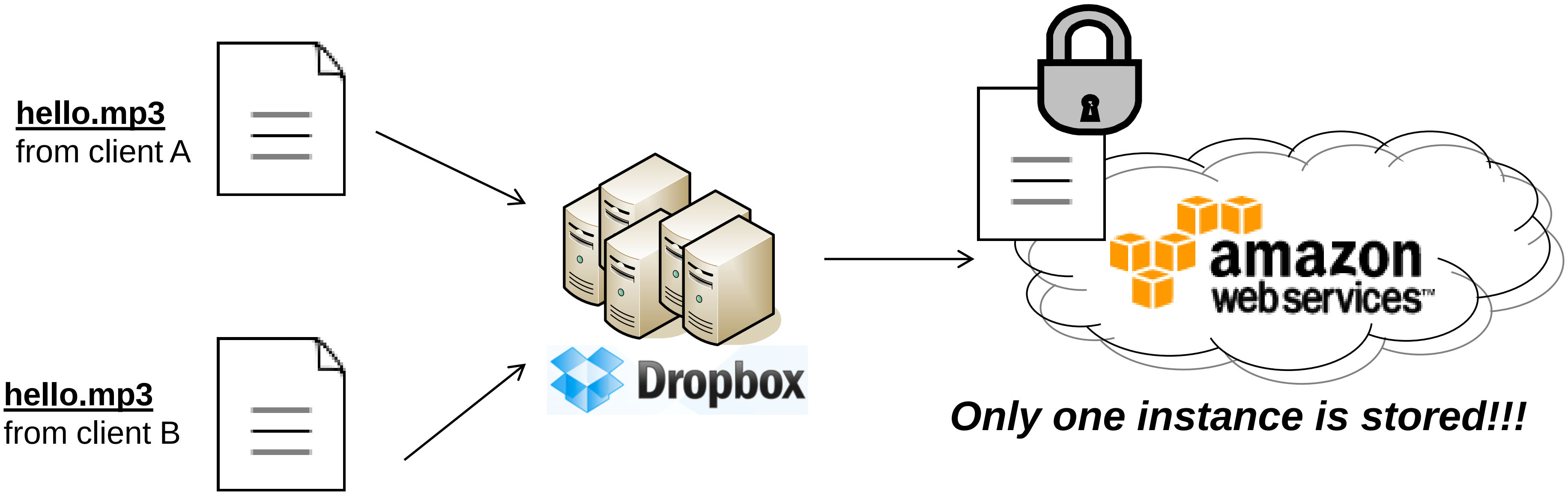
- Server-aided encryption:



DupLESS: Server-Aided Encryption for Deduplicated Storage [Bellare, USENIX Security'13]

Dropbox Security

- Dropbox encrypts clients' data before uploading to Amazon S3
- But is Dropbox entirely safe?



Recap: Dropbox

- Uses Amazon Simple Storage System (S3)
- data deduplication, using SHA-256
- files split in 4 MB chunks
- AES-256

Side-Channel Attacks

- Each client checks if the same file has been uploaded. If not, upload it; else, stop.
  - Save unnecessary upload bandwidth
- **Side-channel attacks**:
  - The attacker begins uploading a file and observes whether deduplication occurs. If there's deduplication, the attacker knows the file copy exists
  - Generalized to infer file contents: performing a brute-force attack by trying all possible contents of a file
- Root cause?

Harnik et al., "Side Channels in Cloud Services Deduplication in Cloud Storage", IEEE Security and Privacy, 2010

DropBox Attacks

- Researchers outline three cloud storage attacks and show their feasibility on Dropbox
  - Hash Value Manipulation Attack
  - Stolen Host ID Attack
  - Direct Up-/Download Attack

- All attacks work even the chunks are encrypted

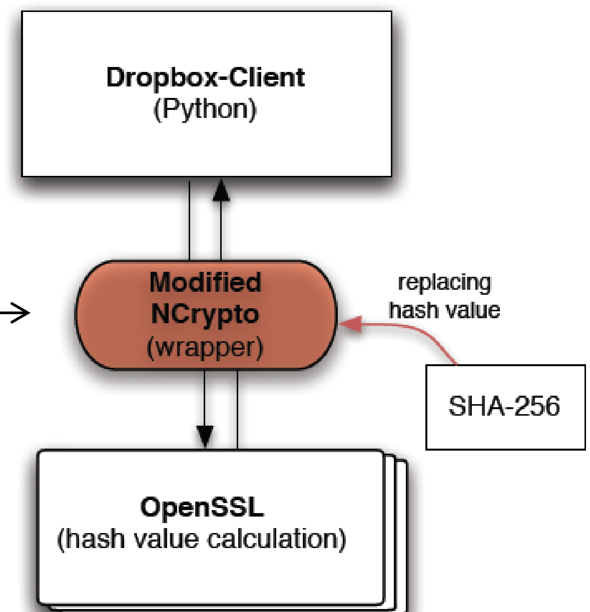
- All attacks are fixed by Dropbox

Mulazzani et al. “Dark Clouds on the Horizon: Using Cloud Storage as Attack Vector and Online Slack Space”, USENIX Security 2011

Hash Value Manipulation Attack

- Dropbox clients for Linux and Mac OS X dynamically link to NCrypto to generate SHA-256 hashes
  - Dropbox clients do not verify integrity of NCrypto

Hacker modifies the publicly available source code of NCrypto with their own fake hash value



Hash Value Manipulation Attack

- If the fake hash doesn't exist on Dropbox server:
  - Server requests the file from client
  - Nothing happens

- If the fake hash exists:
  - Server doesn't ask for the content
  - Instead, the server links the corresponding file/chunk to the hacker's Dropbox account
  - Hacker can access the file/chunk (which belongs to others)!

Stolen Host ID Attack

- In Dropbox, a unique **host ID** is used for client authentication
  - Host ID is a 128-bit key generated during the setup of the Dropbox client on a computer or smartphone
  - It will be linked the specific device to the owner's Dropbox account
  - No further authentication is required (no username/password required)

- If the host ID is stolen (by social engineering / malware)
  - All files of the user account can be accessed by the attacker

Direct Download Attack

- Transmission protocol is built upon HTTPS
  - Simple HTTPS request: **https://dl-clientXX.dropbox.com/retrieve**
  - As POST data: SHA-256 value & a valid host ID
- No check if chunk is linked with account!
  - Any valid host ID is fine!
- Dropbox hardly deletes any data
- Same effect as hash manipulation attack, but less stealthy
- Can be detected / prevented by Dropbox

Direct Upload Attack

- Same as retrieval, but for storing chunks
  - Uploading without linking
  - Simple HTTPS request:
    https://dl-clientXX.dropbox.com/store
- No storage quota / unlimited space
  - Create **Online slack space**!
- If host ID is known: push data to other people's Dropbox
- Can be detected / prevented by Dropbox

How Dark in Dropbox?

- Is Dropbox storing pirated files?
- Evaluation:
  - Download a first 4MB from some torrents of copyright stuffs (http://thepiratebay.org)
  - Generate SHA-256 checksums of those copyright files.
  - Using hash manipulation attack, one can confirm if the first 4MB exists!
- Results:
  - 97% retrievable from Dropbox
  - Interpretation: at least one seeder uses Dropbox

How Dark in Dropbox?

- Using direct upload attack to upload random data pieces… 100% of the data still exists after 1 month…

- Regular upload: unlimited undelete possible (> 6 months)

Solutions

- Aftermath - Dropbox reacted in April 2011:
  - They fixed the HTTPS Up-/Download Attack
  - Host ID is now encrypted on disk
  - No more client-side data deduplication (recently)

  - Server-side deduplication implies more upload bandwidth!!

Design Trade-Offs

- Design trade-off in deduplication:
  - Space savings
  - Indexing size
  - Write performance
  - Read performance
- How does the chunk size affect the design trade-off?

- Secure deduplication also presents a trade-off between space savings and security
  - See "*Balancing Storage Efficiency and Data Confidentiality with Tunable Encrypted Deduplication*" at EuroSys'20
  - http://www.cse.cuhk.edu.hk/~pclee/www/pubs/eurosys20.pdf