西北工业大学计算机学院生产实习-第四队学习资料

指导教师：崔禾磊 副教授

2019年7月1日

目录

- 项目背景介绍
- 重要文献总结

Background

- Digital data are explosively generated nowadays.
  - It is expected to reach **44 zettabytes** by 2020.*
  - The Internet is under tremendous pressure due to the exponential growth in bandwidth demand.

**IDC Report, Executive Summary: Data Growth, Business Opportunities, and IT Imperatives, 2014.*

- **Online storage service is demanded.**
  - Trend from simple backup services to cloud storage infrastructures.

Dropbox

amazon web services™ S3

iCloud

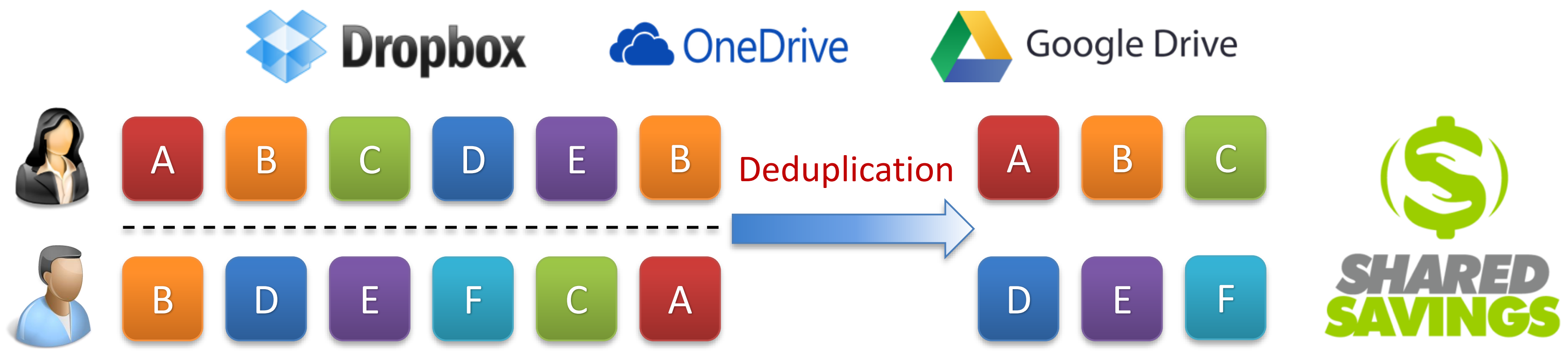
Windows Azure

OneDrive

Google Drive

Deduplication = Savings!

- **Data redundancy are everywhere!**
  - Can waste valuable **storage** and **bandwidth**.
- **Data deduplication** has been widely adopted by existing cloud storage services.

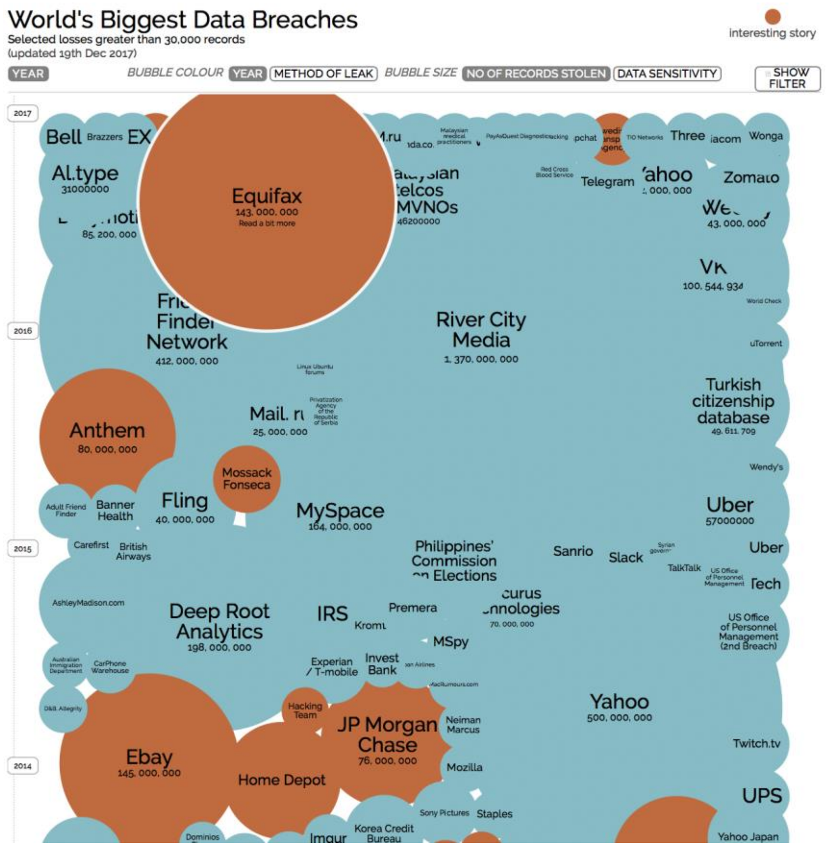


- Cross-user data deduplication can **save** storage costs by **over 50% in standard file systems**, and by **up to 90-95% for back-up applications***.

*Armknecht et al., "Transparent Data Deduplication in the Cloud", in Proc. of ACM CCS'15.

Caution!

- Exposing **content-sensitive data** to cloud raises **privacy concerns**.
  - Internal threats, e.g., software bugs;
  - External threats, e.g., compromised by an adversary.



Source: http://www.informationisbeautiful.net/ visualizations/worlds-biggest-data-breaches-hacks/

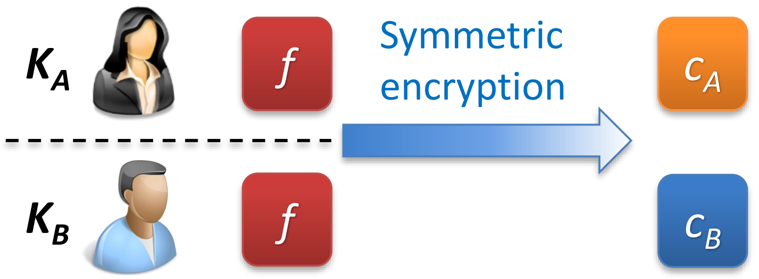
- It is desired to **achieve deduplication and encryption** simultaneously.

However, …

- Conventional encryption approaches:
  - **Randomized** ciphertexts.
- Deduplication methods:
  - Find **identical** files even being encrypted.

Diametrically opposed to each other.

Security of symmetric encryption.

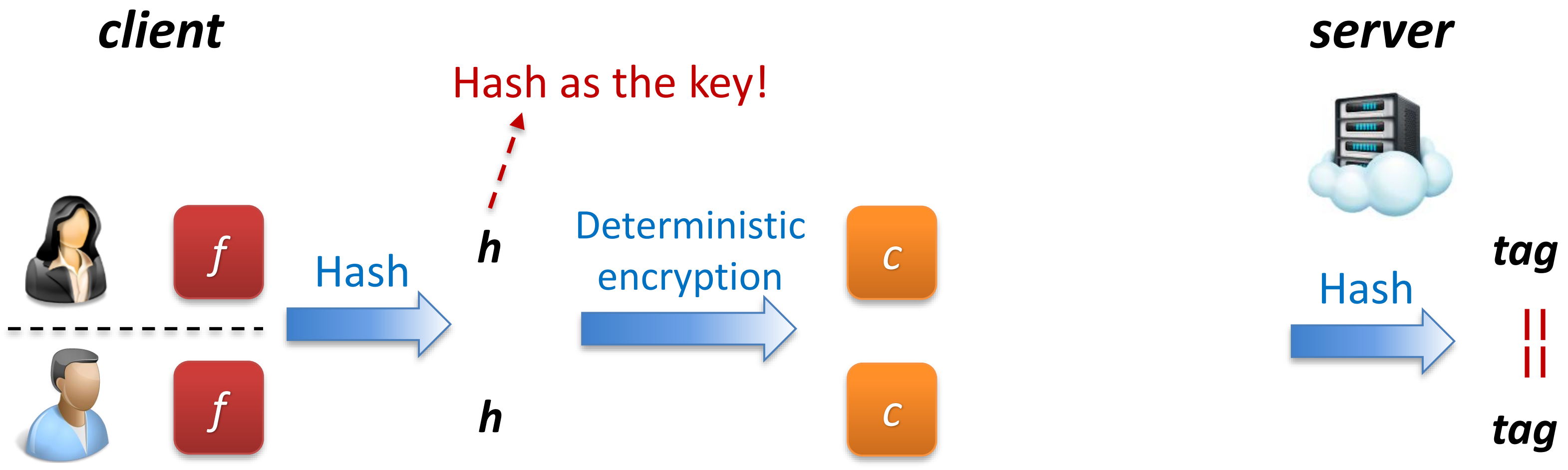


Pr[c_A = c_B] is negligible.
------>
Server has to store both c_A and c_B.

- Possible fix - 1: attach file hash **H(*f*)** to ciphertexts?
  - **Cross-user decryption is not possible.**
    - E.g., the user with K_B cannot decrypt c_A.
- Possible fix - 2: share a **network-wide key**?
  - **No compromise resilience**.
    - E.g., All data is insecure even if one client is compromised.

Convergent Encryption (CE)

- Later formalized as Message-Locked Encryption (MLE).
  - A cryptographic primitive for **deduplication over ciphertexts.**
  - **Core idea: the encryption key is derived from the message itself.**



same file --> same ciphertext --> same tag

* Douceur et al., “Reclaiming Space from Duplicate Files in a Serverless Distributed File System”, in Proc. of IEEE ICDCS, 2002.
* Bellare et al., “Message-Locked Encryption and Secure Deduplication”, in Proc. of EUROCRYPT, 2013.

Secure Data Deduplication

- **Goal**: provide data security, against both inside and outside adversaries, without compromising the space efficiency achievable through deduplication techniques.
- A **hot topic** in recent years: (to just list a few)
  - **[DABST, ICDCS’02]** -> convergent encryption.
  - [BKR, Eurocrypt’13] -> formalization of all the ad-hoc designs.
  - [BKR, USENIX Security’13] -> address offline attack via additional server.
  - [LAP, CCS’15] -> remove additional server.
  - [CMYG, TIFS’15] -> target big data and reduce metadata size.
  - [ZYWJWG, AsiaCCS’15] -> connect with multimedia applications.
  - [ZC, AsiaCCS’17] -> efficiently update large encrypted files.
  - …
- **However, secure and advanced applications are not fully studied.**

Strategy – Deduplication Granularity

- **File-level** deduplication:
  - The data redundancy is exploited on the file level.
  - Only one instance of the file is saved and subsequent copies are replaced with a "stub" that points to the original file.
- **Block-level** deduplication:
  - The data redundancy is exploited on the block level, where each file is divided into multiple blocks (or segment, chunks).
  - The block size can be either fixed or variable in practice.

Metric	File-level	Block-level
Searching Index size	Small	Large
Processing time	Quick	Slow
Deduplication ratio	Low	High

Strategy – Deduplication Architecture

- **Client-side** deduplication:
  - Deduplication acts on the data at the client before it is transferred.
    - The client communicates with the backup server (by sending hash signatures) to check for the existence of files or blocks.
    - If the server has the file, no need to upload.
  - **Saves bandwidth as well as storage.**
  - Known also as “Source-based deduplication” or “WAN deduplication”.
- **Server-side** deduplication:
  - The target storage server handles deduplication, and the client is unaware of any deduplication that might occur.
  - **Improve storage utilization, but does not save bandwidth.**
  - Known also as “Target-based deduplication” or “Destination-based deduplication”.

目录

- 项目背景介绍
- 重要文献总结

Reclaiming Space from Duplicate Files in a Serverless Distributed File System

John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer

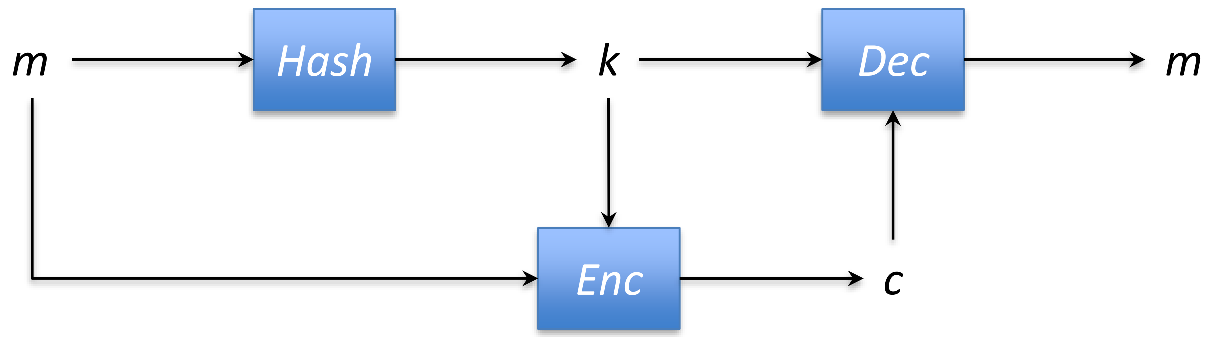
In *Proc. of **IEEE ICDCS**, 2002*

Target Problem

- To free space for storing data replicas, the system needs to group incidentally duplicated files. Meanwhile, the stored data should be encrypted for privacy protection.
- However, using a conventional cryptosystem to encrypt files is not suitable here.
  - Two identical files encrypted with different users' keys would have different encrypted representations, and the system can neither recognize that the files are identical or coalesce the encrypted files into the space of a single file.
  - Unless it had access to the users' private keys.

Convergent Encryption

- A cryptosystem that produces identical ciphertext files from identical plaintext files, irrespective of their encryption keys.
  - ***k* ← *Hash(m)***, where *Hash()* is a cryptographic hash function;
  - ***c* ← *Enc(k, m)***, where *Enc()* is a symmetric encryption algorithm;
  - ***m* ← *Dec(k, c)***, where *Dec()* is a symmetric decryption algorithm.



Conclusion

- Using convergent encryption (CE), the system, without knowledge of users' keys, can
  - determine that two files are identical;
  - store them in the space of a single file.

- However, it deliberately leaks a controlled amount of information, namely whether or not the plaintexts of two encrypted messages are identical.

Side channels in cloud services: Deduplication in cloud storage

Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg

IEEE Security & Privacy, 2010, Nov. 8(6):40-7.

Target Problem

- Although deduplication is most effective when applied across multiple users, cross-user deduplication has serious privacy implications.
  - Source-based deduplication: deduplication must be performed at the client side.
  - Cross-user deduplication: Each file or block is compared to the data of other users, and is deduped if an identical copy is already available at the server.

Security Issues

- They performed the following test to identify services that perform source-based cross-user deduplication:
  - Installed the service's client software on two different computers and created two different user accounts.
  - Used one account to upload a file.
  - Then used the second account to upload the same file, checking whether it was indeed uploaded.

- **If the file wasn't retransmitted over the network, concluded that the backup service performed source-based.**

Security Issues

- They identified the following services that perform cross-user, source-based deduplication (2010, then):
  - Dropbox
  - MozyHome
  - Memopal
- Three attacks on online storage services are described.
  - The first two let an attacker learn about the contents of other users' file, the third attack describes a new covert channel.

Attack I: Identifying Files

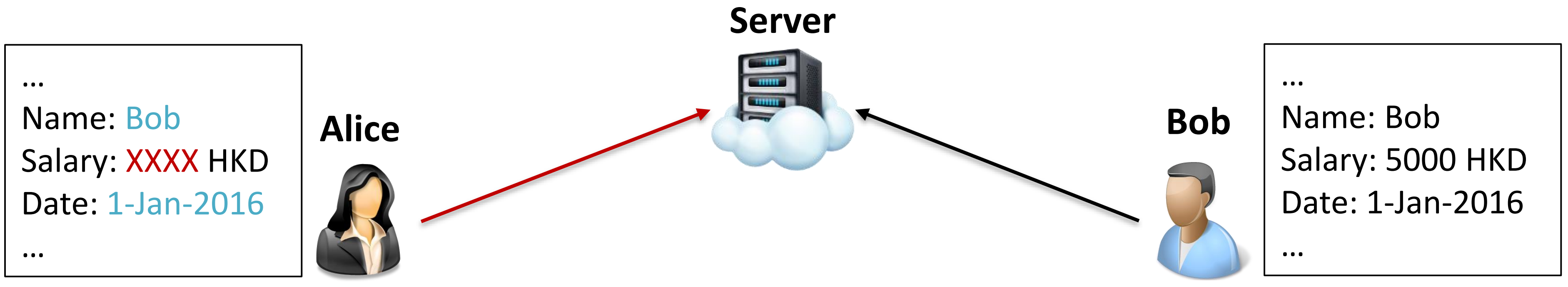
- An attacker Alice wants to learn information about Bob, a cloud storage service user.

- If Alice suspects that Bob has some specific sensitive file X that's unlikely to be in the possession of any other users, she can use deduplication to check whether this conjecture is true.

- All Alice needs to do is try to backup a copy of X and check whether deduplication occurs.

Attack II: Learning the Contents of Files

- Attack I only lets the attacker check whether a specific file is stored in the cloud storage service.
- The attacker might apply this attack to multiple versions of the same file, essentially performing a brute-force attack over all possible values of the file contents.
- For example:



Tries all possible values to detect which version occurs deduplication.

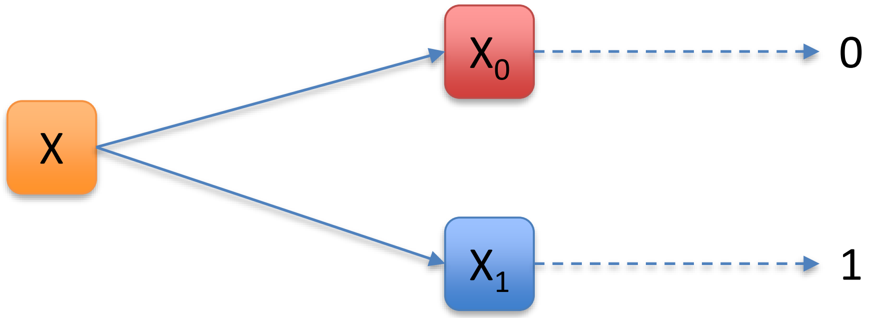
Attack III: A Covert Channel

- Suppose Alice installed some malicious software on Bob’s machine.
- The malicious software can send bits by using deduplication services (covert channel) from Bob’s machine to Alice, without detection.

How a single bit can be transferred?

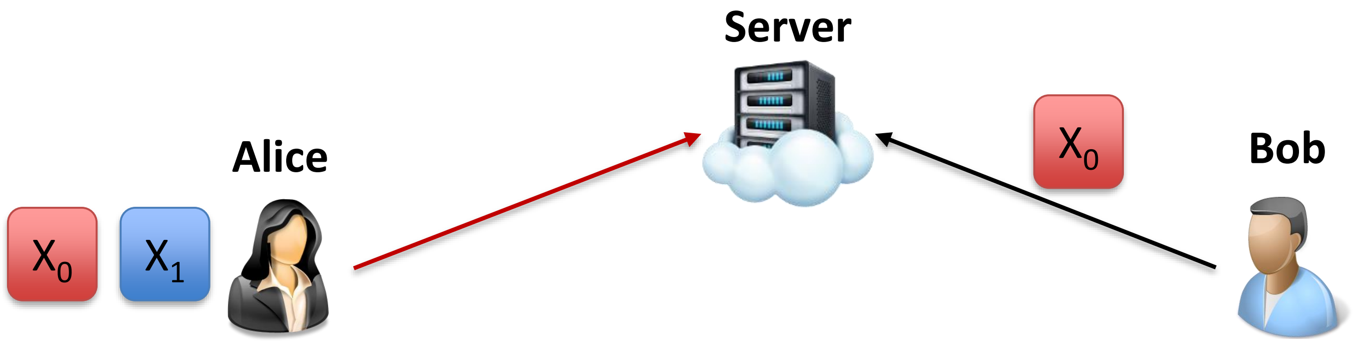
- The software generates one of two versions of a file, X_0 or X_1, and saves it on Bob's machine.
  - Alice can reproduce X_0 or X_1 on demand, as she creates the software.
- If it wants to transfer the message "0", it saves the file X_0; otherwise, it saves the file X_1.

- The files must be sufficiently random so that it's unlikely that any other user generates identical files.



How a single bit can be transferred?

- When Bob runs a backup and stores the file on the online storage service.
- Alice then performs a backup with the same service as Bob and learns which of the files, X_0 or X_1, was previously stored.
- **She learns what message the software sent.**



Notices that X_0 is uploaded.
Thus, obtain “0”.

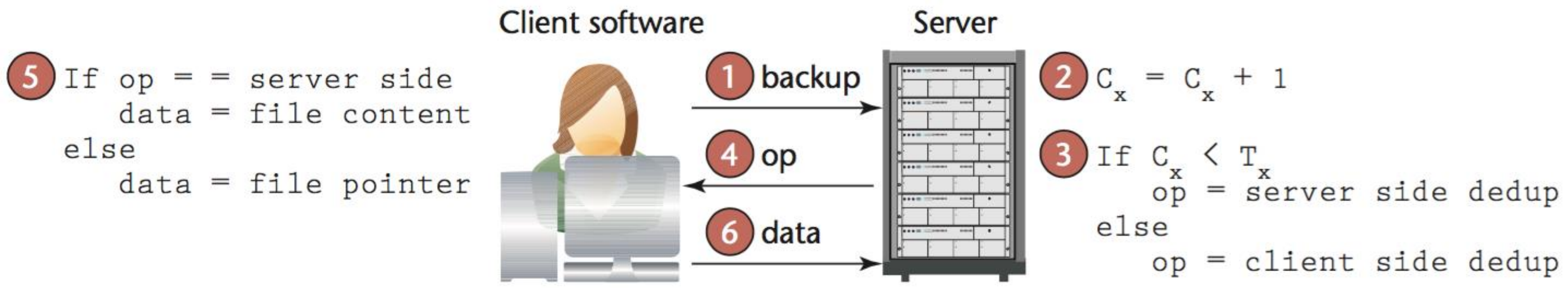
Solutions

- Using encryption to stop deduplication: ☹
  - Encrypt their data with their own personal keys.
  - Deduplication benefits will be dismissed.
- Performing deduplication at the servers: ☹
  - Files are always uploaded and the deduplication occurs at server side.
  - It will eliminate all of deduplication's bandwidth savings.
- **A randomized solution**: ☺
  - Assigning a random threshold for every file and performing deduplication only if the number of copies of the file exceeds this threshold.
  - **Weakening the correlation between deduplication and the existence of files in the storage service can reduce the risks.**

A Naïve Solution

- Using a global threshold?
  - The server sets a **global threshold** *t* (say, *t* = 10), and performs deduplication of a file only if at least *t* copies of the file have been uploaded.
  - In this case, Alice's uploading of a single copy of the file doesn't reveal whether Bob previously uploaded this file.
  - However, Alice can upload many copies of the file (even using multiple user identities) and check whether deduplication occurs after she uploads t or *t* − *1* copies of it.
  - The latter case indicates that a different user uploaded a copy.
- We can always assume that Alice can know the threshold *t* by conducting simple experiments to reveal the value of *t*.
  - E.g., using an unique file to test when deduplication occurs.

A Randomized Solution



- The figure details the operations performed when a client makes a backup request.
  - The server keeps an independent random threshold (T_X) for **every** file, which is chosen uniformly at random in a range [2, *d*], where *d* is a parameter that might be public.
  - If # of the document upload times (C_X) is above the threshold, the system performs client-side deduplication. No content is sent over the network.
  - Otherwise, deduplication is performed only at the server side.

Security Analysis

- To show that this solution doesn't reveal too much information about the inclusion of any file X in the data stored by the server, we compare the attacker's views in two instances.
  - **First, another user has already uploaded the file X;**
  - **Second, no copy of X has previously been uploaded.**
- If we can ensure that distinguishing between these two cases is difficult, then we can say that uploading a copy of the file doesn't substantially affect the attacker's view.

Security Analysis Cont.

- Consider three types of events in which the attacker wants to identify whether a (single) copy of a document was uploaded:
  - First, the attacker uploads a single copy of X and finds that deduplication occurs.
    - It therefore immediately learns that $T_X = 2$ and that a copy of X was previously uploaded by another user.
    - The probability of $T_X = 2$ is $1/(d-1)$.
  - Second, the attacker must upload *d* copies of X under different identities before deduplication occurs.
    - It therefore knows that $T_X = d$ and no copy of X was previously uploaded.
    - The probability of $T_X = d$ is also $1/(d-1)$.
  - Note that the two events cannot occur simultaneously.

Security Analysis Cont.

- Third, if deduplication occurs after the attacker uploads $2 \leq t < d$ copies of X, one of two cases could have occurred:
  - Either a copy of X was previously uploaded, and the threshold is $T_X = t + 1$;
  - Or no copy of X was previously uploaded, and the threshold is $T_X = t$.
- **We cannot distinguish the above two cases, because the T_X is chosen uniformly from [2, d].**
  - The probability that T_X was set to either t or t + 1 is exactly the same, regardless of whether X was uploaded.
  - So this does not leak any information about whether the file has been uploaded before.
    - That means the occurrence of deduplication does not add any information about whether X was uploaded to the server.

Conclusion

- Observe that under the condition that *X* was previously uploaded,
  - the first event (i.e., dedup after 1 copy upload), which leaks information, occurs with probability $1/(d-1)$,
  - whereas the third event, which doesn't leak any information, occurs with probability $1-1/(d-1)$.
- If *X* wasn't previously uploaded,
  - the second event (i.e., dedup after *d* copies upload) occurs with probability $1/(d-1)$,
  - whereas the third event occurs with probability $1-1/(d-1)$.
- Note that for any file only one of the two conditions holds.

Conclusion

- ***Theorem 1***. For a fraction of $1 - 1/(d - 1)$ of the files, the solution they have described leaks no information that lets an attacker distinguish between the case in which a single copy of a file was previously uploaded and the case in which the file wasn't previously uploaded. (d is the number of users who own a same file.)

- More detail in paper.

Proofs of Ownership in Remote Storage Systems

Shai Halevi, Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg

In *Proc. of **ACM CCS**, 2011*

[Slides credits in part to HHPS, CCS’11]

Client-Side Cross-User Deduplication

- Prior to uploading the file:
  - The client computes a hash value (*tag*) over the file, and sends it to the server.
  - The server checks if the file already exists in its storage (via the *tag*).
    - If not, it uploads the file from the client.
    - If yes, it does not need to upload it.
- Benefits:
  - Saves storage space (at the server).
  - Saves bandwidth (at both sides).

However, …

- Server state is a “joint resource” across different users.
- Answer to “does-file-exist-on-server” leaks one bit of information about other users.
  - [HPS, IEEE Security & Privacy’10] uses this channel to leak “interesting” information.
- Opens the door to stealing files.
  - Hash of file serves as identifier for content

    -> **A File-Stealing Attack**

A File-Stealing Attack (Back then 2011)

- Attacker obtains hash for victim’s file.
  - More on how to do it later.
- Connects to server, tries to upload the file.
  - Server asks for hash, attacker complies.
  - Server skips upload, remembers that attacker owns the file.
- Attacker asks to restore the file, downloads it from the server.

If you can get the hash of the file, you can get the file.

Getting the Hash Value

- Hash is not meant to be secret.
  - The deduplication procedure may use a common hash function (e.g., SHA1, MD5).
- May be used for other purposes:
  - "Should not reveal anything about the file."
  - Fingerprint software/media, timestamp contributions, …
    - E.g., I publish a fingerprint of my software, one user backs it up, now everyone can get it from server.

Threats of Getting the Hash Value

- Malicious software.
  - A malicious software on Bob's machine wants to stealthily leak all his files to Alice (attacker).
  - Instead of sending huge files, can send the short hash values of the files.
    - Much harder to detect and prevent.
- Also true for server break-in.
  - Dump all hashes in memory and run…
  - Even if detected, only remedy is to turn off deduplication for affected files (essentially forever).

Threats of Getting the Hash Value

- Content distribution network (CDN).
  - Alice wants to share a huge file with her friends;
  - Uploads file to server, sends hash to friends;
  - Friends use backup service to download file.
- Server used as a CDN, unknowingly.
  - Might break its cost structure.
    - If it planned on serving only a few restore operations.
  - Might break the law.
    - If huge file was copyrighted.

Solution:
Proofs-of-Ownership (PoW)

A Naïve Solution

- Use application-specific hash, salt
  - E.g., SHA("service name" || salt || file).
  - Other applications won't use the same hash.
  - Solves fingerprinting/timestamping scenarios.
- But hash is still not secret.
  - All clients must know hash function.
- Does not address root cause of problem.
  - **Large file is still represented by a short string, if you can get the short string then you get the file.**
- Many attack scenarios remain (CDN, break-in, etc.).

A Better Naïve Solution

- Use a challenge-response mechanism.
  - E.g., for every upload, server picks a random nonce, asks client to compute SHA(nonce || file).
  - This “proves” that client knows the file. ☺
  - But server must retrieve the whole file from secondary storage to check the answer. ☹
- We want a better proof mechanism.

Proofs of Ownership (PoW)

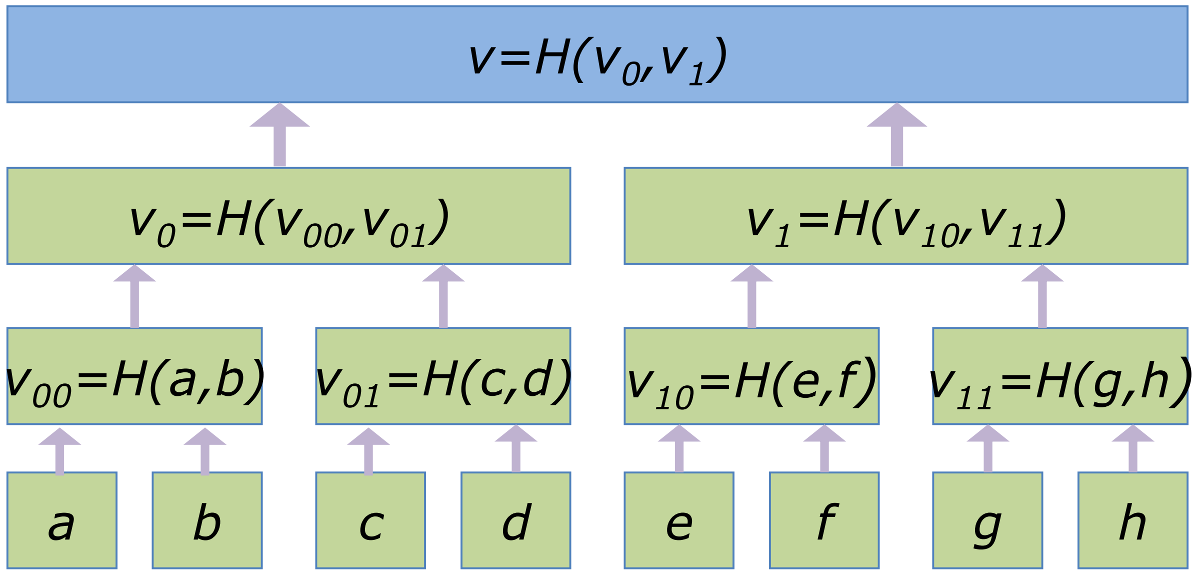
- Protocol for client (prover) and server (verifier).
  - Client has the file;
  - Server stores only short verification information;
    - Verification information computed from the file.
  - The proof itself is bandwidth-efficient.
    - Much shorter than sending the whole file.
- Adversary may have partial information about the file.
  - E.g., its hash value, maybe more.
- Want proof to succeed only if client has the whole file.

Practical Considerations

- Low bandwidth.
- Very short verification information.
  - Only a few bytes per file.
- Efficient processing by client and server.
  - File itself may be very large, perhaps does not even fit in main memory.
  - Would be nice to have a streaming solution, (e.g., similar to just computing SHA(file)).

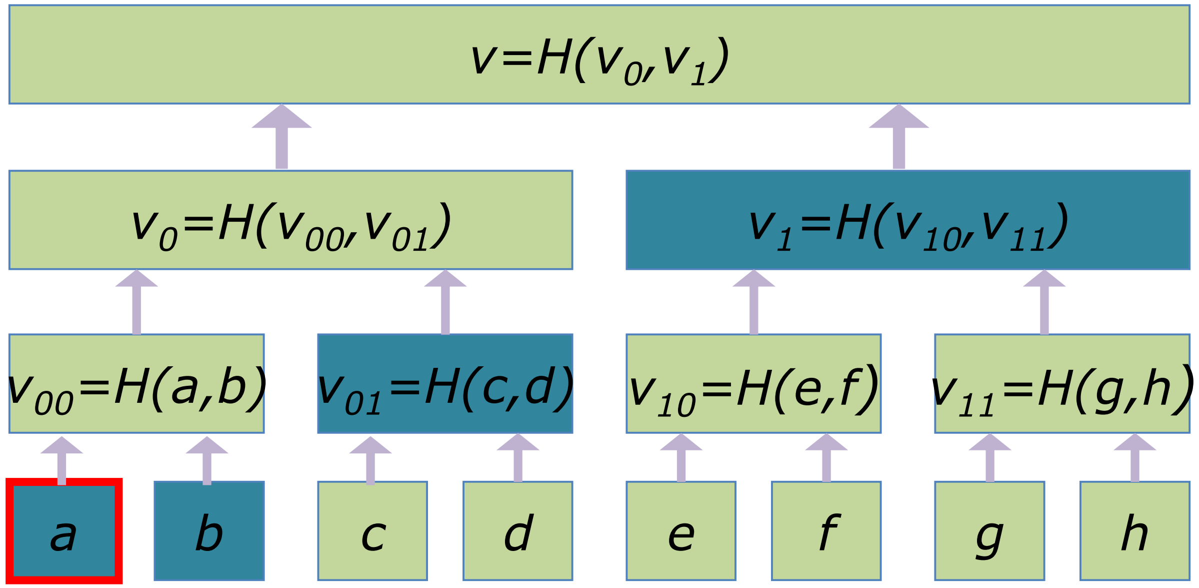
Background: Merkle Hash Trees

- Committing to *n* values, $x_1,\ldots,x_n$, such that
  - The commitment is short (a single hash value);
  - Can "open any x_i" with a de-commitment message of length $O(\log n)$.



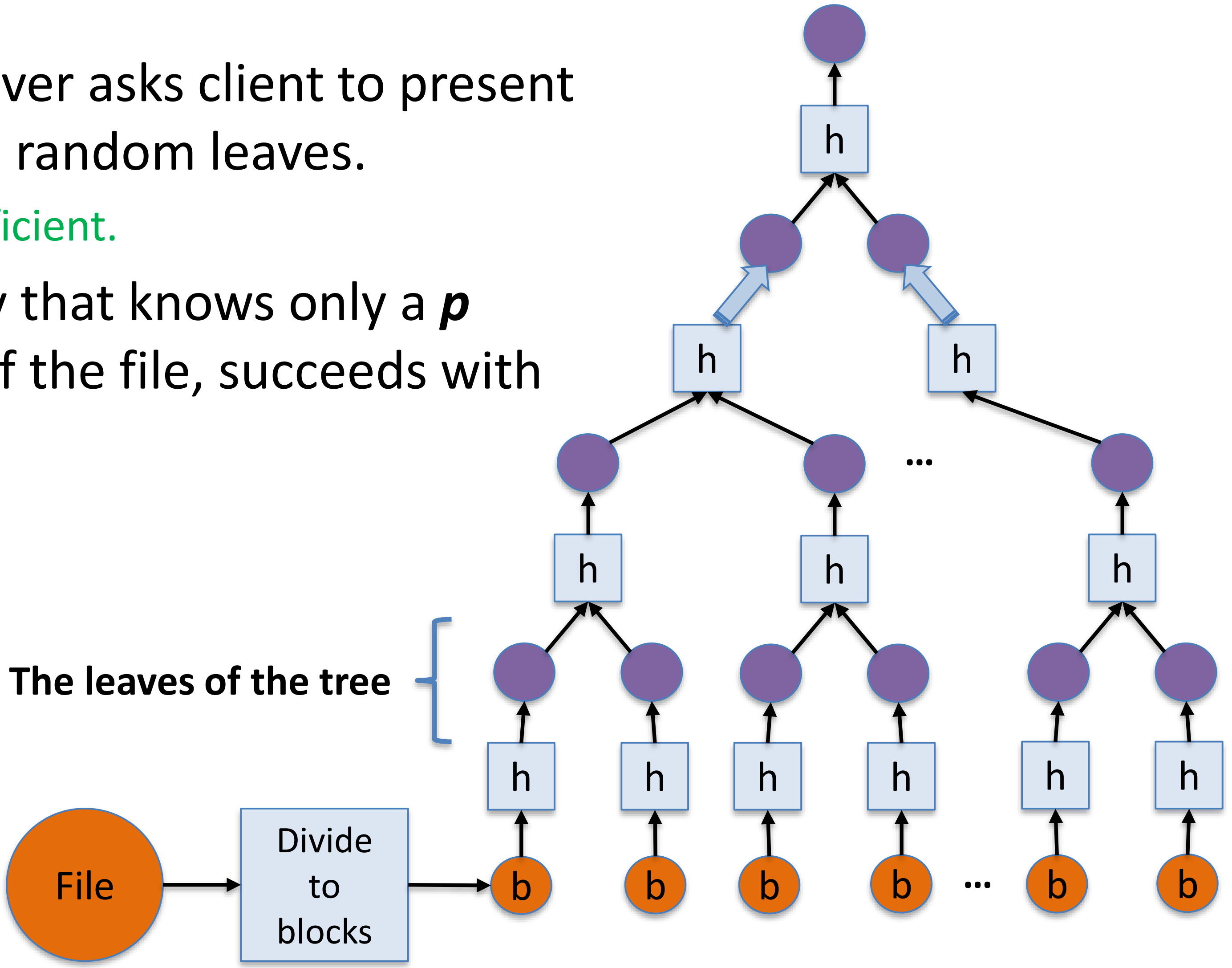
Background: Merkle Hash Trees

- The commitment is the root value *v*.
- To open a leaf, send the **sibling path** from that leaf to the root.
  - **Sibling-path** of a leaf:
    - The leaf together with the siblings of all the nodes in the path from the leaf to the root.
  - E.g., opening leaf *a* by providing *b*, v_{01}, and v_1.



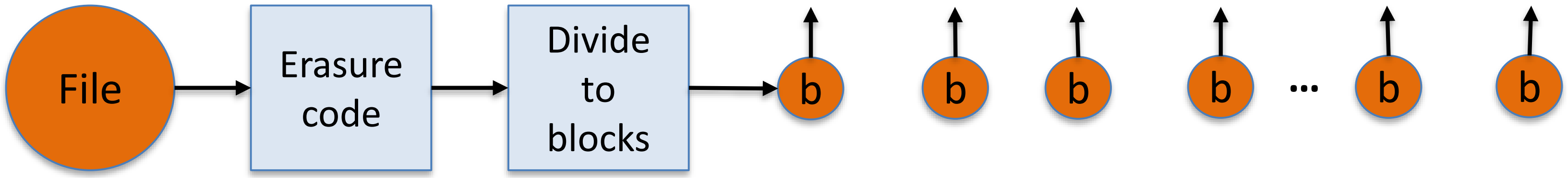
Solution: First Attempt

- Proof: server asks client to present paths to *L* random leaves.
  - Very efficient.
- Adversary that knows only a ***p*** fraction of the file, succeeds with **prob < p^L**.



Problem and Solution

- Adversary that knows a large fraction of the blocks (say, 95%), can pass the test with reasonable probability ($0.95^{10}=0.6$).
- Solution: apply Merkle tree to encoded file.



- Erasure code property: knowledge of, say, 50% of the encoding suffices to recover original file.
  - Attacker who misses even a single block of the file, does not know > 50% of the encoding.
  - Fails in each Merkle tree query with probability 50%.
  - Cheating probability is 2^{-L}. (A simple “hardness amplification” result.)

Efficiency?

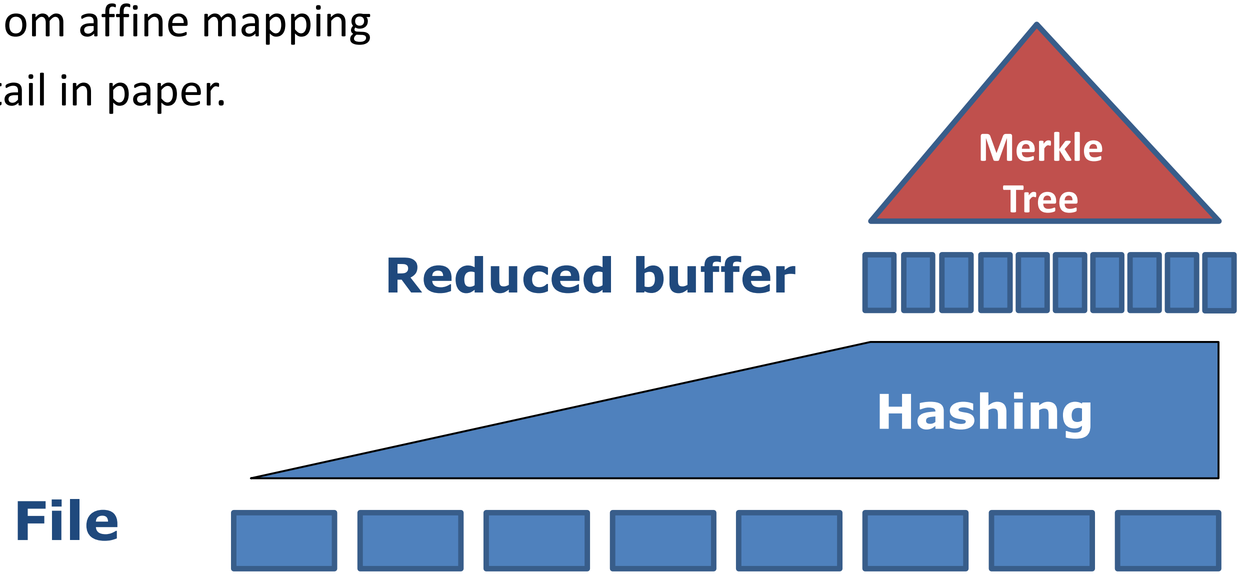
- Computing an erasure code for a large file.
  - No streaming solution (that we know of).
  - Need random-access to either input or output of the encoding procedure.
- Very expensive if file doesn't fit in memory.
  - Too many disk-seeks.

- Also, small space at client?

Construction 2: Hash & Merkle Tree

- Hashing to reduce file to a L bits buffer, then building Merkle-tree over the buffer.

- Requirement: min-entropy of the original file should not be reduced.
  - Using pairwise independent hashing h: $\{0,1\}^M \rightarrow \{0,1\}^L$
    - E.g., random affine mapping
    - More detail in paper.



Efficient Enough?

- Hashing output fits in memory, and can compute it in "streaming fashion". ☺

- But such implementation would be prohibitively expensive for large *M*, *L*.
  - File size *M*, buffer size *L*, hashing takes $\Omega(M \cdot L)$ time. ☹

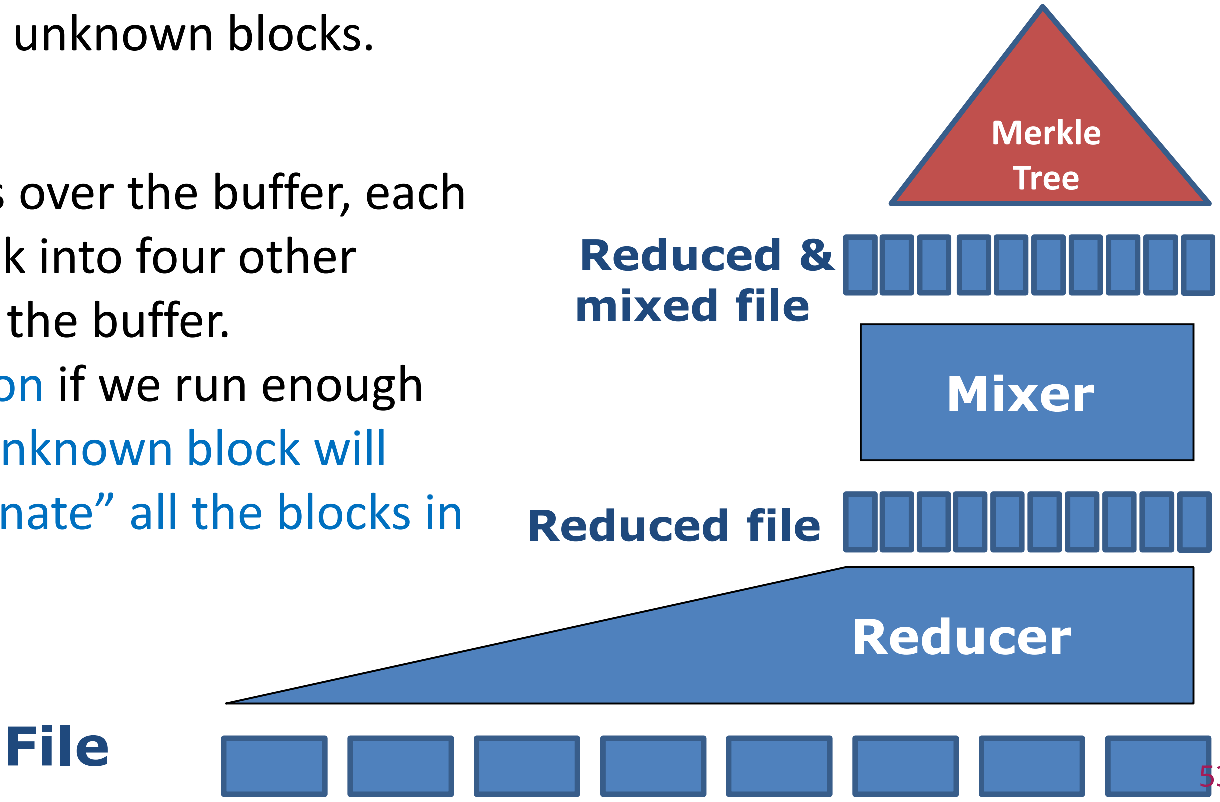
- Can we do better?

Construction 3: Reduce, Mix & Merkle

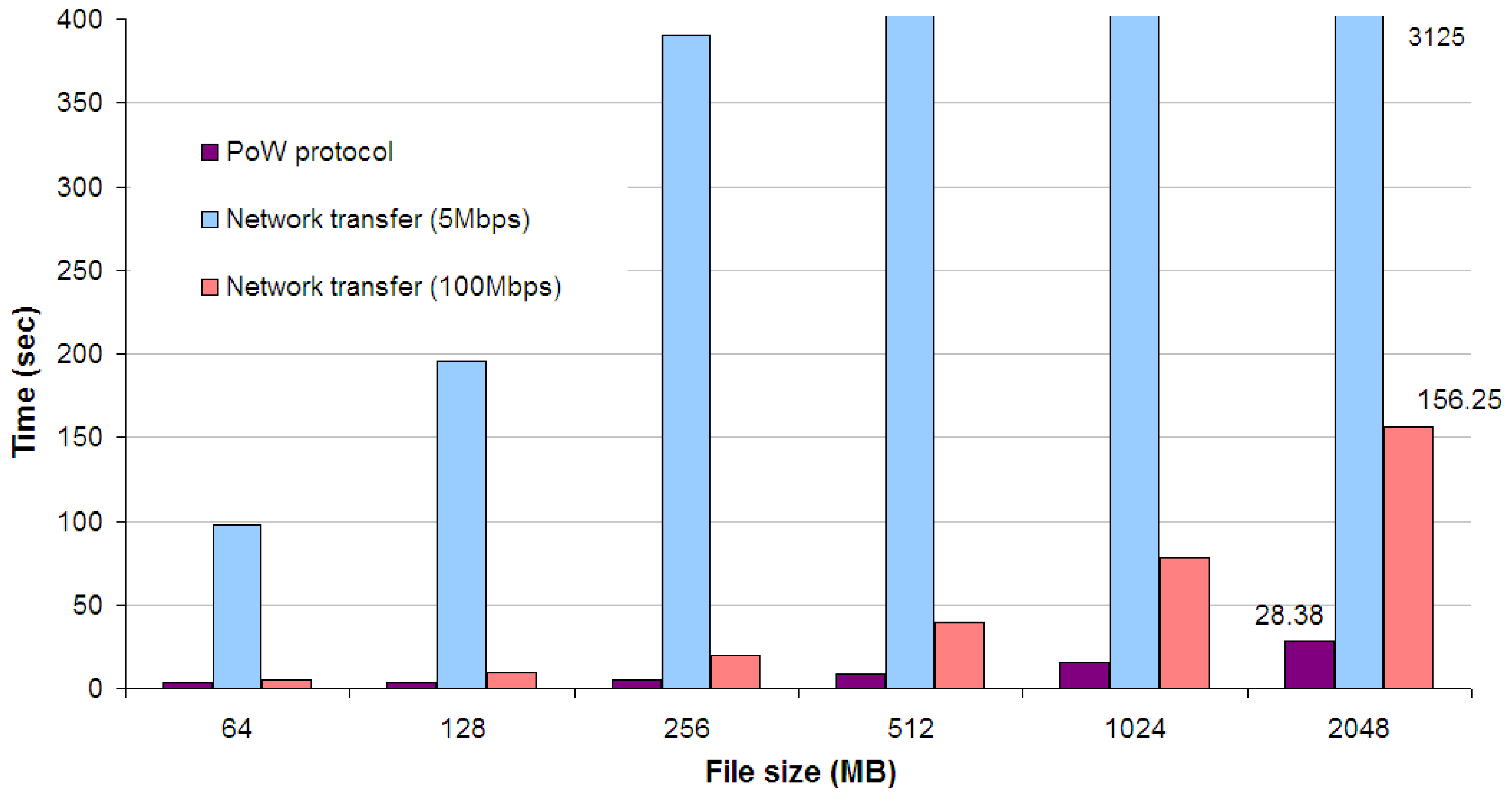
- Want to use a simpler length-reduction than universal hashing.
  - Goal: If adversary is missing even a small part of the file (after leakage), it will miss a large fraction of the reduced-length buffer.
- Authors design an efficient ad-hoc procedure, “hope that it works”.
  - They prove security against a certain class of input distributions, under a coding assumption

Construction 3: Reduce, Mix & Merkle

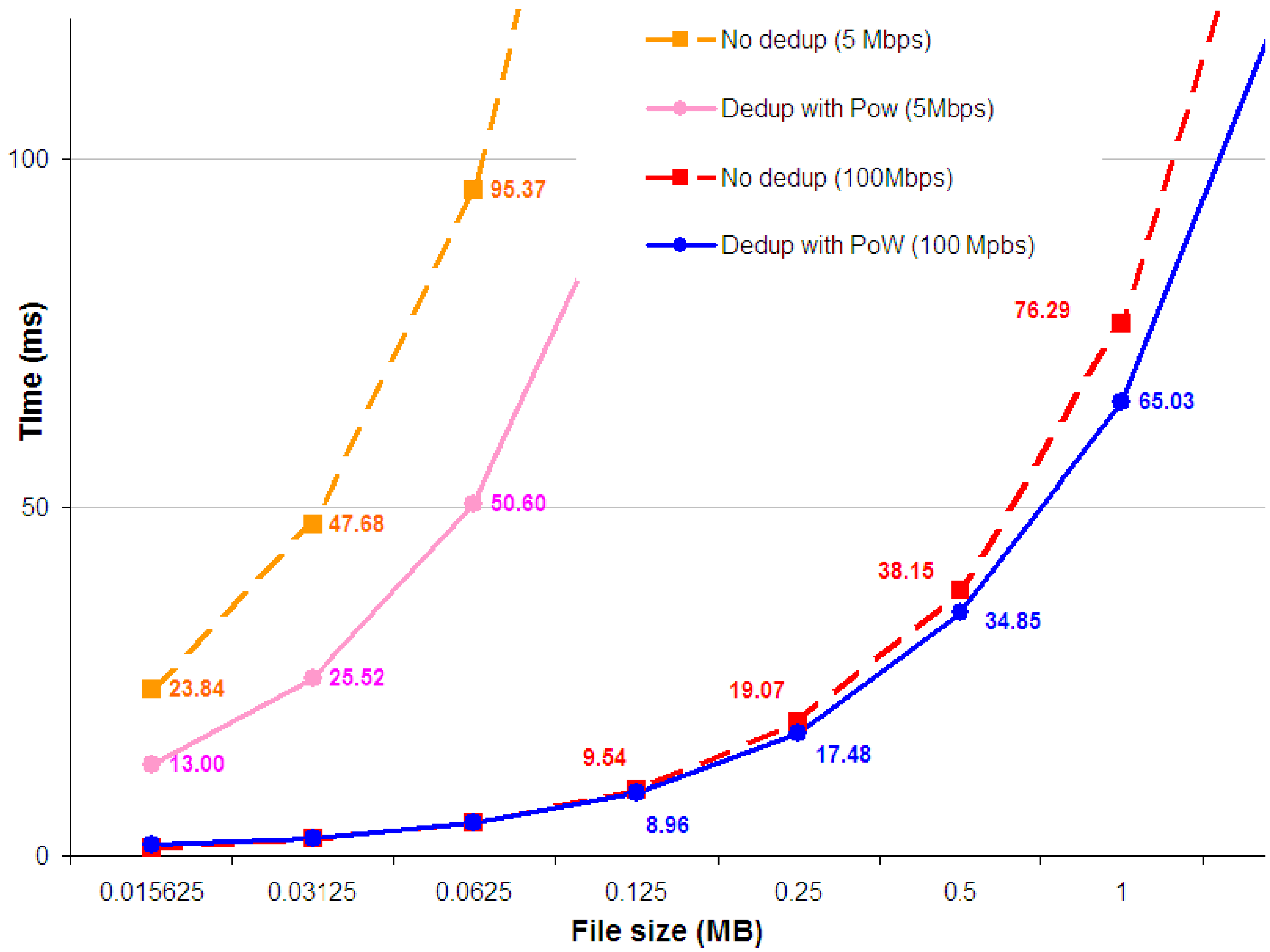
- Reducer:
  - XOR each block to a constant number (e.g., 4) of random locations in buffer.
  - Runs in O(M+L) time.
  - E.g., starting from a file with only a single unknown block, we end up with a buffer with only four unknown blocks.
- Add a mixing phase:
  - Make several passes over the buffer, each time XOR every block into four other random locations in the buffer.
  - Give us good diffusion if we run enough passes, since each unknown block will eventually "contaminate" all the blocks in the buffer.



Running PoW vs. Sending the File



When is it Worth the Effort?



Conclusion

- Deduplication offers huge savings and yet might leak information about other users.
- They put forward the notion of **proof-of-ownership (PoW)**, by which a client can prove to a server that it has a copy of a file without actually sending it.
  - This allows to defend attacks on file-deduplication systems where the attacker obtains a “short summary” of the file and uses it to fool the server into thinking that the attacker owns the entire file.

Message-Locked Encryption and Secure Deduplication

Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart

In *Proc. of* ***Eurocrypt****, 2013*

[Slides credits in part to BKR, EUROCRYPT’13]

Convergent Encryption

- CE seems to be widely used:
  - Cloud storage, filesystems, backup, etc.
- However, …
  - What kind of security can schemes like CE provide?
  - Are the deployed schemes/variants secure?
- No cryptographic treatment for deduplication over encrypted data.
  - Syntax of such schemes?
  - Best possible security?
  - How to support
    - Equality checking/deduplication?
    - Cross-user decryption?

This paper answers these questions!

This Work

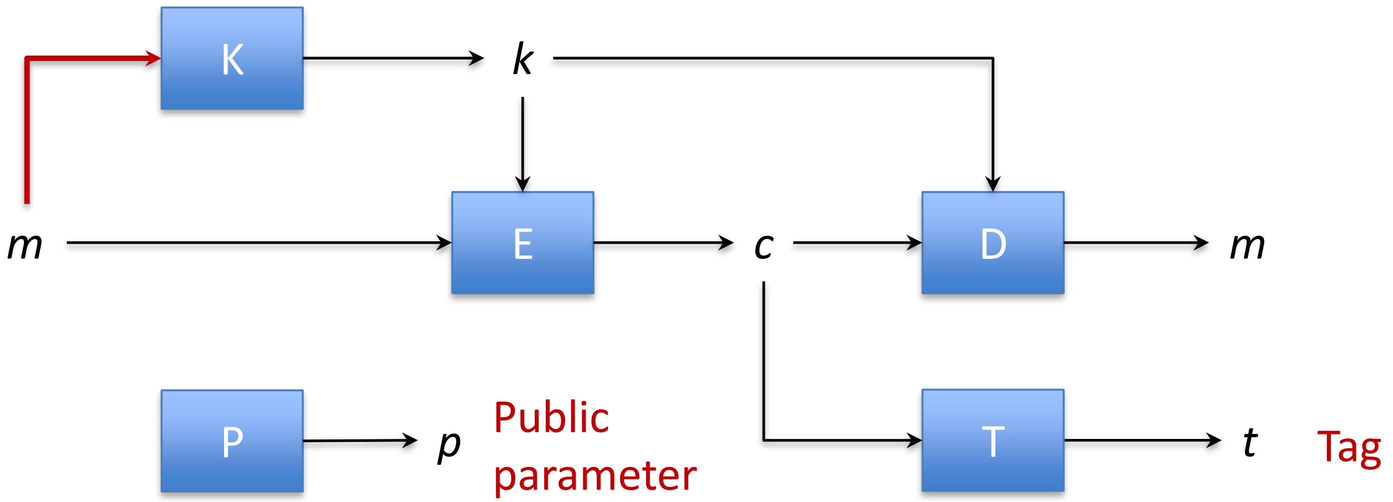
- **Message-locked encryption (MLE)**
  - Syntax and correctness
  - Security goals and notions

- **Practical contributions**
  - Attacks and proofs for CE and variants
  - New, faster schemes

Message-Locked Encryption (MLE)

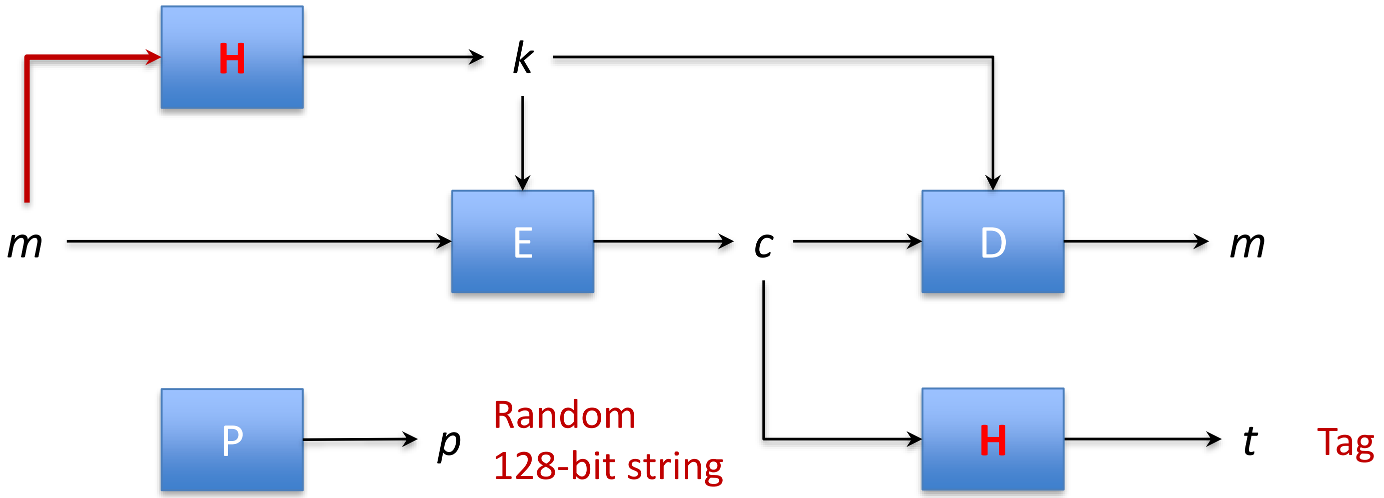
- A cryptographic framework for schemes which achieve deduplication over ciphertexts.
- MLE scheme: ***M*** = (P, K, E, D, T)
  - D and T are deterministic;
  - P, E, K can be randomized or deterministic
    - If K and E are deterministic, we say MLE is deterministic.
- Key used for encryption is derived from the message itself.

So named because the message is locked under itself.



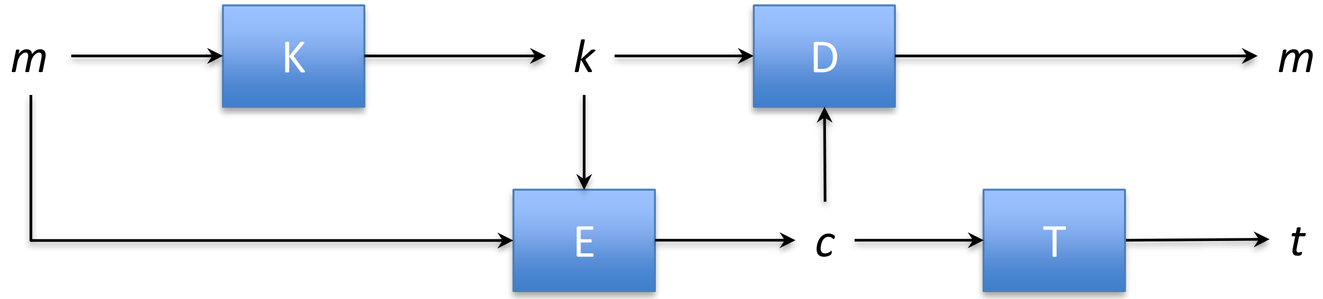
CE as an MLE scheme

- Recipe:
  - *H*: {0, 1}* → {0, 1}^{k}: Hash function
  - *SE* = (K, E, D): Encryption scheme with *k*-bit keys
- *CE* = (P, K, E, D, T)
  - *CE* is captured by the syntax as the MLE scheme that lets *k* = H(M) and tag T = H(C).



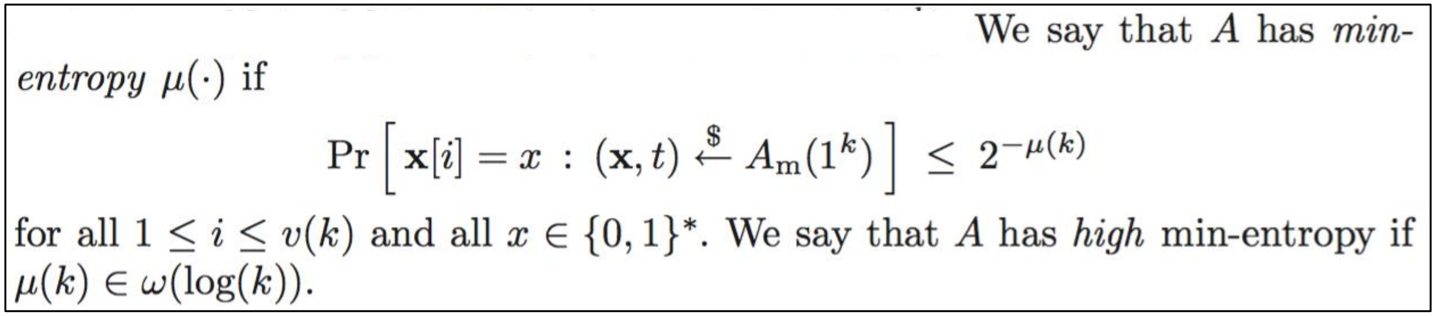
MLE Correctness

- **Decryption correctness**
  - Any key k derived from m can decrypt any m-ciphertext c.
  - $D(k, c) = m$ ∀ valid message m, $\forall\ k \in [K(m)]$, $\forall\ c \in [E(k, m)]$
- **Tag correctness**
  - All ciphertexts c over message m produce the same tag t.
  - $T(c_1) = T(c_2)\ \forall\ m, \forall\ k_1, k_2 \in [K(m)], \forall\ c_1 \in [E(k_1, m)], \forall\ c_2 \in [E(k_2, m)]$
- **Non-triviality**
  - All keys k are of the same, fixed length, and should be shorter than $|m|$.
  - $|K(m)| = \kappa\ \forall\ m, \forall\ k \in [K(m)]$



Privacy

- **No MLE scheme can achieve semantic-security-style privacy.**
- Indeed, if the target message *m* is drawn from a space *S* of size *s* then an adversary, given an encryption *c* of *m*, can recover *m* in O(*s*) trials.
  - For each *m'* ∈ *S*, test whether D(K(*m'*), *c*) = *c'*. If so, return *m'*.
- We therefore ask for the best possible privacy, namely semantic security when messages are **unpredictable** (have high min-entropy).
  - Similar concept appears in [BBO, CRYPTO'07], [BBNRSSY, ASIACRYPT'09], [BFOR, CRYPTO'08].

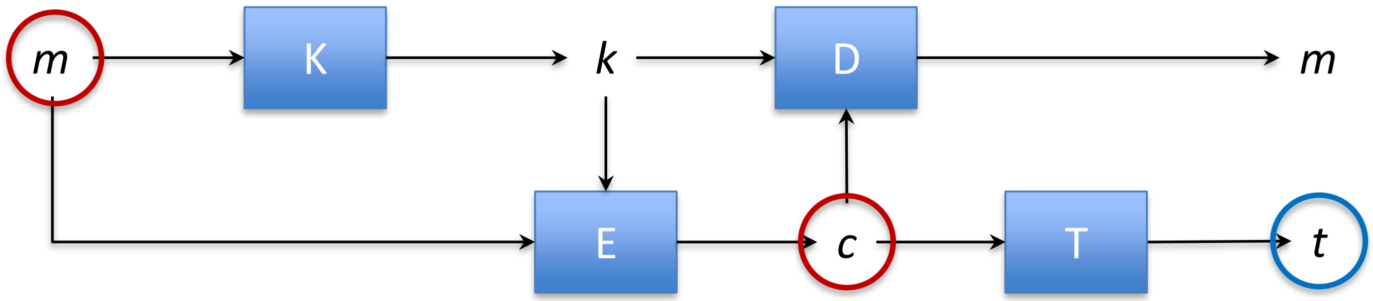


We say that A has *min-entropy* $\mu(\cdot)$ if

$$\Pr\left[\mathbf{x}[i] = x \ : \ (\mathbf{x}, t) \xleftarrow{\$} A_{\mathrm{m}}(1^k)\right] \ \le \ 2^{-\mu(k)}$$

for all $1 \le i \le v(k)$ and all $x \in \{0,1\}^*$. We say that A has *high* min-entropy if $\mu(k) \in \omega(\log(k))$.

Security, Informally

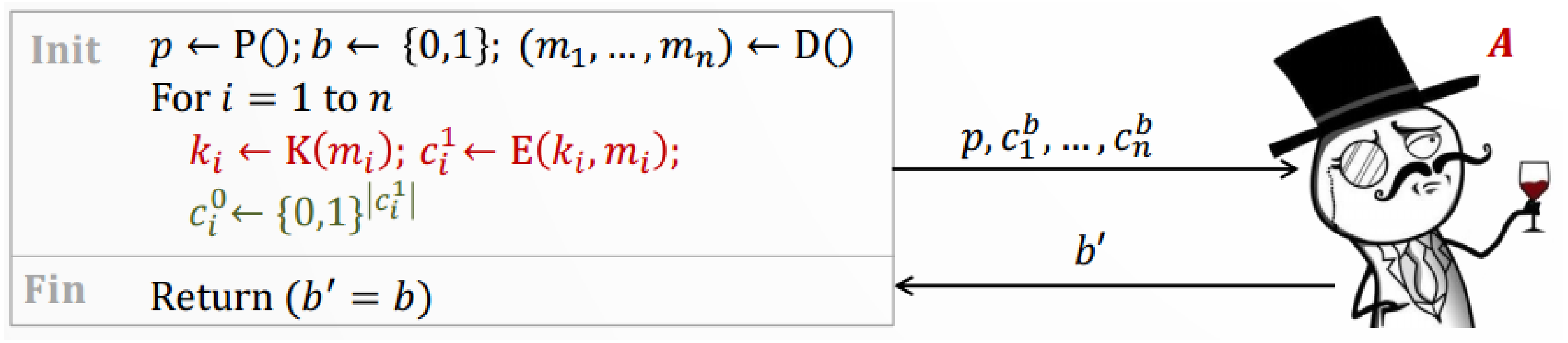


- Privacy
  - PRV-CDA (chosen-distribution attack) notion
    - Encryptions of two unpredictable(high min-entropy) messages should be indistinguishable.
  - **PRV$-CDA** notion
    - “**$**” means the encryption of an unpredictable message *m* must be indistinguishable from **a random string of the same length.**

Privacy: The PRV$-CDA Notion

- No efficient adversary can distinguish encryptions of *unpredictable message* from *random strings*.
- PRV$-CDA(A, D)

MLE scheme *M* = (P, E, K, D, T)



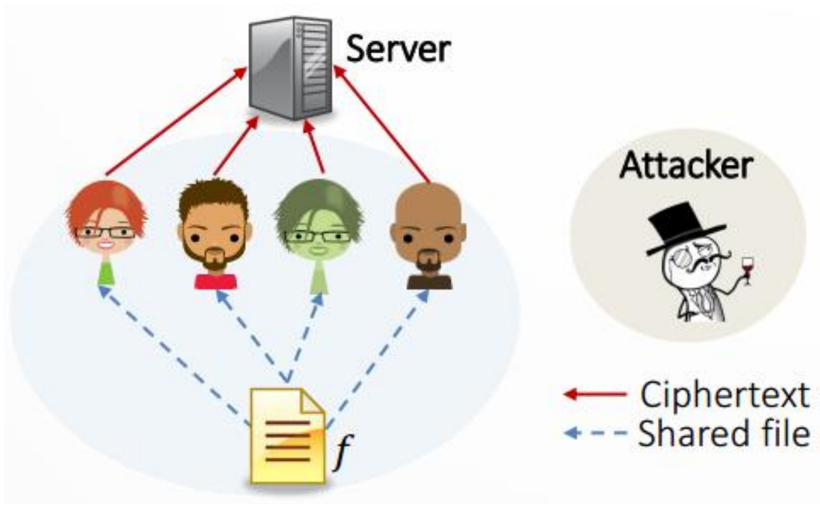
```
Init   $p \leftarrow P(); b \leftarrow \{0,1\}; (m_1, \dots, m_n) \leftarrow D()$
       For $i = 1$ to $n$
          $k_i \leftarrow K(m_i); c_i^1 \leftarrow E(k_i, m_i);$
          $c_i^0 \leftarrow \{0,1\}^{|c_i^1|}$
Fin    Return $(b' = b)$
```

- Security: No efficient ***A*** has non-negligible advantage for any unpredictable D.
  - Details in paper also shows PRV$-CDA implies PRV$-CDA-A (adaptive), a preferred design.

Deduplicability vs. Privacy

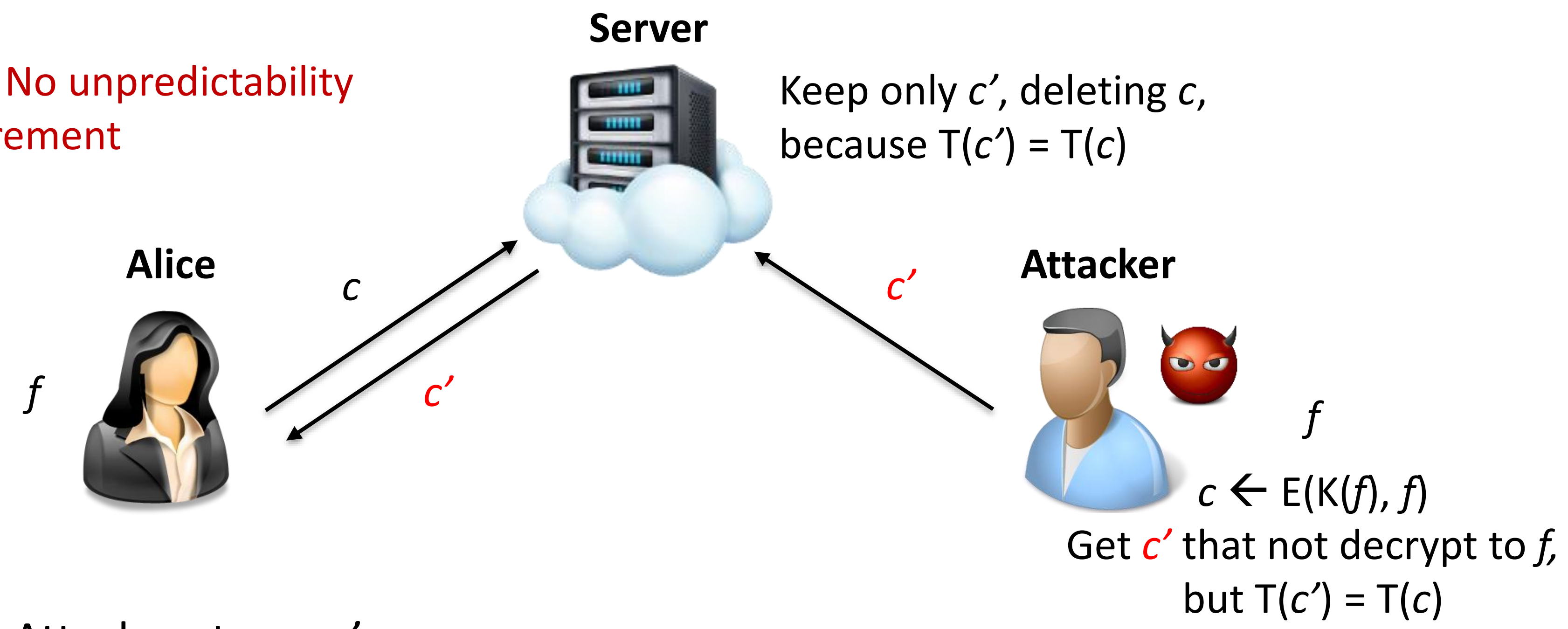
Deduplication	Privacy
Only when messages repeat	Only when messages unpredictable

- A possible contradiction? NO!
- Data unpredictable to attacker, not to legitimate clients.
- Large random file *f*:
  - Shared among group of clients;
  - Unknown to attacker.
- Inherent to secure deduplication => **PRV$-CDA provides best possible security**.



Security for predictable messages ------> DupLESS (later will be discussed)

Duplicate Faking Attacks



Server
Note: No unpredictability requirement
Keep only *c′*, deleting *c*, because T(*c′*) = T(*c*)
Alice
f
c
c′
c′
Attacker
f
c ← E(K(*f*), *f*)
Get *c′* that not decrypt to *f*, but T(*c′*) = T(*c*)

1. Attacker stores *c′*
2. Alice tries to store c, server already has a matching ciphertext *c′*
3. When Alice downloads *c′*, it decrypts to *f′* ≠ *f*

This is a serious concern, and not mere speculation.
They want to rule out these types of integrity violations.

Tag Consistency

- Tag consistency (TC) aims to provide security against duplicate faking attacks in which a legitimate message is undetectably replaced by a fake one.

- Notion TC asks that it be hard to create (m, c) such that T(c) = T(E(K(m), m)), but D(K(m), c) is a string different from m.
  - I.e., an adversary cannot make an honest client recover an incorrect message, which is different from the one it uploaded.

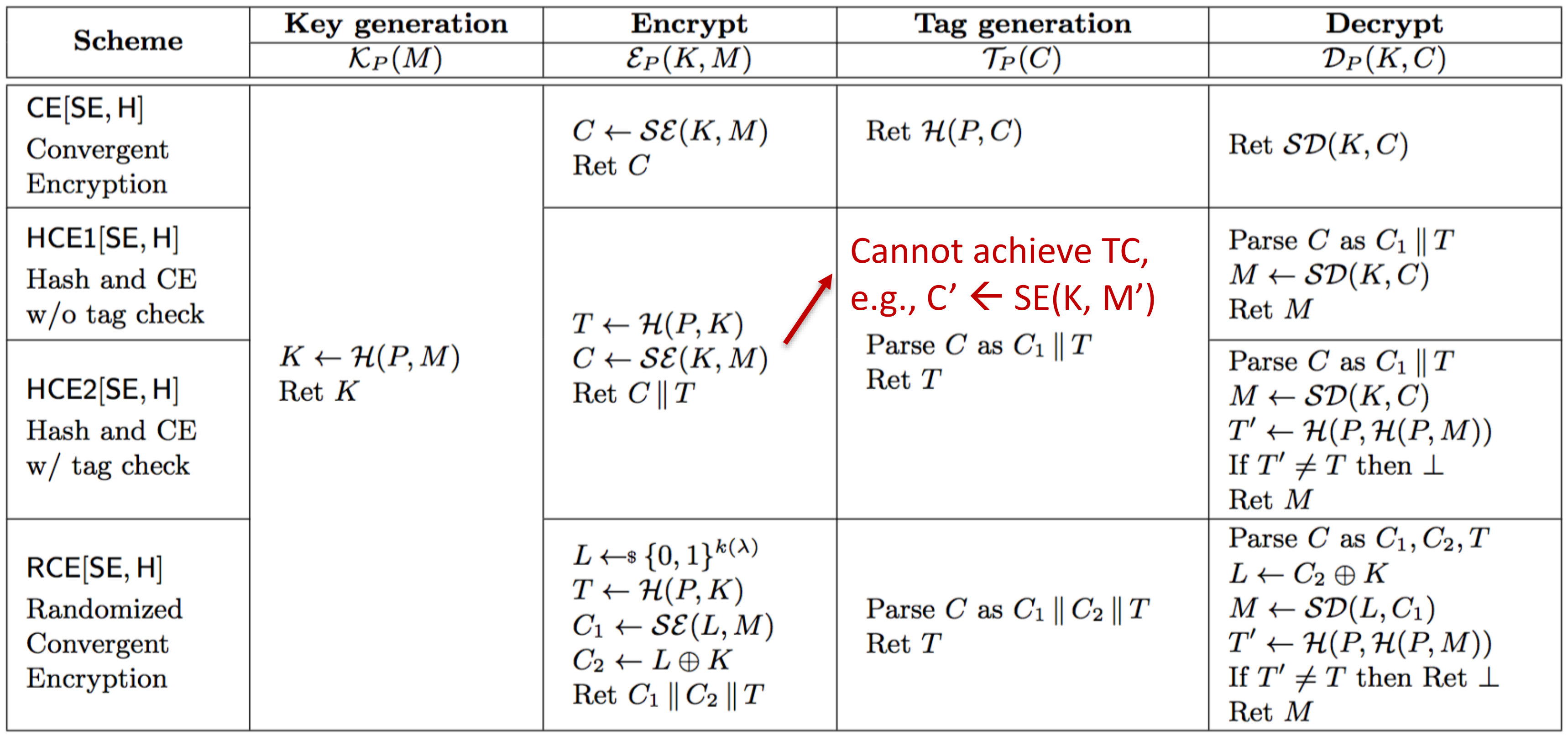
This Work

- **Message-locked encryption (MLE)**
  - Syntax and correctness
  - Security goals and notions

- **Practical contributions**
  - Attacks and proofs for CE and variants
  - New, faster schemes

All MLE schemes achieves PRV$-CDA.

MLE Schemes



Scheme	Key generation $\mathcal{K}_P(M)$	Encrypt $\mathcal{E}_P(K, M)$	Tag generation $\mathcal{T}_P(C)$	Decrypt $\mathcal{D}_P(K, C)$	
CE[SE, H] Convergent Encryption	$K \leftarrow \mathcal{H}(P, M)$ Ret K	$C \leftarrow \mathcal{SE}(K, M)$ Ret C	Ret $\mathcal{H}(P, C)$	Ret $\mathcal{SD}(K, C)$	TC
HCE1[SE, H] Hash and CE w/o tag check		$T \leftarrow \mathcal{H}(P, K)$ $C \leftarrow \mathcal{SE}(K, M)$ Ret $C \\| T$	Cannot achieve TC, e.g., C' ← SE(K, M') Parse C as $C_1 \\| T$ Ret T	Parse C as $C_1 \\| T$ $M \leftarrow \mathcal{SD}(K, C)$ Ret M	
HCE2[SE, H] Hash and CE w/ tag check				Parse C as $C_1 \\| T$ $M \leftarrow \mathcal{SD}(K, C)$ $T' \leftarrow \mathcal{H}(P, \mathcal{H}(P, M))$ If $T' \neq T$ then $\perp$ Ret M	TC
RCE[SE, H] Randomized Convergent Encryption		$L \leftarrow_{\$} \{0,1\}^{k(\lambda)}$ $T \leftarrow \mathcal{H}(P, K)$ $C_1 \leftarrow \mathcal{SE}(L, M)$ $C_2 \leftarrow L \oplus K$ Ret $C_1 \\| C_2 \\| T$	Parse C as $C_1 \\| C_2 \\| T$ Ret T	Parse C as C_1, C_2, T $L \leftarrow C_2 \oplus K$ $M \leftarrow \mathcal{SD}(L, C_1)$ $T' \leftarrow \mathcal{H}(P, \mathcal{H}(P, M))$ If $T' \neq T$ then Ret $\perp$ Ret M	TC

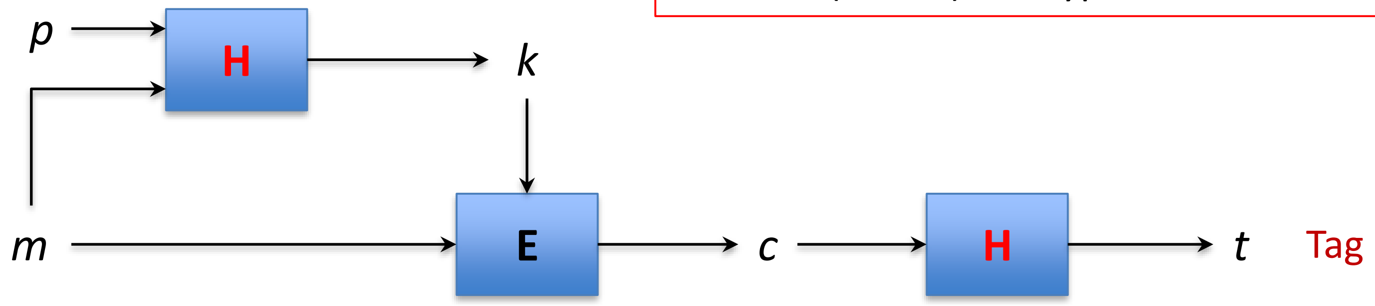
Figure 4: MLE schemes built using symmetric encryption scheme SE = $(\mathcal{SK}, \mathcal{SE}, \mathcal{SD})$ and hash function family H = $(\mathcal{HK}, \mathcal{H})$. All schemes inherit their message space from SE (typically $\{0,1\}^*$), use as parameter generation $\mathcal{HK}$, and share a common key generation algorithm. HCE1 and HCE2 additionally use the same encryption and tag generation algorithms.

Convergent Encryption

- *CE* = (P, K, E, D, T)
- Encryption in CE

Recipe:

1. *H*: $\{0, 1\}^* \rightarrow \{0, 1\}^k$: Hash function
2. *SE* = (K, E, D): Encryption scheme with *k*-bit keys



- Theorem: *CE* is PRV$-CDA-secure in the **RO** model if *SE* is Real-or-Random secure and key-Recovery secure.
- Theorem: *CE* is TC secure if *H* is CR secure.

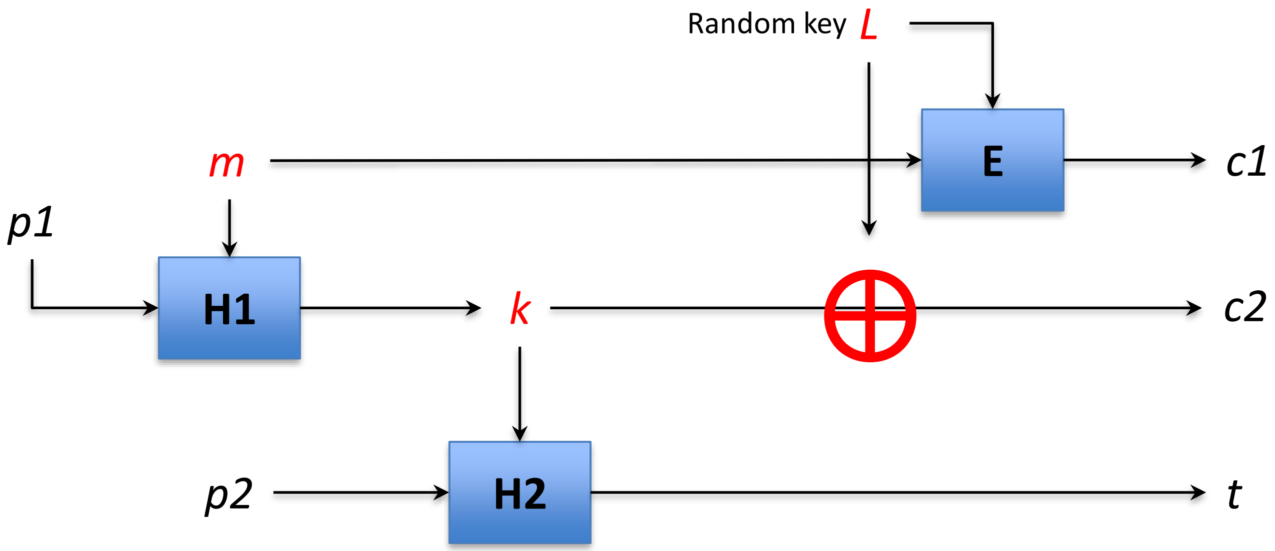
Collision resistance

Randomized CE

Recipe:

1. *H1, H2*: $\{0, 1\}^* \rightarrow \{0, 1\}^k$: Hash functions
2. *SE* = (K, E, D): Encryption scheme with *k*-bit keys

- One pass, randomized MLE scheme.
- Key generation and encryption:



- Theorem: *RCE* is PRV$-CDA-secure secure in the **RO** model if *SE* is Real-or-Random secure and Key-Recovery secure.
- Theorem: *RCE* is TC secure if *H* are CR secure.

Conclusion

- They formalize a new cryptographic primitive, **Message-Locked Encryption (MLE)**, where the key under which encryption and decryption are performed is itself derived from the message.
  - MLE provides a way to achieve secure deduplication (space-efficient secure outsourced storage), a goal currently targeted by numerous cloud-storage providers.
- They also provide definitions both for privacy and for a form of integrity that is called **tag consistency**.

- They do not focus on the server-side or client-side designs.

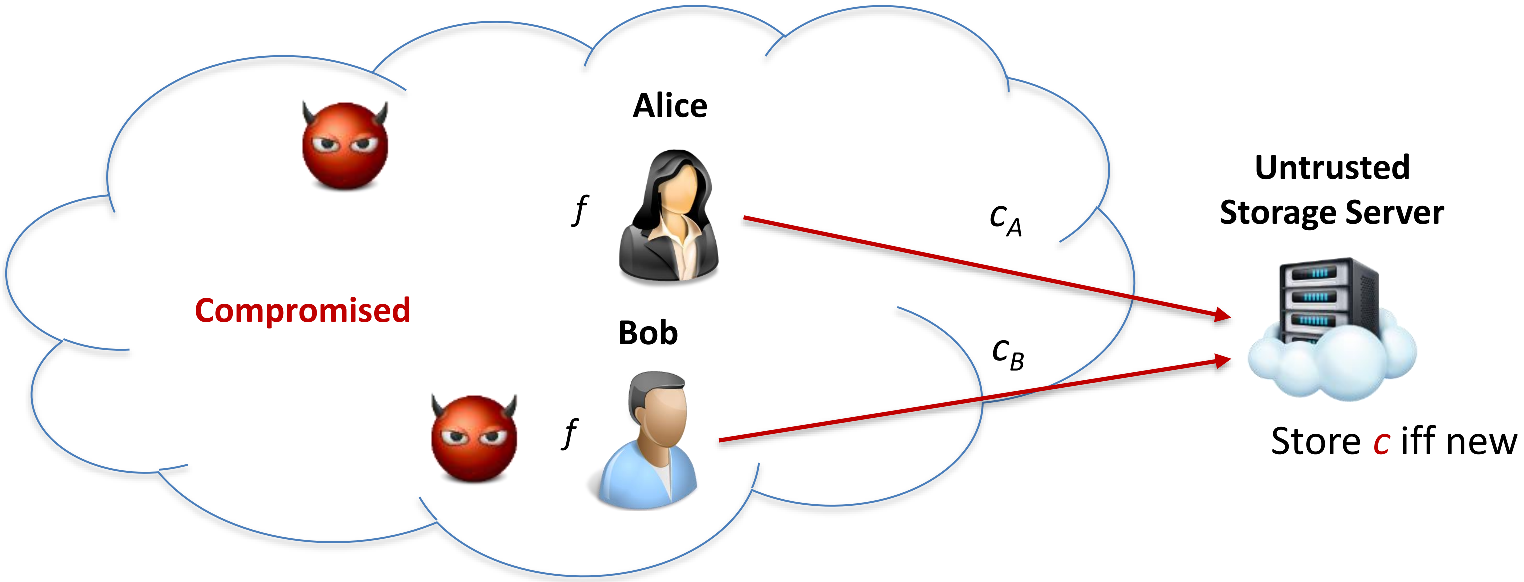
DupLESS: Server-Aided Encryption for Deduplicated Storage

Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart

In *Proc. of **USENIX Security**, 2013*

[Slides credits in part to BKR, USENIX Security'13]

Design Goal



1. **Secure deduplication**: dedup + strong security against untrusted storage;
2. **Compromise resilience**: client might be compromised to decrypt the users' data.
   - E.g., when all users share one secret key, all data is insecure even if one client is compromised.

Attempts

- **Client specific keys**
  - Cross-user deduplication cannot work.
- **Network-wide key**
  - No compromise resilience.
    - All data is insecure even if one client is compromised.
- **Convergent encryption**
  - Support deduplication: Everyone encrypting f gets ciphertext.
  - Compromise resilience: No system-wide secret.
  - However, brute-force attack exists …

Brute-Force Attacks in CE

- The dirty secret of convergent encryption.
  - If m comes from S = {}, attacker can recover m from $c \leftarrow E(H(m), m)$.

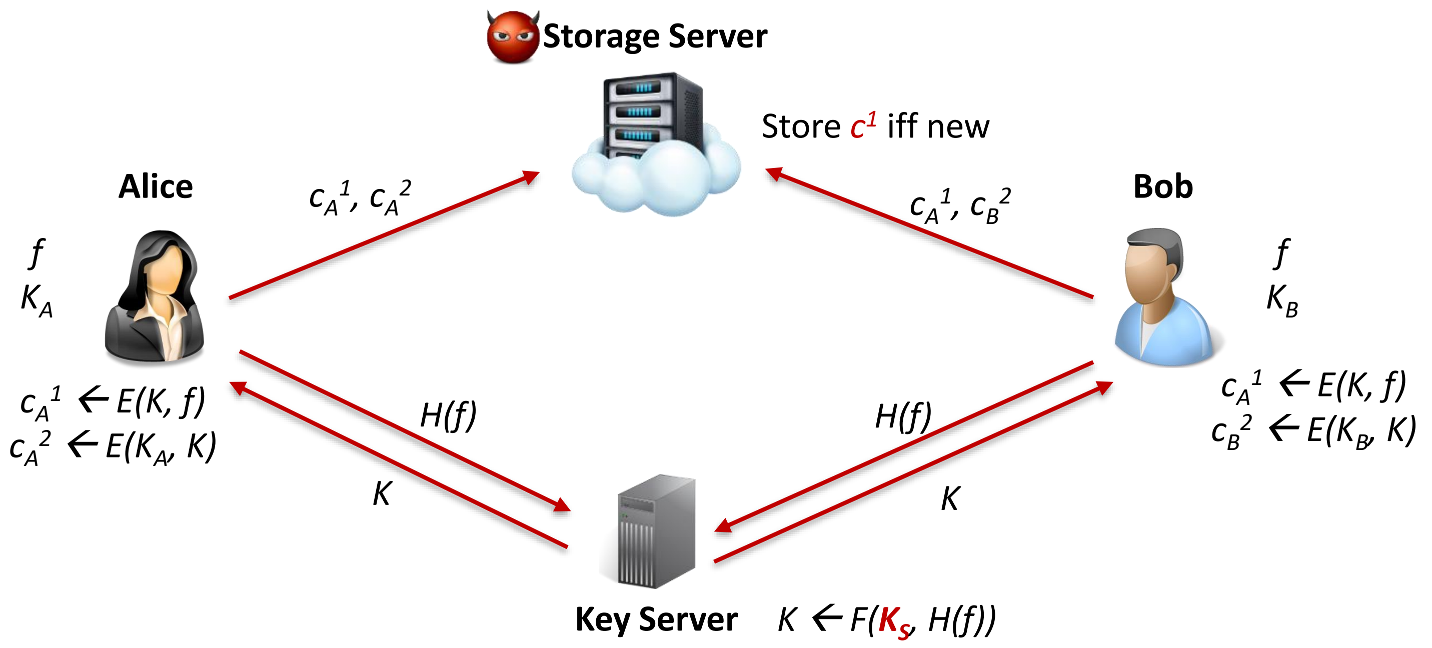
    -
      ```
      BruteForce_S(c): For $m_i \in S$ do
                           $m' \leftarrow D(H(m_i), c)$
                           If $m_i = m'$ then return $m_i$
      ```
  - Attacker runs in time proportional to $|S|$.
  - Security only when $|S|$ too large to exhaust. $\leftarrow$-------- **Unpredictable**
- **However, real files are often predictable!**
- Message-locked encryption:
  - Generalizes convergent encryption;
  - Captures properties needed for secure deduplication.
- Brute-force attacks exist for all message-locked encryption schemes.

Comparison

	Systems			
Systems	**Client specific keys**	**Network wide key**	**Convergent encryption**	**DupLESS**
Deduplication	N	Y	Y	Y
Compromise resilience	Y	N	Y	Y
Brute-force attack resilience	Y	Y	N	Y

DupLESS: First to achieve all three properties!

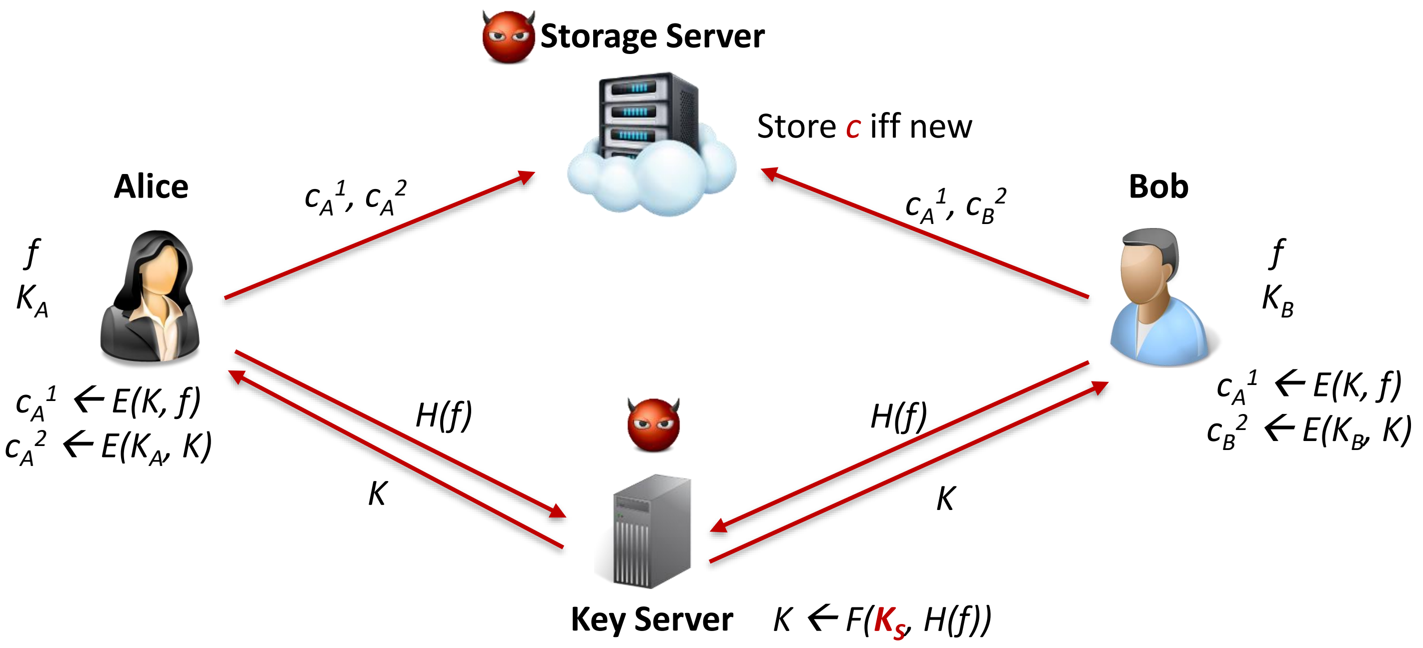
Key Insight: Server-aided Encryption



Storage Server
Store c^1 iff new
Alice
f
K_A
$c_A^1 \leftarrow E(K, f)$
$c_A^2 \leftarrow E(K_A, K)$
c_A^1, c_A^2
Bob
f
K_B
$c_A^1 \leftarrow E(K, f)$
$c_B^2 \leftarrow E(K_B, K)$
c_A^1, c_B^2
$H(f)$
K
$H(f)$
K
Key Server
$K \leftarrow F(\mathbf{K_S}, H(f))$

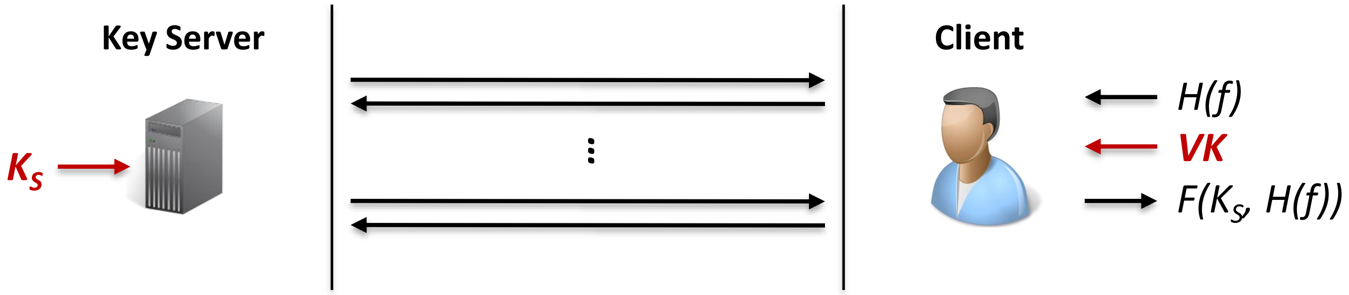
- *F*: A pseudorandom function (PRF)
  - E.g., HMAC [SHA256].
- **Deduplication**:
  - Any client encrypting *f* produces same c^1.
  - c^2 ciphertexts cannot be deduplicated, but they are tiny.

Dealing With Brute-Force Attacks


Storage Server
Store *c* iff new
Alice
Bob
f
K_A
f
K_B
c_A^1, c_A^2
c_A^1, c_B^2
$c_A^1 \leftarrow E(K, f)$
$c_A^2 \leftarrow E(K_A, K)$
$c_A^1 \leftarrow E(K, f)$
$c_B^2 \leftarrow E(K_B, K)$
$H(f)$
K
$H(f)$
K
Key Server
$K \leftarrow F(K_S, H(f))$

- The key server becomes a single point of failure, violating our goal of compromise resilience:
  - An attacker can obtain hashes of files after gaining access to it, and can recover files with brute-force attacks.
- Promising approach: **obliviously evaluating *f*.**

Oblivious PRF (OPRF) Protocol



- Verifiable OPRF: Client can verify $K = F(K_S, H(f))$
- Security, informally:
  1. F is a PRF (when not given ***VK***);
  2. Server learns nothing, client learns only K;
  3. Client can detect when server does not return K.

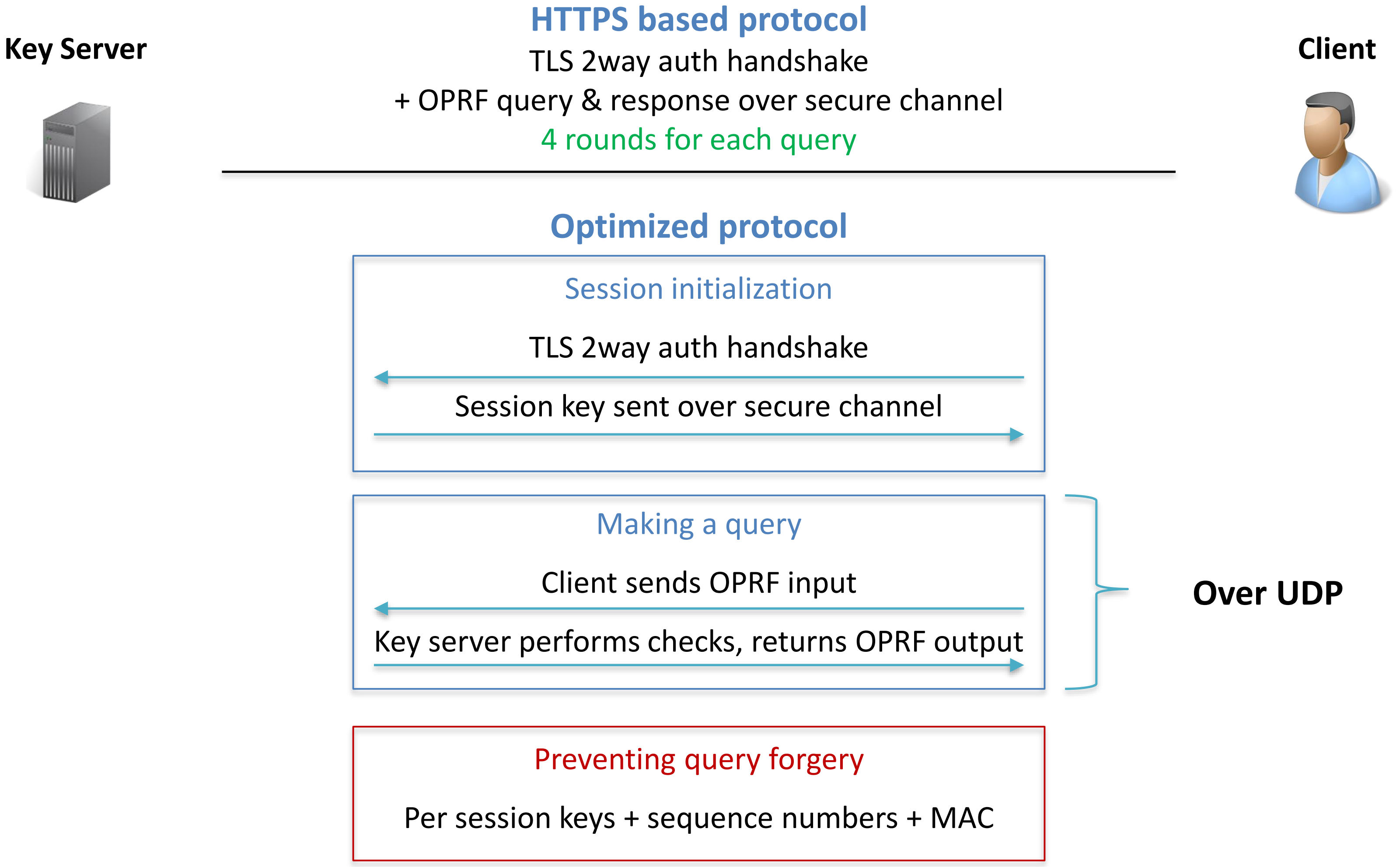
RSA-OPRF Protocol

- Based on **RSA blind signature**:
  - The key generation **Kg** outputs PRF key (N, d) and verification key N.
  - The client uses two hash functions:
    - $H: \{0, 1\}^* \rightarrow Z_N$, and $G: Z_N \rightarrow \{0, 1\}^k$.

Client EvC(N, M)		Key server *EvS*(N, d)
If $e \leq N$ then ret $\perp$		
$r \xleftarrow{\$} Z_N$		
$h \leftarrow H(M)$		
$x \leftarrow h \cdot r^e \bmod N$	$x \rightarrow$	
	$\leftarrow y$	$y \leftarrow x^d \bmod N$
$z \leftarrow y \cdot r^{-1} \bmod N$		
If $z^e \bmod N \mathrel{!=} h$ then ret $\perp$ Else ret $G(z)$		

The public RSA exponent e is fixed as part of the scheme. Key generation algorithm with input e will output N, d such that $ed \equiv 1 \bmod \Phi(N)$, where modulus N is the product of two distinct primes of roughly equal length and $N < e$.

Client-KS (Key Server) Protocol



Key Server

Client

HTTPS based protocol

TLS 2way auth handshake

+ OPRF query & response over secure channel

4 rounds for each query

Optimized protocol

Session initialization

TLS 2way auth handshake

Session key sent over secure channel

Making a query

Client sends OPRF input

Key server performs checks, returns OPRF output

Over UDP

Preventing query forgery

Per session keys + sequence numbers + MAC

Assume a CA provides the KS and clients with verifiable TLS certificates.

1 round for each query

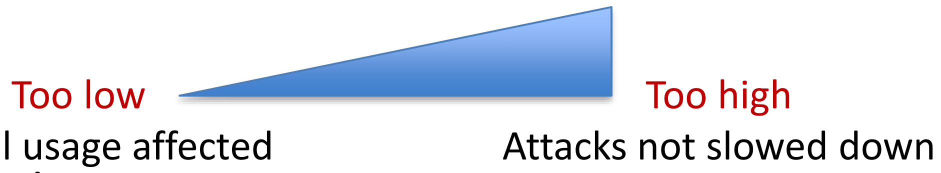
Key server (KS) Performance

	Operation	Latency (ms)
Naive HTTPS based	Query response (low load)	374 ± 34
Optimized	Initialization	278 ± 56
	Query response (low load)	83 ± 16
	Query response (heavy load)	118 ± 37
	Ping times	78 ± 01

- KS is deployed on an Amazon EC2 *m1.Large* instance.
- Heavy load ≈ 3k queries per second.

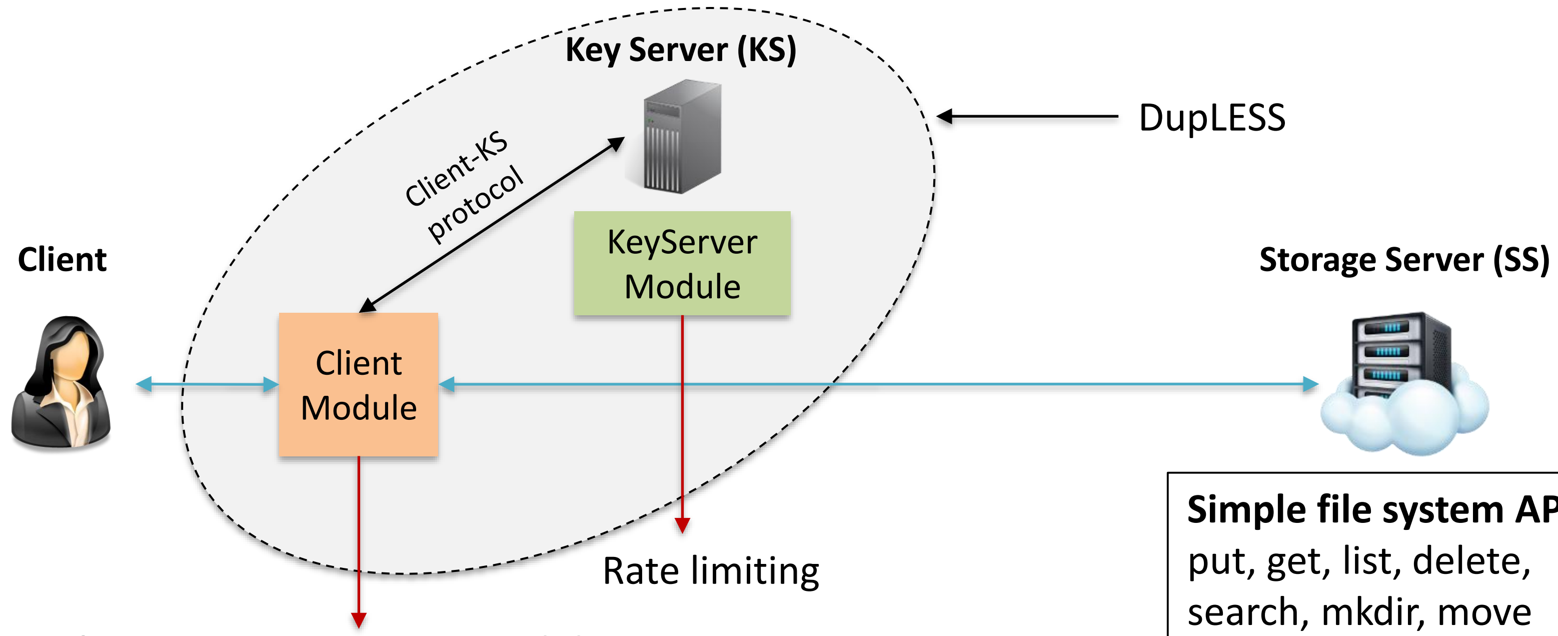
Rate Limiting

- Goal: slow down online brute-force trails from attacker controlled clients.
- Strategy: limit clients to q queries per epoch, where each epoch lasts τ units of time.
  - Setting bound q:



  - Setting epoch duration τ:
    - Must handle bursty workloads;
    - Systems exhibit periodic patterns, e.g., 1 week.
  - When they exceed their budgets (q queries), we can build clients to simply continue with randomized encryption.
    - Thereby alleviating KS availability issues for a conservative choice of q.
- Rate limiting can slow down brute-force attacks by 4000x.

DupLESS System Design



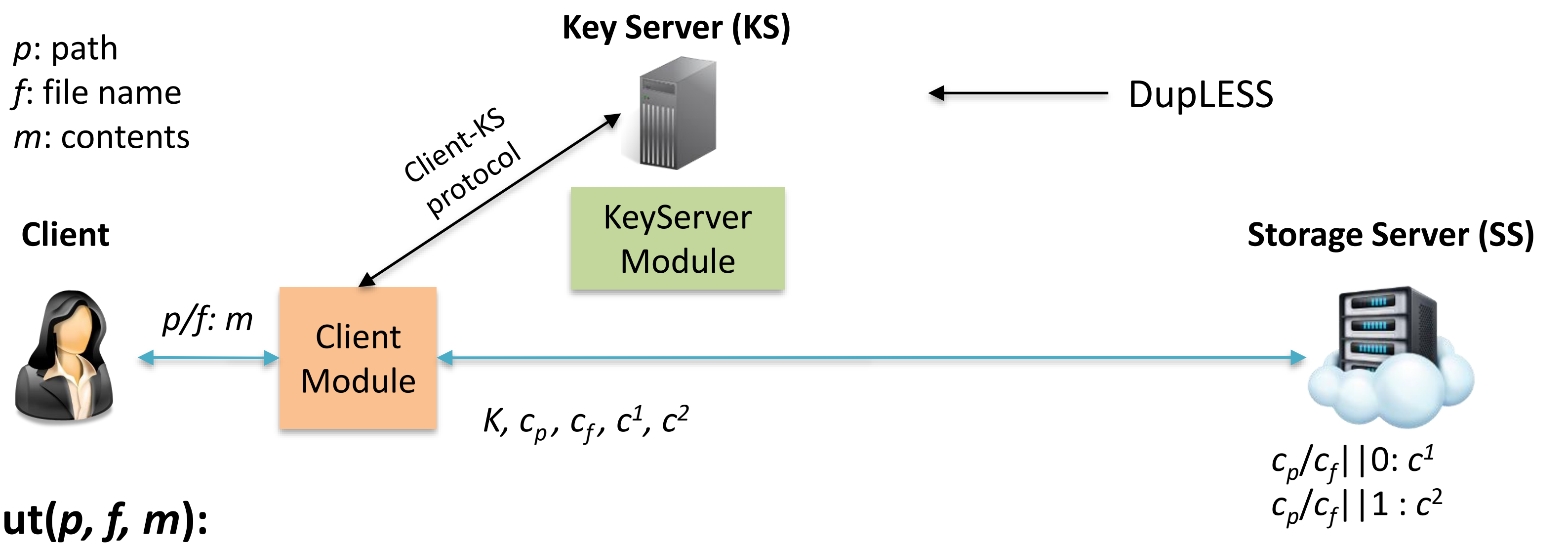
Simple file system API:
put, get, list, delete,
search, mkdir, move

Implement API over encrypted data:

- Encrypt and decrypt files;
- Handle file names and paths;
- Run transparently:
  - ✓ Low overhead;
  - ✓ Works when KS is down;
  - ✓ No client-side state.

Put Query in DupLESS

p: path
f: file name
m: contents



put(*p, f, m*):

1. Derive key K for m from KS;
2. $c_p \leftarrow \mathrm{DAE}(K_A, p)$, $c_f \leftarrow \mathrm{DAE}(K_A, f)$;
3. If not canDedup(p, f, m), then pick K at random;
4. $c^1 \leftarrow \mathrm{E}(K, m)$, $c^2 \leftarrow \mathrm{E}(K_A, K)$;
5. $\mathrm{SSput}(c_p, c_f||0, c^1)$, $\mathrm{SSput}(c_p, c_f||1, c^2)$.

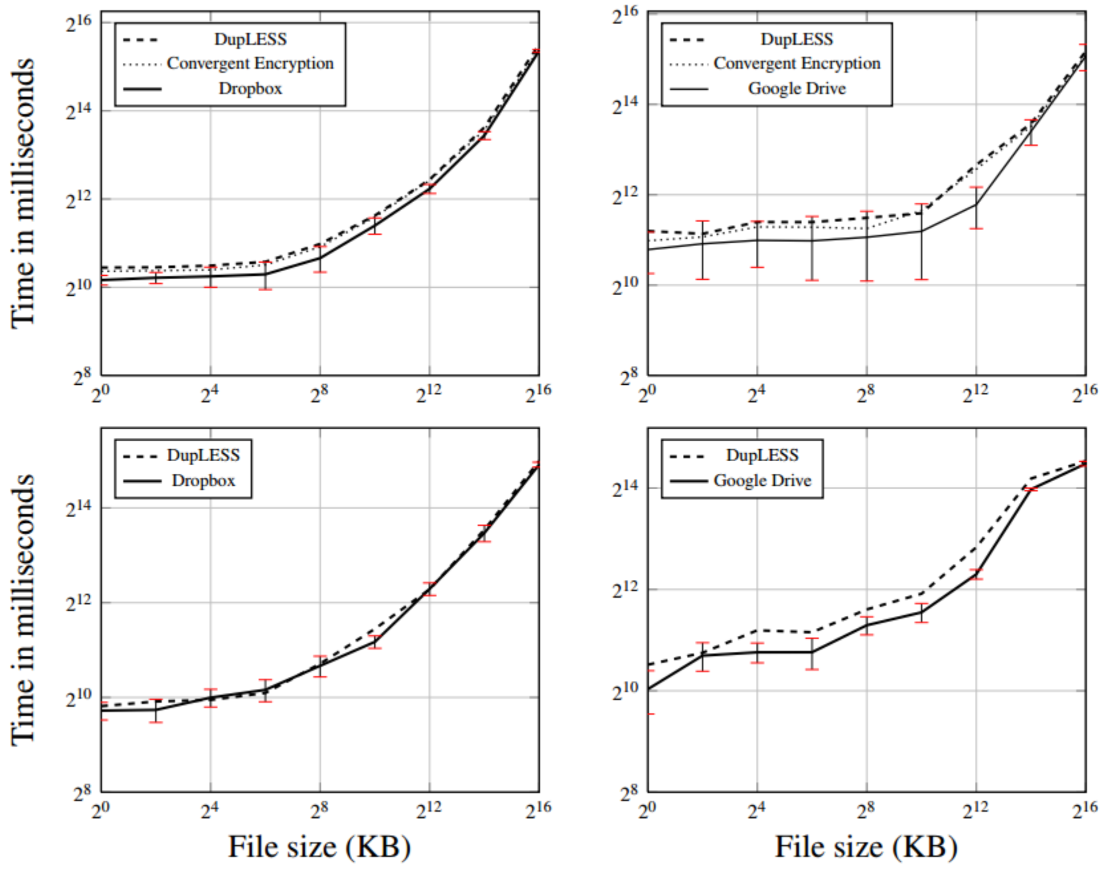
Deterministic authenticated encryption *[Rogaway et al. EUROCRYPT'06]*

Dedup heuristics, e.g., file length

Performance: Latency

Put

Get



- DupLESS client:
  - Written in Python, command-line interface;
  - Dropbox and Google Drive can work as storage service.

Conclusion

- DupLESS: encrypted deduplication with the aid of a key server.
  - First solution to provide secure deduplication + compromise resilience.
  - Can be deployed transparently over existing systems:
    - Implementations over Dropbox, Google Drive.
  - Nominal performance overhead over plaintext deduplication.
  - Storage saving match plaintext deduplication.

- Code available at:
  - http://cseweb.ucsd.edu/~skeelvee/dupless/

Secure Deduplication of Encrypted Data without Additional Independent Servers

Jian Liu, N. Asokan, and Benny Pinkas

In *Proc. of* ***ACM CCS****, 2015*

Target Problem

- Deduplication for **predictable** data needs a key server.
  - E.g., DupLESS.
- The authors want to **avoid additional server** by using a cryptographic primitive known as **Password Authenticated Key Exchange (PAKE)***, which allows two parties to agree on a session key iff they share a short secret (i.e., “password”).
  - The scheme allows a client uploading an existing file to securely obtain the encryption key that was used by the client who has previously uploaded that file.
    - This key is chosen randomly by the initial uploader.

*S. M. Bellovin and M. Merritt. “Encrypted key exchange: password-based protocols secure against dictionary attacks.” In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84, 1992.

Design Overview

- Directly uploading a file hash to locate duplicate is insecure.
  - Because a compromised server can easily mount an **offline brute-force attack** on the hash h if F is **predictable**.
- The proposed design overview:
  - A client uploading a file first sends a short hash of this file (10-20 bits long) to the server.
    - Using short hash, which is high collision rate and short output.
    - **Due to the high collision rate of short hash, server cannot use it to reliably guess the content of F offline.**
  - The server identifies other clients whose files have the same short hash, and let them run a single round PAKE protocol (routed through the server) with the uploader using the (long, but possibly low entropy) hashes of their files as the "passwords".
  - Finally, the uploader gets the key of another client iff their files are identical. Otherwise, it gets a random key.

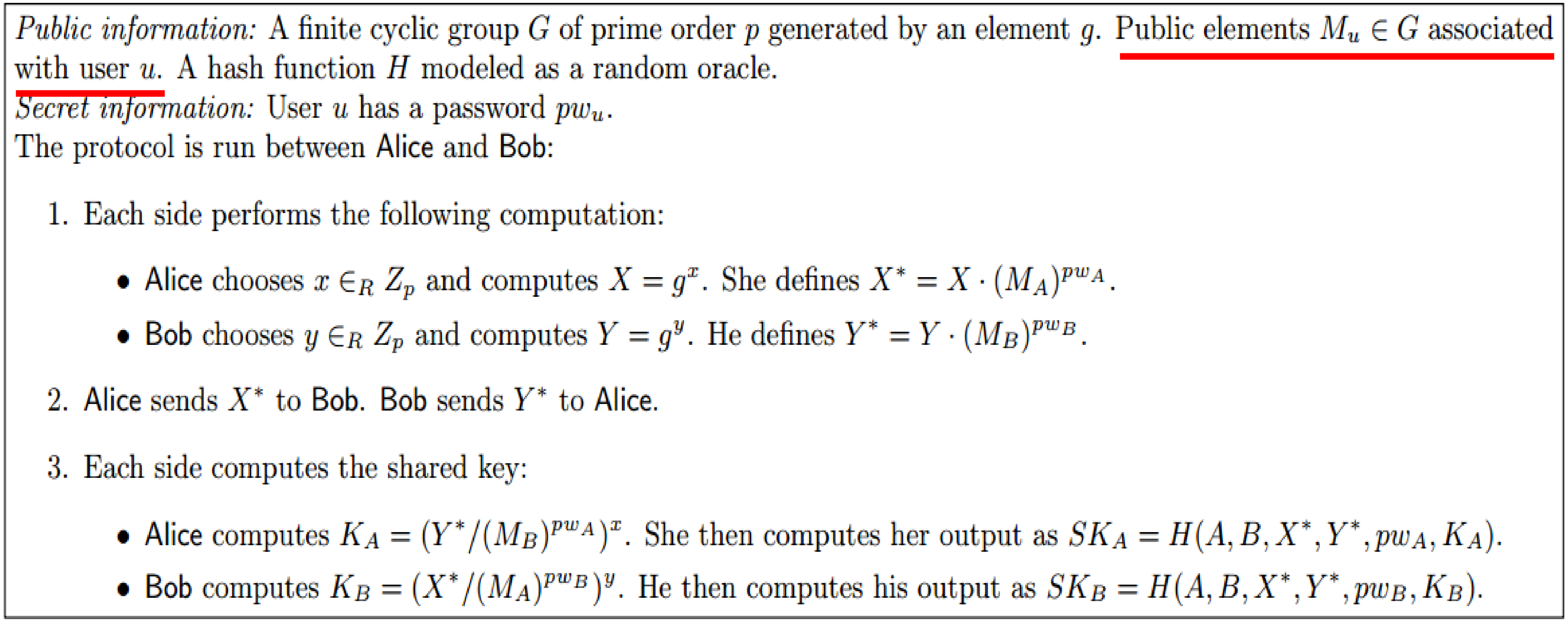
Preliminaries

- **Password authenticated key exchange** (PAKE), which allows two parties to agree on a session key iff they share a short secret (i.e., password).
- **Short hash**, which is high collision rate and short output.
- **Additively homomorphic encryption**, *Dec(sk, Enc(pk, a)+Enc(pk, b)) = a+b*.
- **Randomized threshold**, for each file, the server set a random threshold ***t*** to decide whether dedup or not.
- **Rate limiting**, limiting the number of PAKE runs for each file on both the uploader and the checkers, so as to prevent from the online brute-force attack by a compromised active server.

Threat Model

- Possible attacks:
  - **Online brute-force attack** by a compromised active uploader;
    - To try if some content has been uploaded before by uploading an interest file.
  - **Offline brute-force attack** by a compromised passive *S*;
    - To try if some ciphertext is a known content by using a dictionary attack.
  - **Online brute-force attack** by a compromised active *S*.
    - The *S* masquerades the client to guess the existence.

PAKE Construction



Public information: A finite cyclic group G of prime order p generated by an element g. Public elements $M_u \in G$ associated with user u. A hash function H modeled as a random oracle.
Secret information: User u has a password pw_u.
The protocol is run between Alice and Bob:

1. Each side performs the following computation:
   - Alice chooses $x \in_R Z_p$ and computes $X = g^x$. She defines $X^* = X \cdot (M_A)^{pw_A}$.
   - Bob chooses $y \in_R Z_p$ and computes $Y = g^y$. He defines $Y^* = Y \cdot (M_B)^{pw_B}$.
2. Alice sends X^* to Bob. Bob sends Y^* to Alice.
3. Each side computes the shared key:
   - Alice computes $K_A = (Y^*/(M_B)^{pw_A})^x$. She then computes her output as $SK_A = H(A, B, X^*, Y^*, pw_A, K_A)$.
   - Bob computes $K_B = (X^*/(M_A)^{pw_B})^y$. He then computes his output as $SK_B = H(A, B, X^*, Y^*, pw_B, K_B)$.

- $K_A == K_B$, iff $pw_A == pw_B$.

System Model

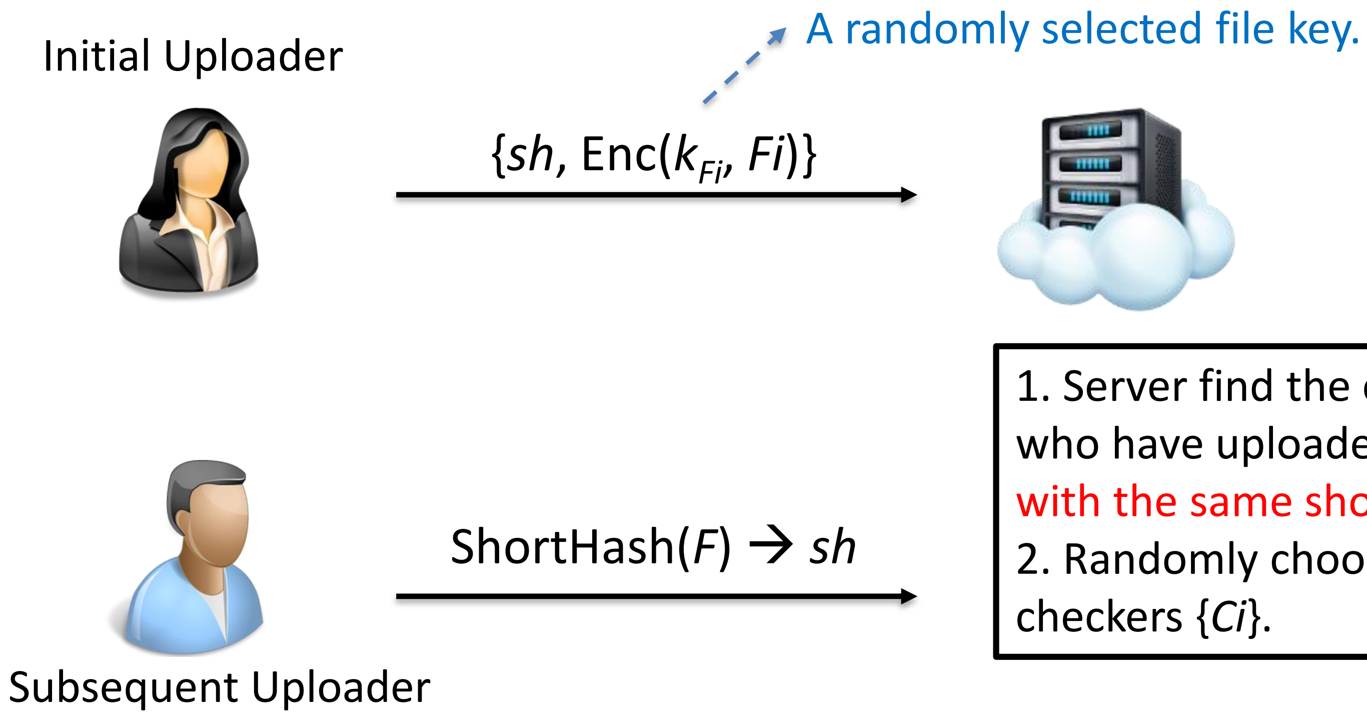
- General setting:
  - Storage server;
  - Multiple clients.
    - Clients never communicate directly, but exchange messages with the server who processes the messages and/or forwards them as needed.
- Participants in a deduplication system:
  - Storage server;
  - Uploader, who attempts to upload a file;
  - Existing clients {*Ci*}, who have already uploaded a file.

No additional key server!

Naïve Construction

- A naïve construction utilizing the PAKE protocol:
  - After running the PAKE protocol, *Ci* derives a key *ki* and the uploader *C* derives a key *k*.
  - *Ci* sends *E(ki,fk)* to *C*, where the *C* can decrypt it iff they share the same secret.
- Privacy leakage:
  - The uploader *C* is able to figure out what files are stored in cloud.
    - By checking whether the file has been uploaded before.
- A better construction requires that:
  - The uploader *C* cannot know whether the file has been uploaded before.

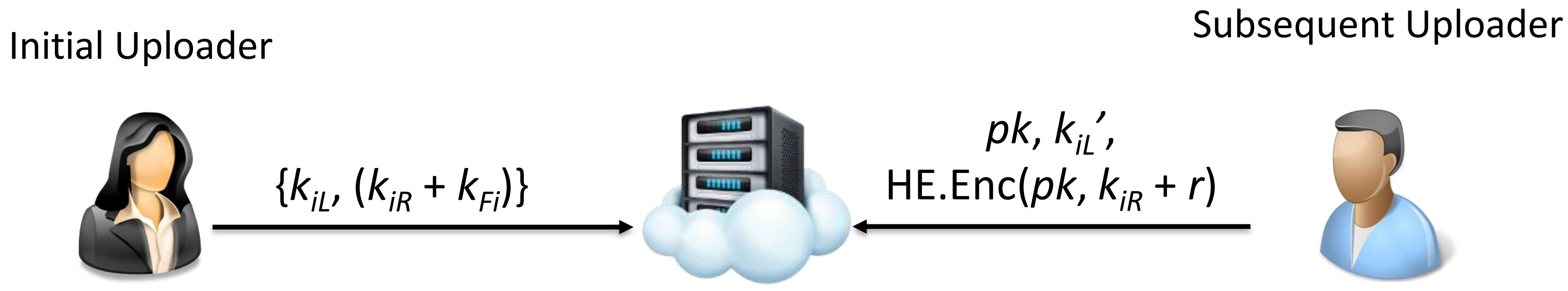
Proposed Construction (Server-side Dedup)


A randomly selected file key.
Initial Uploader
$\{sh, \text{Enc}(k_{Fi}, Fi)\}$
ShortHash(F) → sh
Subsequent Uploader

1. Server find the checkers who have uploaded files with the same short hash *sh*.
2. Randomly choose a set of checkers {*Ci*}.

3. The subsequent uploader runs PAKE protocol with selected clients. (Note that the messages are relayed by the server.)
4. After the invocation of the PAKE protocol, each *Ci* gets a session key *ki*, while *C* gets a set of session keys {*ki'*}.

Proposed Construction (Server-side Dedup)



ki split into $k_{iL}||k_{iR}$
$k_{iR} + k_{Fi}$

ki' split into $k_{iL}'||k_{iR}'$
HE.Enc($pk, k_{iR}' + r$), where r is a random element chosen by *C*

Proposed Construction (Server-side Dedup)

Subsequent Uploader

k_{1L},	$k_{1R} + k_{F1}$	k_{1L}',	HE.Enc$(pk, k_{1R}' + r)$
k_{2L},	$k_{2R} + k_{F2}$	k_{2L}',	HE.Enc$(pk, k_{2R}' + r)$
….		….	
k_{nL},	$k_{nR} + k_{Fn}$	k_{nL}',	HE.Enc$(pk, k_{nR}' + r)$

1. The server checks if there is an index such that $k_{jL} = k_{jL}'$
2. If so, the server computes

$$e = \text{HE.Enc}(pk, k_{jR} + k_{Fj}) * \text{HE.Enc}(pk, (-1)(k_{jR}' + r))$$
$$= \text{HE.Enc}(pk, k_{Fi} - r)$$

$$\xrightarrow{\quad e \quad}$$

1. k_F = HE.Dec$(sk, e) + r$

$$\xleftarrow{\quad \text{Enc}(k_F, F) \quad}$$

2. Enc(k_F, F)

Rate Limiting

- A compromised active server can apply online brute-force attacks against *C* or *Ci* by pretending to be an uploader.

- A ***per-file rate limiting strategy*** can both improve security and reduce overhead (namely the number of PAKE runs) without damaging the deduplication effectiveness.

Support Secure Client-side Dedup

- Their above design is for server-side deduplication.
- To save bandwidth, we transform it to support **client-side dedup**.
  - However, online brute-force attacks by compromised uploader should be considered.
    - For a predictable file, an uploader can construct all candidate files, upload them and observe which one causes deduplication.
- Using randomized threshold approach: Recall [HPS, *IEEE Security & Privacy'10*]
  - For each file, the server set a random threshold t_F *(>=2)* and a counter *c* that indicates the number of *Cs* that have previously uploaded this file;
  - In the case of a match between the messages submitted by the uploader and the checkers, if $c < t$, the server tells the uploader to upload the encrypted file. Otherwise, the server informs the uploader that there is duplicate and the upload is saved.
  - In the case of a no match, the server asks the uploader to upload the encrypted file.

Security Analysis

- Due to the high collision of short hash, the server is not able to guess the file reliably.
  - Offline brute-force attacks by compromised server are prevented.
- In the setting of client-side dedup, the random threshold method is adopted
  - Online brute-force attacks by compromised uploader is prevented.
- Per-file rate limiting is enforced. Both the uploaders and the checkers should limit the number of PAKE run for each file in their respective roles.
  - Online brute-force attacks by compromised active server is prevented.

Conclusion

- They designed a PAKE-based protocol that enables two parties to privately compare secrets and share the encryption key.

- Based on this protocol, they developed the **first** secure cross-user deduplication scheme that supports client-side encryption without requiring any additional independent servers.
  - Recall that DupLESS has such additional key server.

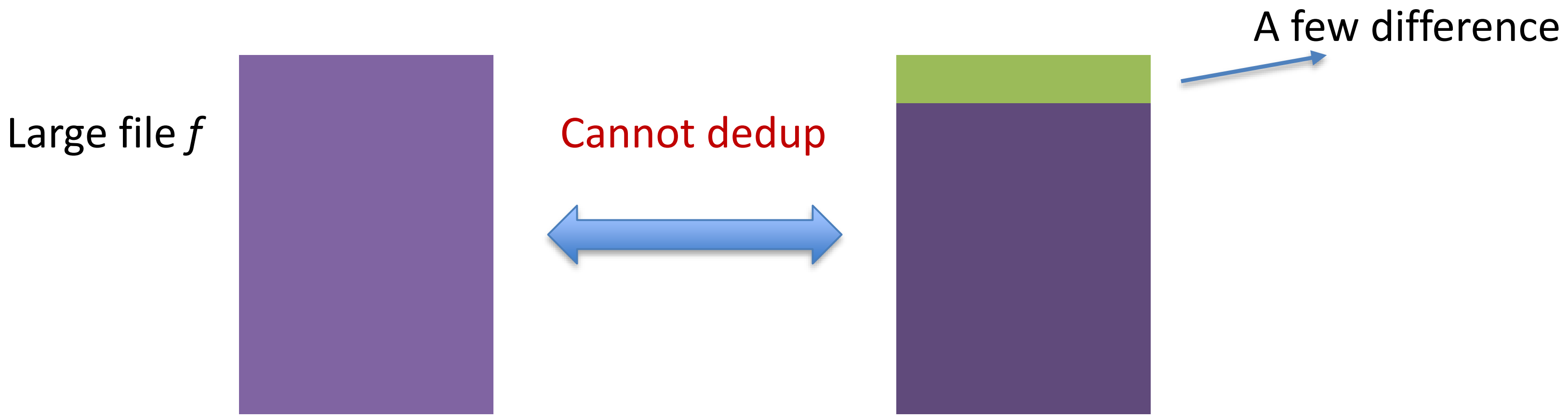
BL-MLE: Block-Level Message-Locked Encryption for Secure Large File Deduplication

Rongmao Chen, Yi Mu, Guomin Yang, and Fuchun Guo

In ***IEEE TIFS***, *2015*

Target Problem

- Although an MLE scheme can be extended to obtain secure deduplication for large files, it requires a lot of metadata maintained by the end user and the cloud server.
- For large file deduplication:
  - Normally **block-level deduplication** can provide more space savings than file-level deduplication does in large file storage.



Dual-Level Source-Based (DLSB) Deduplication

- The user firstly sends a file identifier to the server for file redundancy checking.
  - To perform **file-level deduplication** firstly.
- If the file to-be-stored is duplicated in the server, the user should convince the server that he/she indeed owns the file by performing a PoW protocol.
  - PoW should be implemented along with source-based deduplication.
- Otherwise, the user uploads the identifiers/tag of all the file blocks to the server for **block-level deduplication** checking.
- Finally, the user uploads data blocks which are not stored in the server.

Using MLE for Block-Level Deduplication

- **Metadata I (Block identifiers):**
  - In addition to the file identifier, the server also has to store a large number of block identifiers for redundancy checking.
  - Stored in the **primary memory** for fast access upon upload request.
- **Metadata II (Block keys):**
  - Each file block should be encrypted using a block key which is derived from the data block itself.
  - Using a master key to encrypt them? --> extra space cost
- **Metadata III (PoW tags):**
  - As the actual data blocks are stored in the secondary storage, it is more practical to use **some short PoW tags** to perform the verification.

Reduce Metadata?

- The metadata size: (n is the number of blocks)
  - *O*(|file tag|, n*|block tag|, n * |block key|, |PoW tag|).
- Among all the metadata, the **block identifiers** and the **encrypted block keys** form the major storage overhead.
  - Linear to the number of blocks.
  - This motivated us to design a new scheme that can **combine the block identifier and the encrypted block key into one single element**.

Block-Level Message-Locked Encryption (BL-MLE)

- Using **Symmetric Bilinear Map** as fundamental primitive.
- Consists of the following algorithms:
  - Setup
  - KeyGen (M-KeyGen, B-KeyGen)
  - Enc
  - Dec
  - TagGen (M-TagGen, B-TagGen)
  - ConTest $\dashrightarrow$ to prevent duplicate faking attack
  - EqTest
  - B-KeyRet $\dashrightarrow$ For block key retrieval as the block tag also serves as an encrypted block key.
  - PoWPrf
  - PoWVer

$\left.\begin{array}{l}\text{PoWPrf}\\ \text{PoWVer}\end{array}\right\}$ for PoW protocol

Setup(1^λ)

- Generates a prime p, the descriptions of two groups G, G_T of order p, a generator g of G and a bilinear map e: G × G → G_T .
- Choose an integer s ∈ N and three hash function:
  - H1 : {0, 1}* → Zp,
  - H2 : $\{Zp\}^s$ → G,
  - H3 : G → $\{Zp\}^s$.
- Pick s elements randomly $u_1, u_2, \ldots, u_s \leftarrow_R$ G.
- The system parameters are:
  - P = < p, g, G, GT, e, H1, H2, H3, s, $u_1, u_2, \ldots, u_s$ >.

KeyGen(M)

- M = M[1]||…||M[n].
  - The message will be separated into multiple blocks.
- M-KeyGen(M): *kmas* = H1(M)
  - The master key is based on the entire message.
- B-KeyGen(M[i]): *ki* = H2(M[i])
  - The block key is derived from each block.

- H1(): $\{0, 1\}^* \rightarrow Zp$
- H2(): $\{Zp\}^s \rightarrow G$

Enc(*ki*, M[i]) & Dec(*ki*, C[i])

- C[i] = H3(*ki*) XOR M[i] $\dashrightarrow$ Encryption
- M[i] = H3(*ki*) XOR C[i] $\dashrightarrow$ Decryption
  - If ki = H2(M[i]), accept the result;
  - Otherwise, output NULL.
- H3(): G → {Zp}^{s}

TagGen(M)

- $M = M[1]||\ldots||M[n]$.
- M-TagGen(M):
  - $kmas$ = M-KeyGen(M)
  - $T0 = g^{kmas}$

File tag

- B-TagGen(M, i):
  - ki = B-KeyGen(M[i])
  - $C[i] = Enc(ki, M[i])$
  - Split C[i] into s sectors: $Ti = (ki \cdot \prod_{j=1}^{s} u_j^{C[i][j]})^{kmas}$
  - $AUXi = e(ki, T0)$
- $u_1, u_2, \ldots, u_s \leftarrow_R G$

Block tag

To shorten the size of the block tag, the block ciphertext is split into *s* sectors. Each sector is one element of Zp, and hence the size of a block tag is just 1/s of the corresponding block ciphertext.

Attached to the block tag *Ti*, which is used for the consistency checking, and no need to stored.

ConTest(*Ti*, C[i])

- They require that the block tag construction should achieve strong tag consistency.
- Given C[i] and block tag *Ti* with auxiliary data *AUXi*:
  - Split C[i] into *s* sectors: C[i][1], …, C[i][s]
  - Check $e(Ti, g) \mathrel{?=} AUXi \cdot e(\prod_{j=1}^{s} u_j^{C[i][j]}, T0)$
  - Output 1 or 0.

  - $e((ki \cdot \prod_{j=1}^{s} u_j^{C[i][j]})^{kmas}, g) \mathrel{?=} e(ki, g^{kmas}) \cdot e(\prod_{j=1}^{s} u_j^{C[i][j]}, g^{kmas})$
- Once the server has checked that C[i] and *Ti* are consistent, *AUXi* can be discarded.

EqTest(*Ti*, *Ti'*, *T0*, *T0'*)

- Check e(*Ti*, *T0'*) ?= e(*Ti'*, *T0*).
- Output 1 or 0.

- $e((ki \cdot \prod_{j=1}^{s} u_j^{C[i][j]})^{kmas}, g^{kmas'})$? = $e((ki' \cdot \prod_{j=1}^{s} u_j^{C[i][j]})^{kmas'}, g^{kmas})$
- *kmas* != *kmas'*, due to H1(M) may not equal H1(M'), but if the blocks are identical, this equation can still be satisfied.

Key idea!

B-KeyRet(*kmas*, *Ti*, C[i])

- Split C[i] into s sectors: C[i][1], …, C[i][s]
- $ki = Ti^{kmas} \cdot (\prod_{j=1}^{s} u_j^{C[i][j]})^{kmas})^{-1}$

- $ki = (ki \cdot \prod_{j=1}^{s} u_j^{C[i][j]})^{kmas} \cdot (\prod_{j=1}^{s} u_j^{C[i][j]})^{kmas})^{-1}$

Block key

Before decrypt the block ciphertext, the user should retrieve the key first.

PoW in Brief

1. The server generates a challenge query $Q = \{(i, v_i)\}$:
   - The "i" is the position of a queried block, $v_i \leftarrow_R Zp$.
2. The client compute Ti as a proof based on specific C[i]:
   - $P_T = \prod_{(i,\ vi)} Ti^{vi} \leftarrow \mathrm{PoWPrf}(M, Q)$.
3. The server computes $V_T = \prod_{(i,\ vi)} Ti^{vi} \leftarrow \mathrm{PoWVer}(P_T, \{Ti\}_{i<i<n}, Q)$.
   - Check P_T ?= V_T.
   - Output 1 or 0.

Performance

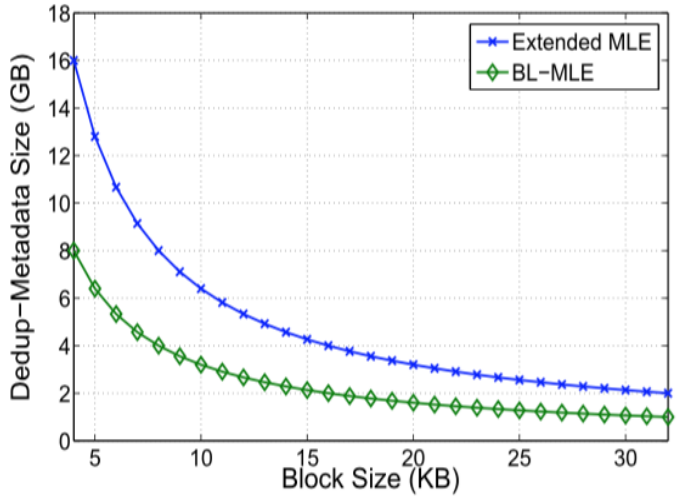


Fig. 1. Impact of Block Size on Dedup-Metadata Size ($f = 1$).

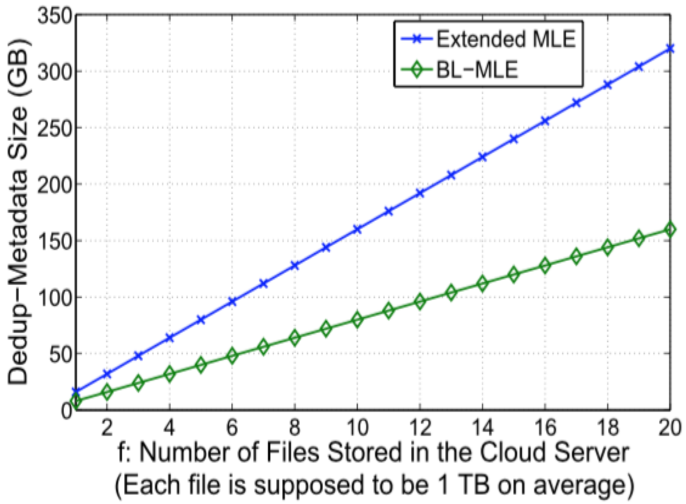


Fig. 2. Impact of f on Dedup-Metadata Size (Block size: 4 KB).

Conclusion

- BL-MLE is a newly developed cryptographic primitive for dual-level source-based deduplication of large files to achieve space-efficient storage.
    - **To avoid extra storage for the block keys** at either the cloud side or the user side, BL-MLE encapsulates the block keys into the block tags, and later the party holding a master key is able to recover the block keys from the block tags.
    - Therefore, to store a file, the metadata of each block ciphertext for supporting block-level deduplication is just one block tag.

To Learn More

- John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. “Reclaiming Space from Duplicate Files in a Serverless Distributed File System.” In *Proc. of* ***IEEE ICDCS***, 2002.
- Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. “Side channels in cloud services: Deduplication in cloud storage.” In ***IEEE Security & Privacy***, 2010, Nov. 8(6):40-7.
- Shai Halevi, Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. “Proofs of Ownership in Remote Storage Systems.” In *Proc. of* ***ACM CCS***, 2011.
- Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. “Message-locked encryption and secure deduplication.” In *Proc. of* ***EUROCRYPT***, 2013.
- Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. “DupLESS: Server-Aided Encryption for Deduplicated Storage.” In *Proc. of* ***USENIX Security***, 2013.
- Jia Xu, Ee-Chien Chang, and Jianying Zhou. “Weak leakage-resilient client-side deduplication of encrypted data in cloud storage.” In *Proc. of* ***ACM ASIACCS***, 2013.
- Jan Stanek, Alessandro Sorniotti, Elli Androulaki, and Lukas Kencl. “A Secure Data Deduplication Scheme for Cloud Storage.” In *Proc. of* ***FC***, 2014.
- Jian Liu, N. Asokan, and Benny Pinkas. “Secure Deduplication of Encrypted Data without Additional Independent Servers.” In *Proc. of* ***ACM CCS***, 2015.
- Rongmao Chen, Yi Mu, Guomin Yang, and Fuchun Guo. “BL-MLE: Block-Level Message-Locked Encryption for Secure Large File Deduplication.” In ***IEEE TIFS***, 2015.
- Yifeng Zheng, Xingliang Yuan, Xinyu Wang, Jinghua Jiang, Cong Wang, and Xiaolin Gui. “Enabling Encrypted Cloud Media Center with Secure Deduplication.” In *Proc. of* ***ACM AsiaCCS***, 2015.

To Learn Even More

- Tao Jiang, Xiaofeng Chen, Qianhong Wu, Jianfeng Ma, Willy Susilo, and Wenjing Lou. “Secure and Efficient Cloud Data Deduplication With Randomized Tag”, In ***IEEE TIFS***, 2017.
- Mihir Bellare, Sriram Keelveedhi, “Interactive message-locked encryption and secure deduplication”, In *Proc. of* ***PKC***, 2015.
- Frederik Armknecht, Jens-Matthias Bohli, Ghassan O. Karame, and Franck Youssef. “Transparent Data Deduplication in the Cloud.” In *Proc. of* ***ACM CCS***, 2015.
- Jin Li, Xiaofeng Chen, Mingqiang Li, Jingwei Li, Patrick PC Lee, and Wenjing Lou. “Secure deduplication with efficient and reliable convergent key management.” ***IEEE Trans. on Parallel and Distributed Systems (TPDS)***, 2014.
- Jin Li, Yan Kit Li, Xiaofeng Chen, Patrick PC Lee, and Wenjing Lou. “A hybrid cloud approach for secure authorized deduplication.” ***IEEE Trans. on Parallel and Distributed Systems (TPDS)***, 2015.
- Yukun Zhou, Dan Feng, Wen Xia, Min Fu, Fangting Huang, Yucheng Zhang, and Chunguang Li, “SecDep: A User-Aware Efficient Fine-Grained Secure Deduplication Scheme with Multi-Level Key Management.” In *Proc.* ***IEEE MSST***, 2015.
- Jin Li, Xiaofeng Chen, Xinyi Huang, Shaohua Tang, Yang Xiang, Mohammad Mehedi Hassan, and Abdulhameed Alelaiwi, “Secure Distributed Deduplication Systems with Improved Reliability.” ***IEEE Trans. on Computers (TC)***, 2015.
- …

Thanks.