ICT Risk Assessment

Fabrizio Baiardi
f.baiardi@unipi.it

Syllabus

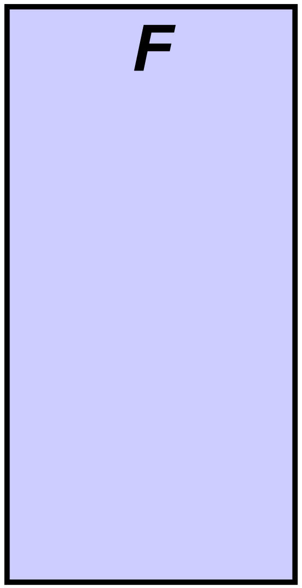
- Security
  - New Threat Model
  - New Attacks
  - Countermeasures

← Service provider attack to data

Attacks against Cloud Services

•Not only Software as a Service but also "Storage-as-a-service" becoming a more common business model where a client pays server to store file *F*

- Without retrieving file, how a client can be sure that server still has it?
- Or, more generally, can provide it within an agreed response time?
- *Archiving* is a typical case: Client retains only metadata



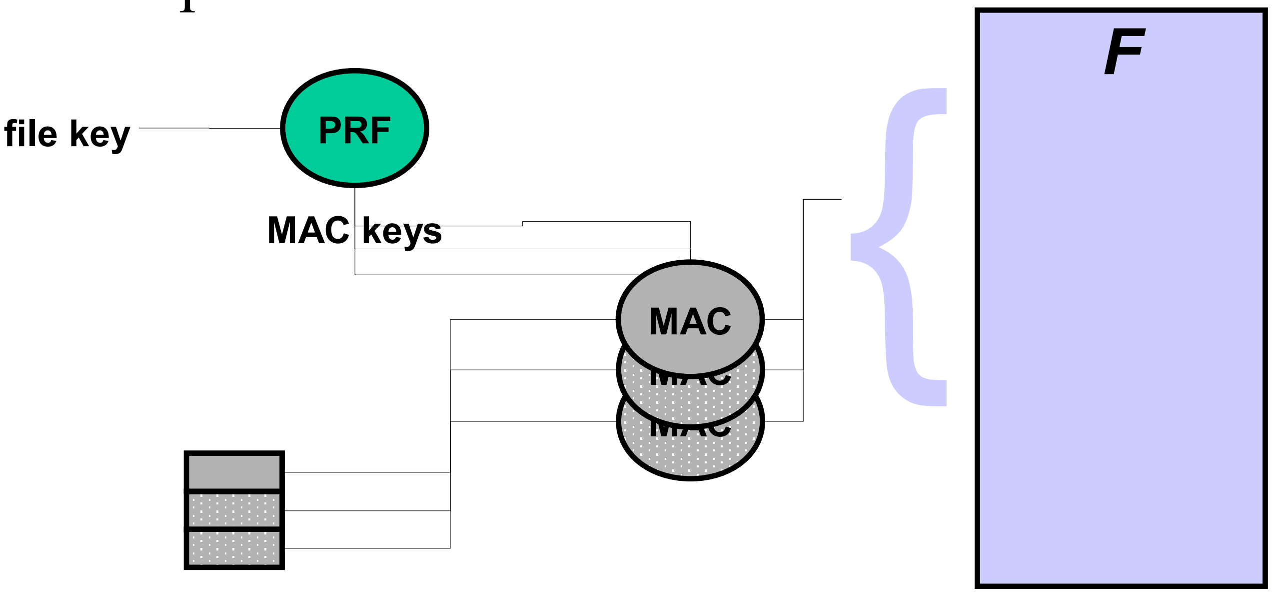
Proofs of Possess - Retrievability

- A *proof of Posses - Retrievability* (POP, POR) provides some assurance that a party possesses a file, without actually retrieving it
- Objective: Provide “early warning” of deletion, corruption, or other failure to meet service levels, in time to recovery e.g., exclude this server and add another one (checking SLA)
- POR shows that at time of test, adversary’s state is sufficient (with high probability) to enable retrieval – thereby limiting time period during which undetected corruption may occur
- Since adversary can distinguish POR (= modest number of queries) from actual retrieval (= large number), it can always pass test now then deny service at a later time

A Challenge-Response MACs

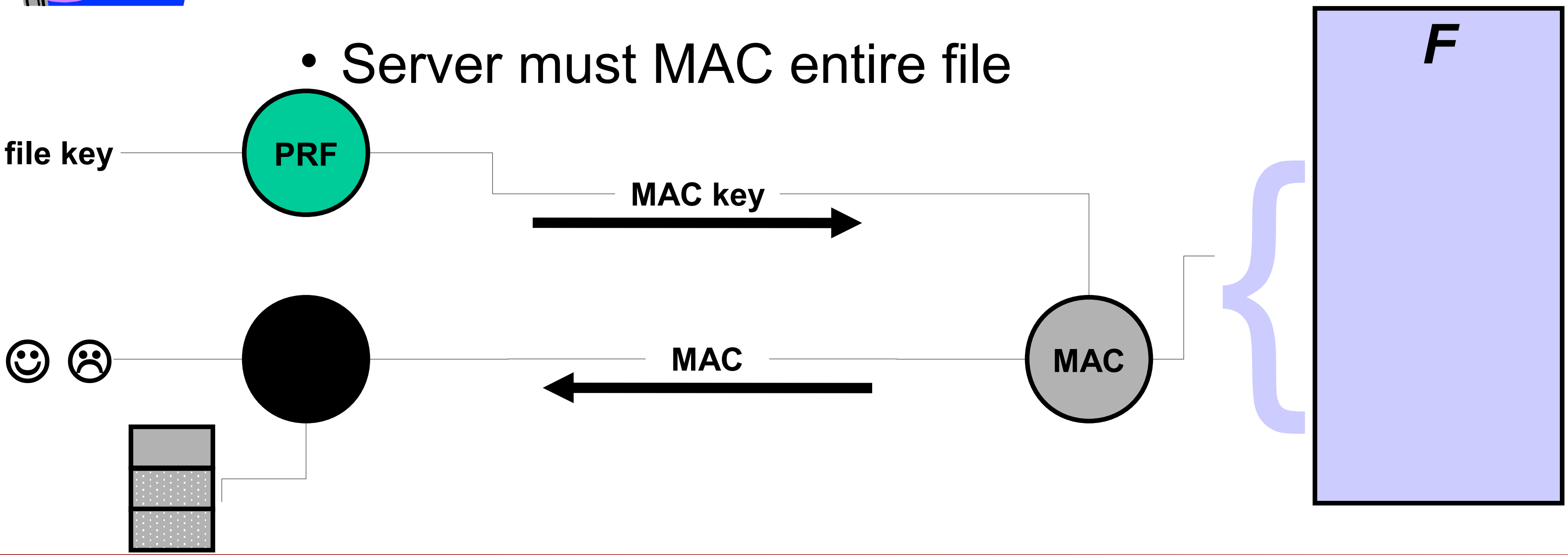
- MAC file with different keys, try one at a time

pseudo random function



Simple Approach, cont'd

- MAC file with different keys, try one at a time
- # runs limited by client storage
- Server must MAC entire file



Block approach

- The file is splitted into d blocks at upload
- We check whether some blocks is still there
- The probability of non detecting that some block have been erased (an eraser) is

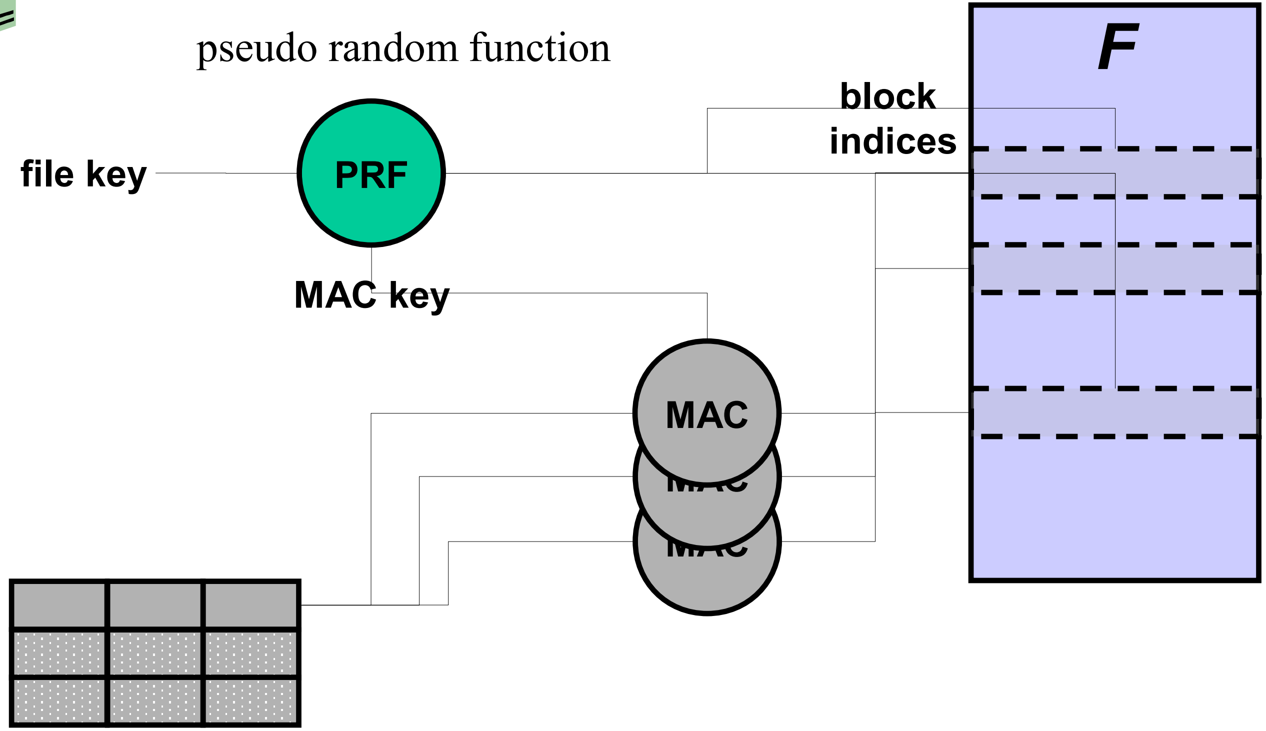
$$P_{esc} = \left(1 - \frac{m}{d}\right)^r$$

where

- r is the number of blocks we control
- m/d is the percentage of blocks that have been erased
- 1-m/d is the probability of selecting one block that has not been erased

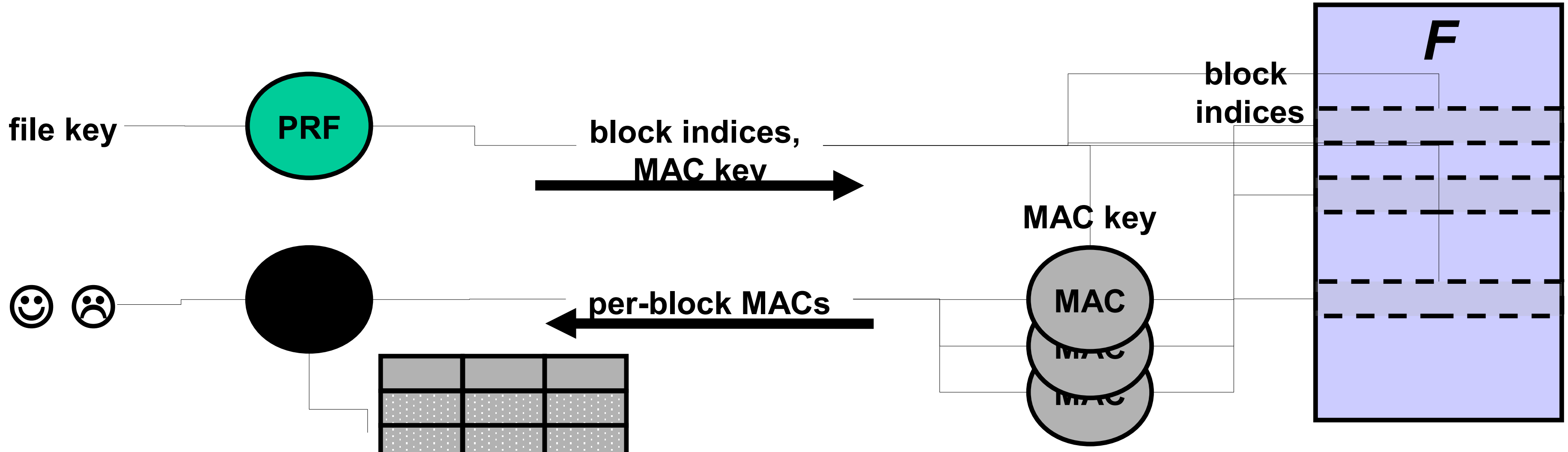
Per-Block MACs

- MAC selected blocks, and sample *q*



Per-Block MACs, cont'd

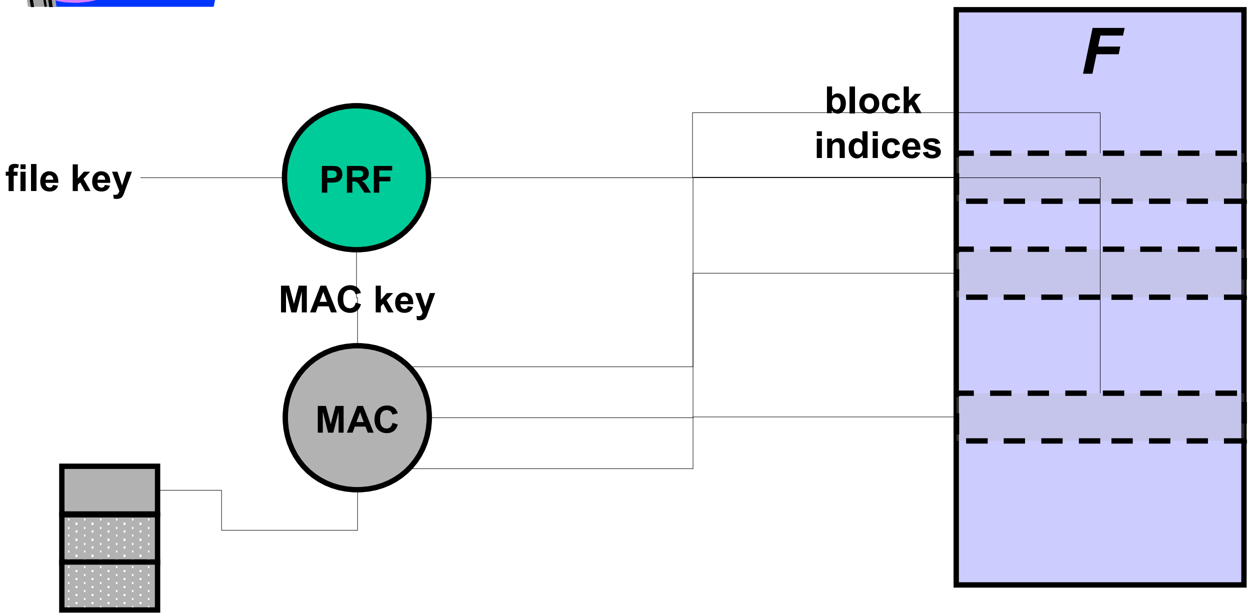
- MAC selected blocks, and sample q
- Server works now only q MACs / run
- But message exchange ~ q



- With error rate ε, Pr [undetected] $\leq (1 - \varepsilon)^q$

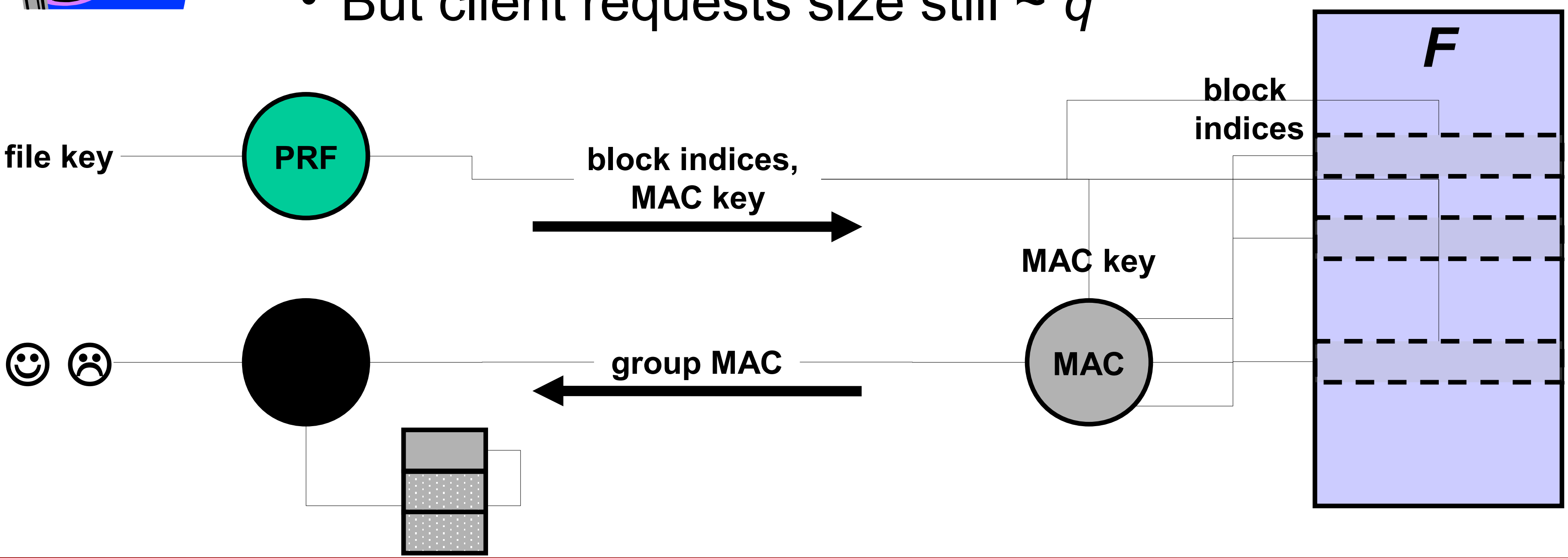
Group MACs

- MAC group of sampled blocks



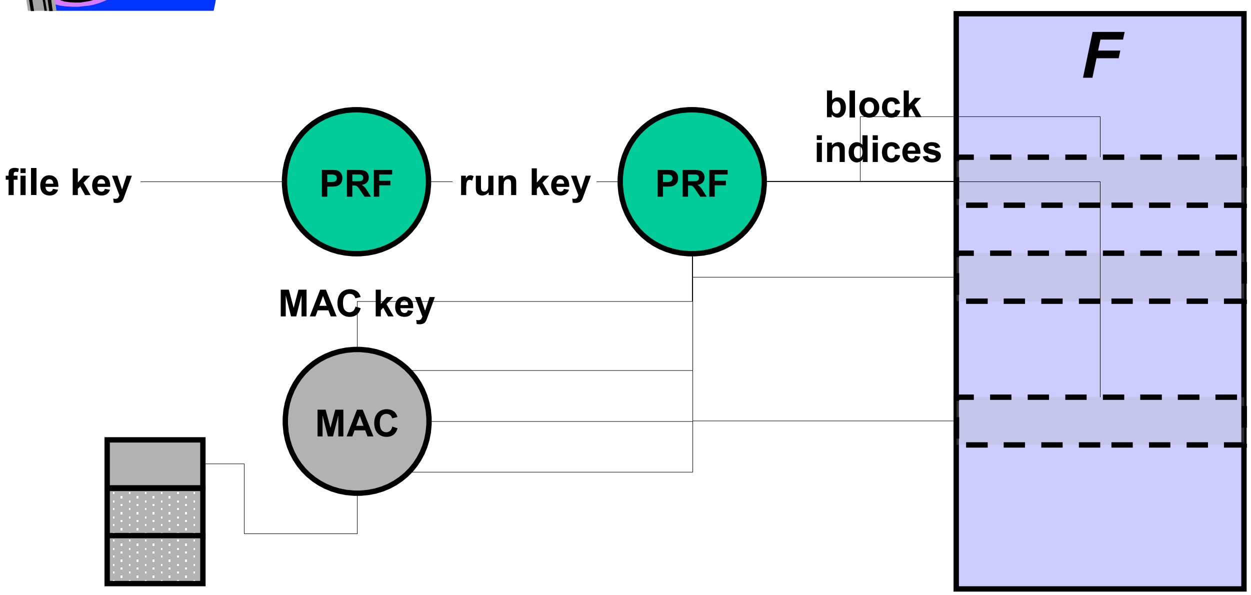
Group MACs, cont'd

- MAC group of sampled blocks
- Server response now constant size
- But client requests size still ~ q



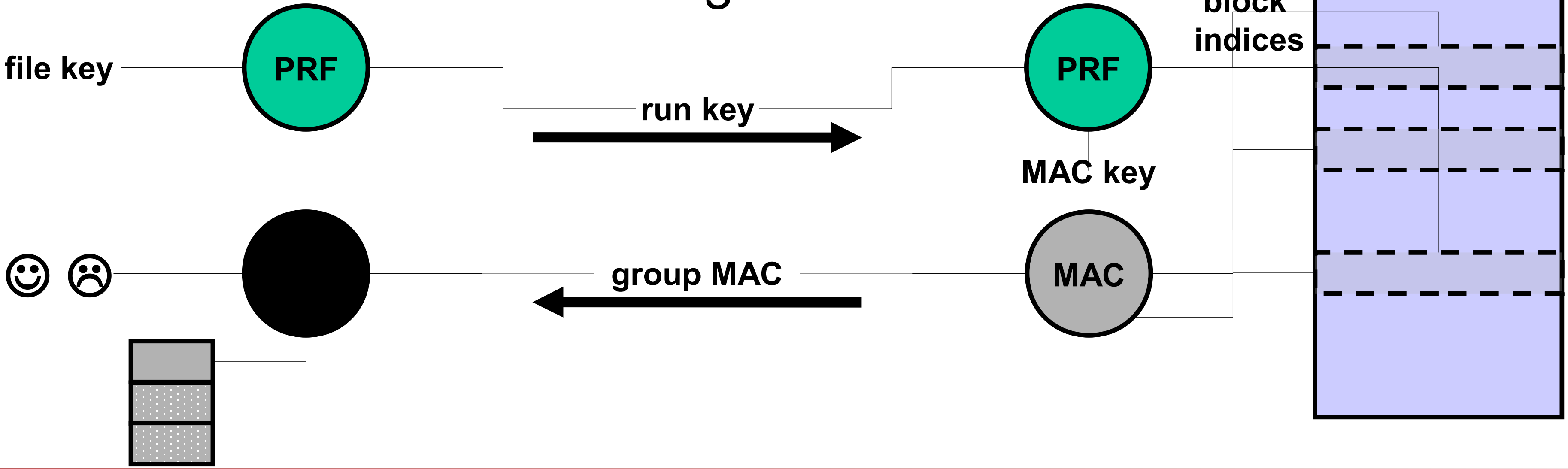
Index Derivation

- Derive block indices from *run key*



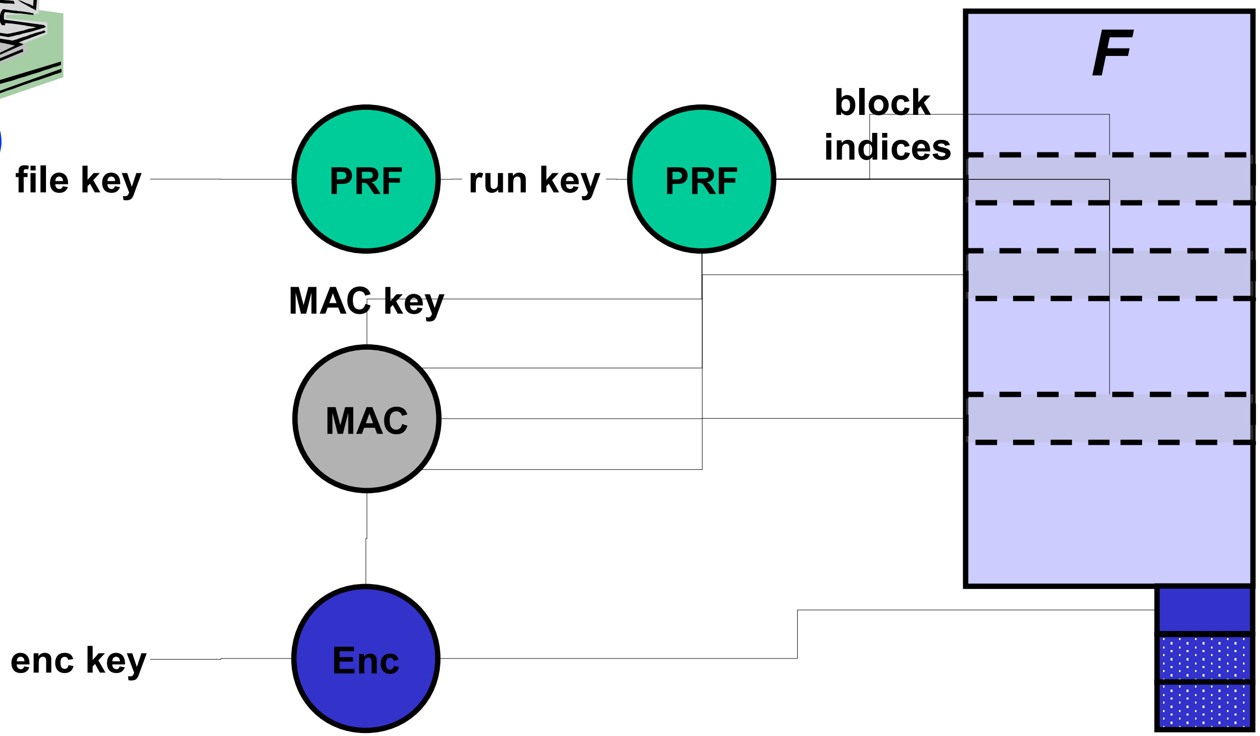
Index Derivation, cont'd

- Derive block indices from run key
- Both message exchanges now constant size
- But client storage still ~ # runs



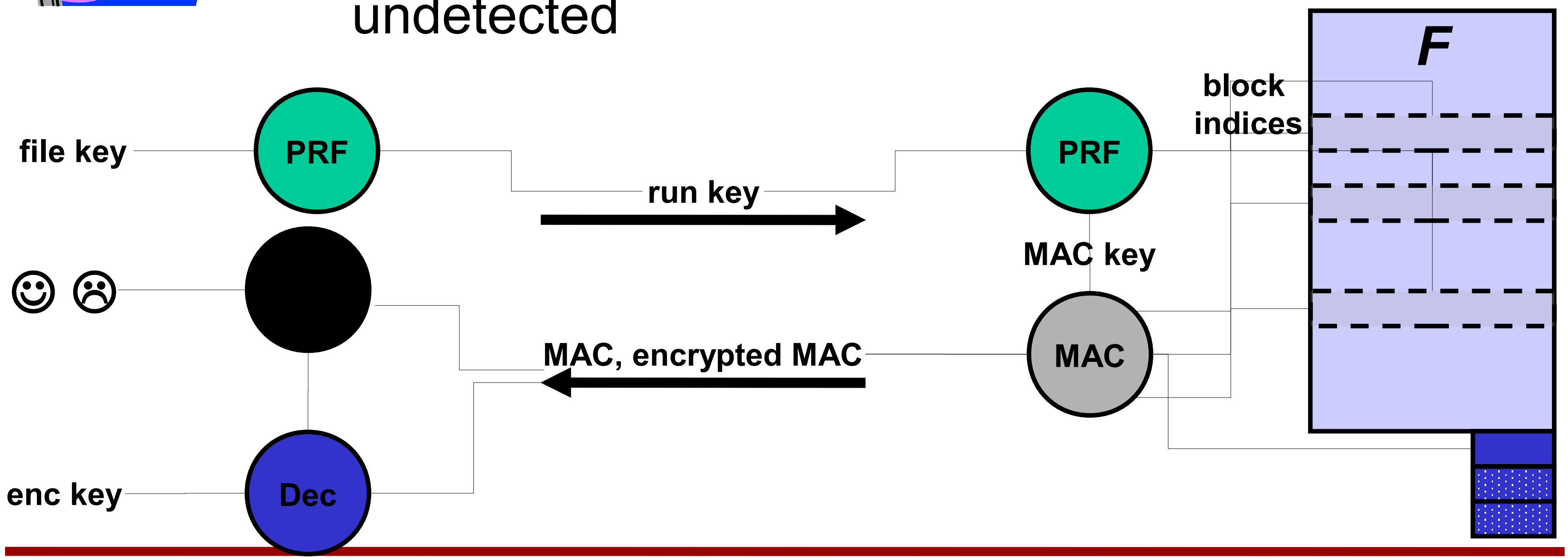
Server Storage of Encrypted MACs

- Encrypt group MACs, store on server



Server Storage of Encrypted MACs, cont'd

- Encrypt group MACs, store on server
- Client storage now constant
- But small error rate (< ε) may go undetected



Recovering from Errors

MAC sampling detects server error rate ≥ ε with high probability

Smaller error rate (< ε) may go undetected, but can be *corrected*

First solution: Apply error-correcting code to file before storing

But non-trivial: No efficient simple codes known that are robust against arbitrary adversarial errors

Second solution: Encrypt file, apply error-correcting code, then apply *pseudorandom permutation* to block order then compute error correction code

Remaining Challenges …

- There are schemes that support update of the file
- Other scheme based upon homomorphic encryption allow any one to check that the server stores the file
- Number of runs is limited by server storage of encrypted MACs but this is not very compelling

Homorphic encryption = Holy gray of encryption

Let *R* and *S* be sets and E an encryption $R \rightarrow S$ E is

- **Additively homomorphic if**

  E(a+b)=PLUS(E(a), E(b))

- **Multiplicatively homomorphic if**

  E(a×b)=MULT(E(a), E(b))

- **Mixed-multiplicatively homomorphic**

  $E(xy)$=Mixed-mult($E(x)$,y)

- **Fully homomorphic**

  no limitations on manipulations

Homomorphic encryption

- Data + Computation at the provider
- Inputs are encrypted by the client
- Outputs are transmitted to the client that decrypt it
- No trivial solution = the provider executes most computations to prevent cases where
  - the data is transmitted to the client,
  - the client decrypts the data, computes the results and encrypts
  - the results are transmitted to the provider

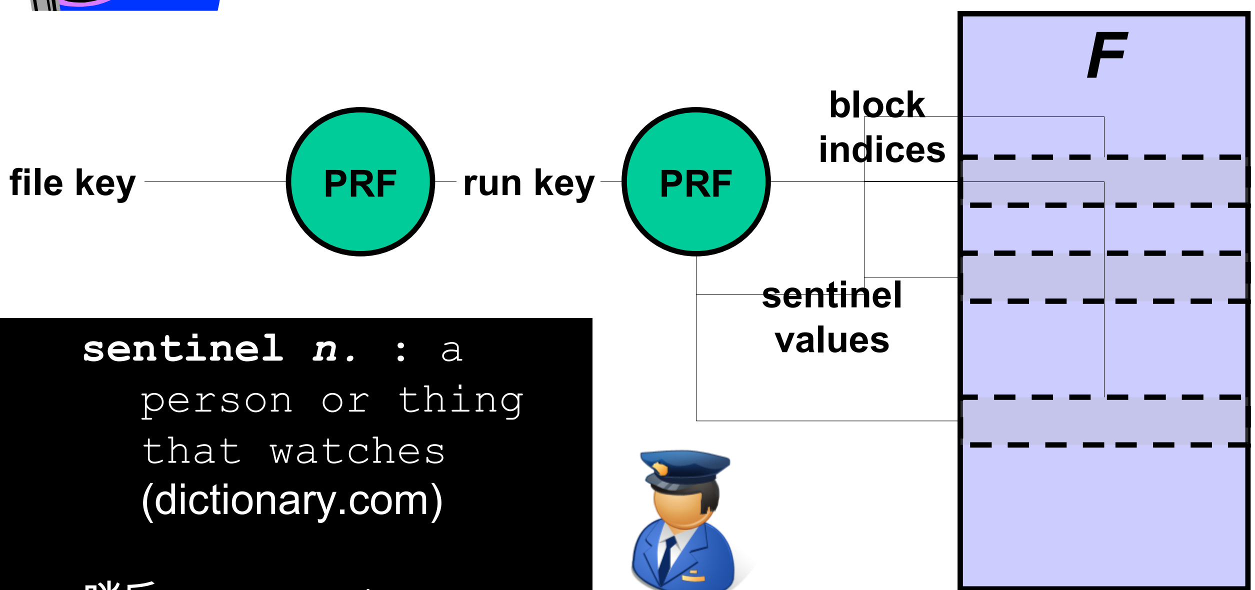
Sentinels – Another Approach

- Sentinels= randomly constructed check values.
- F' = F encryption + embedded sentinels,

  F is encrypted so that sentinels cannot be discovered
- *Verification phase:* V specifies the positions of some sentinels in F' and asks the archive to return the corresponding values.
- *Security:* Because F is encrypted and sentinels are randomly valued, the archive cannot feasibly distinguish *a priori* between sentinels and portions of the original file F.
  - If the provider deletes or modifies a substantial, fraction of F', high probability this also changes a fraction of sentinels.
  - If V requests and verifies enough sentinels, the V detects whether a substantial fraction of F' has been altered
- Individual sentinels are, however, only one-time verifiable.

Sentinel Overwriting

- Insert into selected blocks pseudorandom values, and check

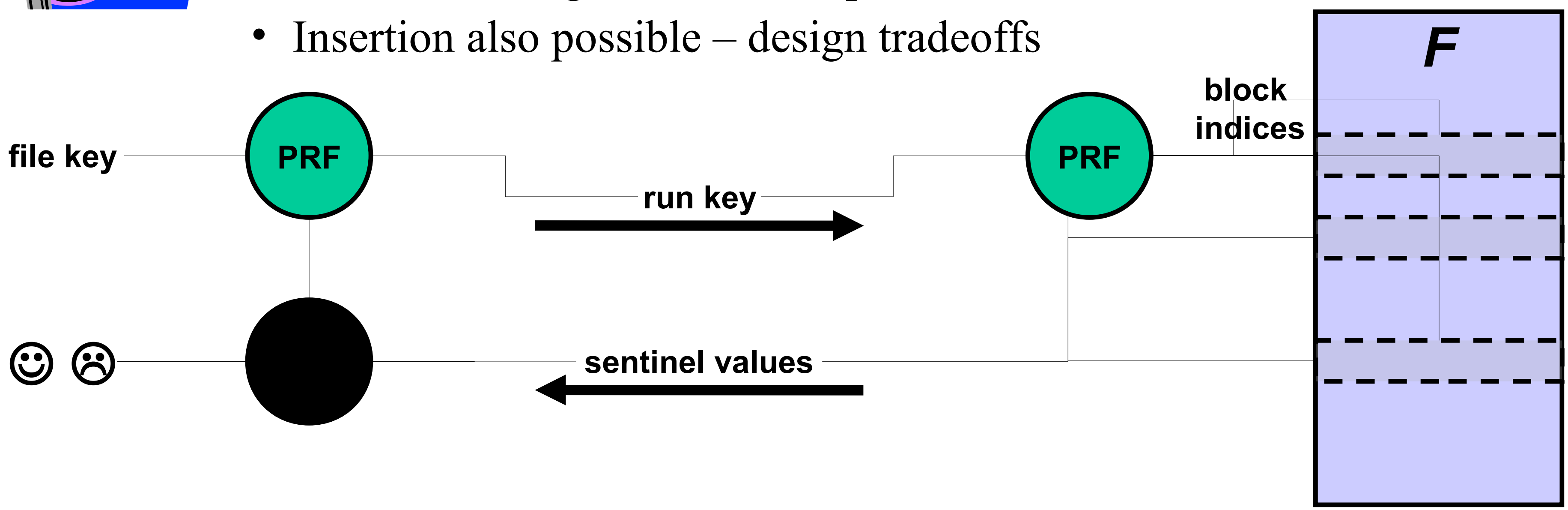


sentinel ***n.*** : a person or thing that watches (dictionary.com)

哨兵 : security guard, watchman, watcher (babylon.com)

Sentinel Overwriting, cont'd

- Insert pseudorandom values, and check
- Security proof in standard model
- Size limitations … but can optimize
- No special storage at server
- Error correcting code makes up for overwrite
- Insertion also possible – design tradeoffs



Theoretical Considerations

- Proof of retrievability is a protocol for demonstrating that a party possesses a file
  - Successful verification ←→ Successful retrieval
  - Party's "response" interface is preferred building block for reduction
- Different from *proof of knowledge*, which demonstrates that a party possesses a witness related to a public value
  - e.g., discrete log x of g^x
  - No corresponding public value for file
- In the sentinel POR scheme the sentinels and protocol messages are *independent* of the file whose possession is being proved

Conclusions

- Proofs of retrievability provide assurance that file stored on server can be retrieved – with only a modest number of operations and overhead
- Multiple design steps lead to practical schemes based on MACs, sentinels with many variants, optimizations to explore
- Next step: Integration with actual file systems for a real test of performance, parameterization

HAIL goals

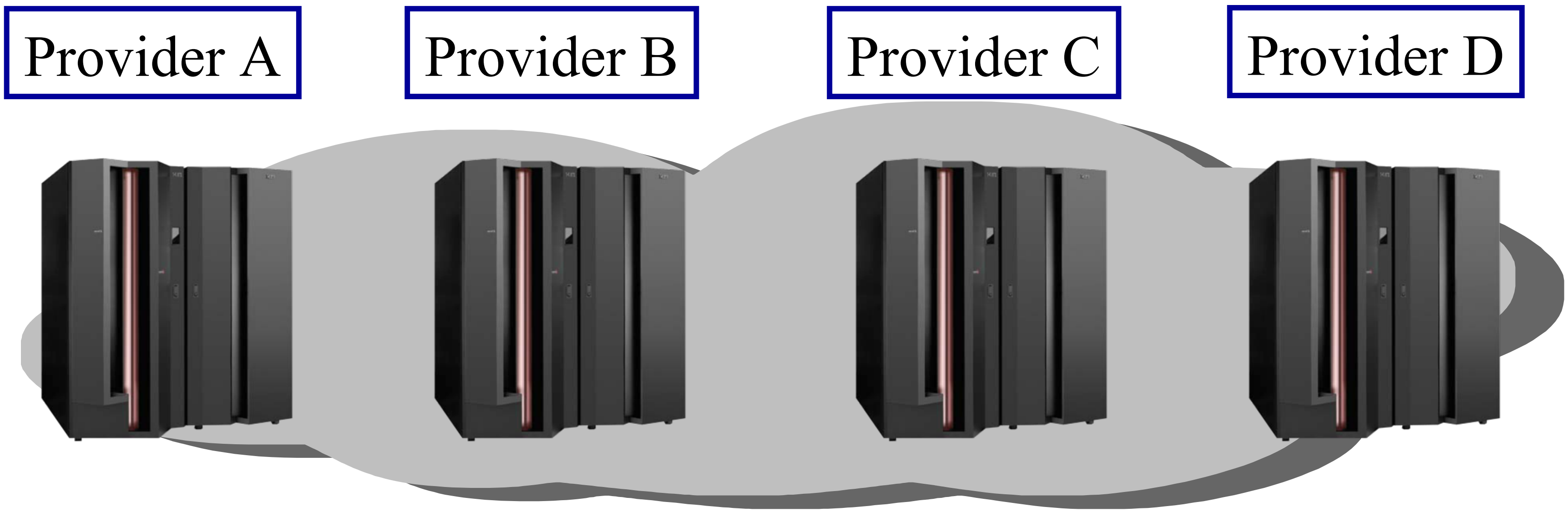
- Resilience against cloud provider failure and temporary unavailability
- A mobile adversary capable of progressively attacking storage providers and, in principle, ultimately corrupting all providers at different times.
- Use multiple cloud providers to construct a reliable cloud storage service out of unreliable components
- RAID (Reliable Array of Inexpensive Disks) for cloud storage under adversarial model
- Provide clients or third party auditing capabilities
- Efficient proofs of file availability by interacting with cloud providers
- Test-And-Redistribute the client uses PORs to detect file corruption and trigger reallocation of resources when needed
- On detecting a fault in a given server via challenge-response, the client recovers the corrupted shares from cross-server redundancy

RAID (Redundant Array of Inexpensive Disks)

Stripe	Data block	Data block	Data block	Parity block
	B_1	~~B_2~~	B_3	$P_1=B_1\oplus B_2\oplus B_3$
		$B_1\oplus B_3\oplus P_1$		

Shift from monolithic, high-performance drives to cheaper drives with redundancy

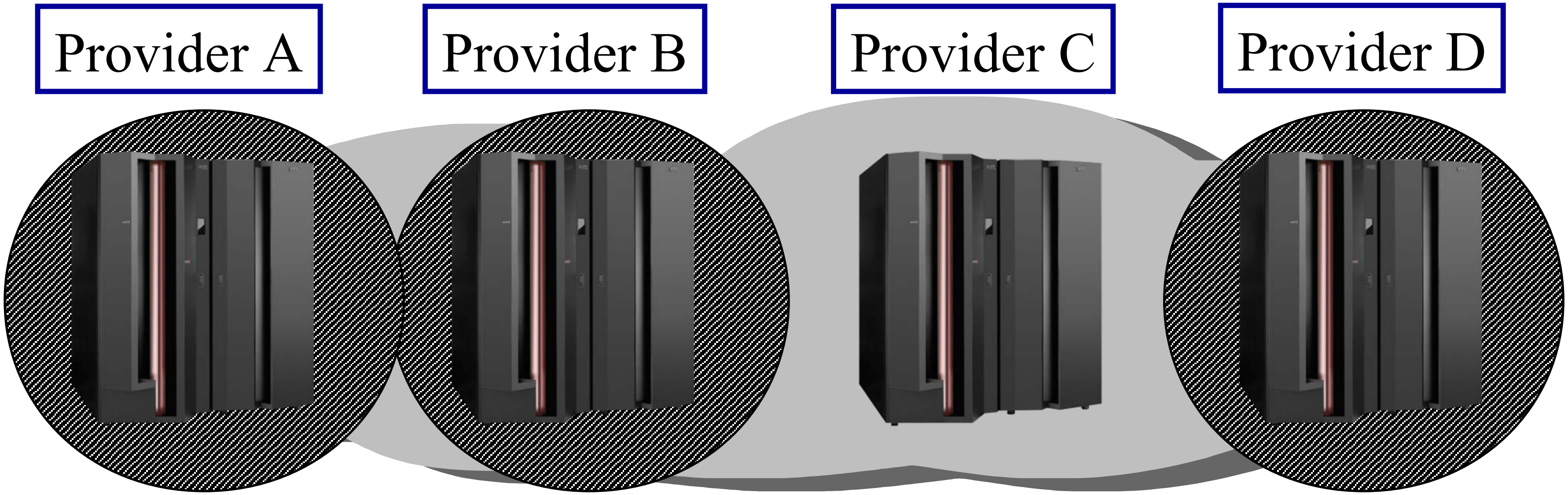
RAID in the Cloud



Fuse together cheap cloud providers to provide high-quality (reliable) abstraction

- E.g., Memopal offers $0.02 / GB / Month storage on a 5-year contract vs. Amazon at $0.15 / GB / Month

…But the cloud is adversarial!



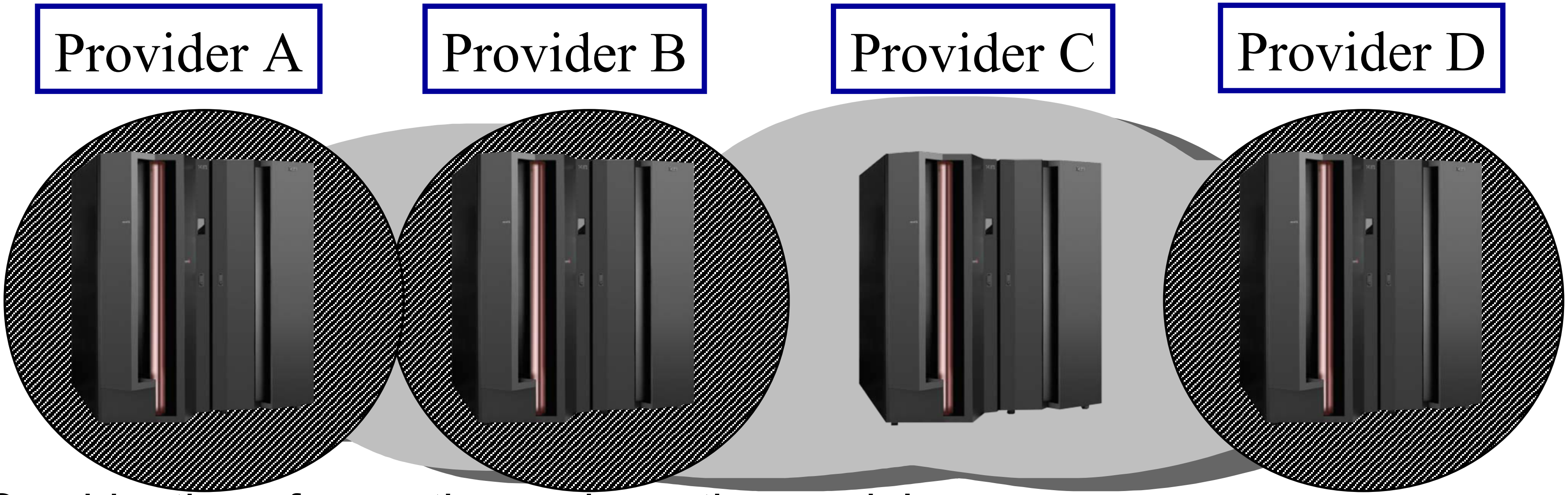
RAID designed for benign failures (*drive crashes*)

Static adversaries are not realistic

The *mobile adversary* moves from provider to provider (epoch)

- System failures and corruptions over time
- Corrupts a threshold of providers in each epoch (b out of n)

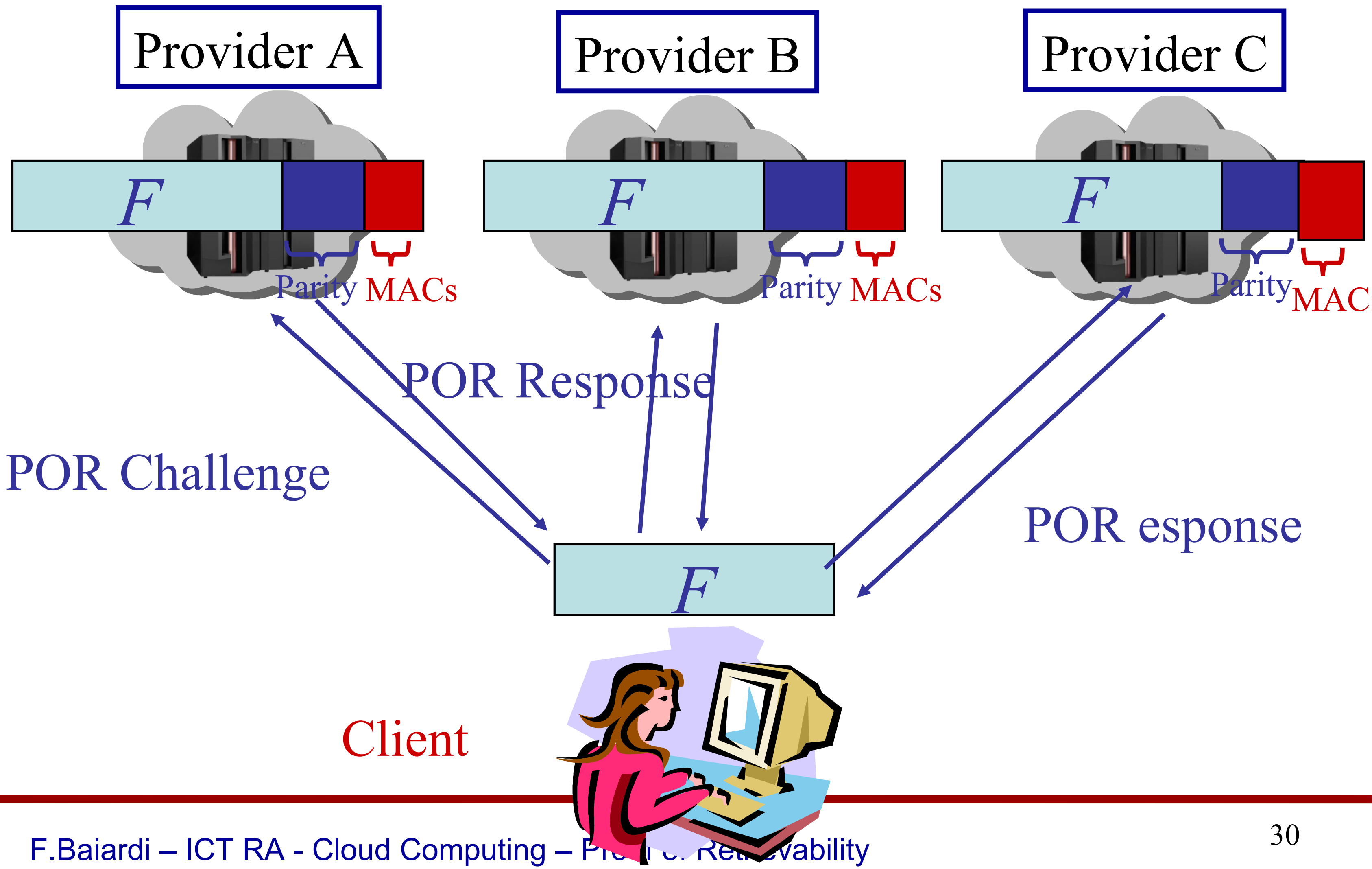
Mobile adversary



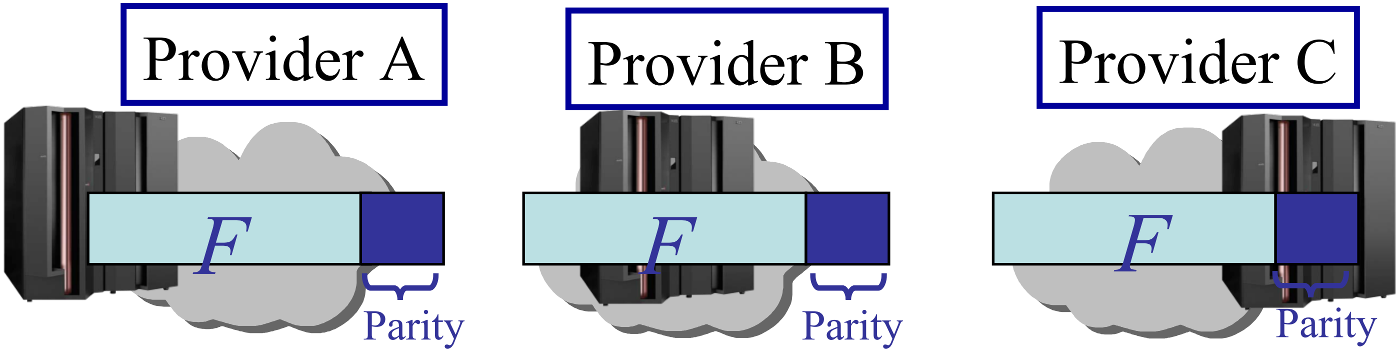
Combination of proactive and reactive models

- Separate each server into *code base* and *storage base*
  - Code base of servers cleaned at beginning of epoch (e.g., through reboot)
  - At most b out of n server have corrupted code in each epoch
- Challenge-responses used for detection of failure
  - Corrupted storage recovered when failure is detected

First idea: file replication with POR



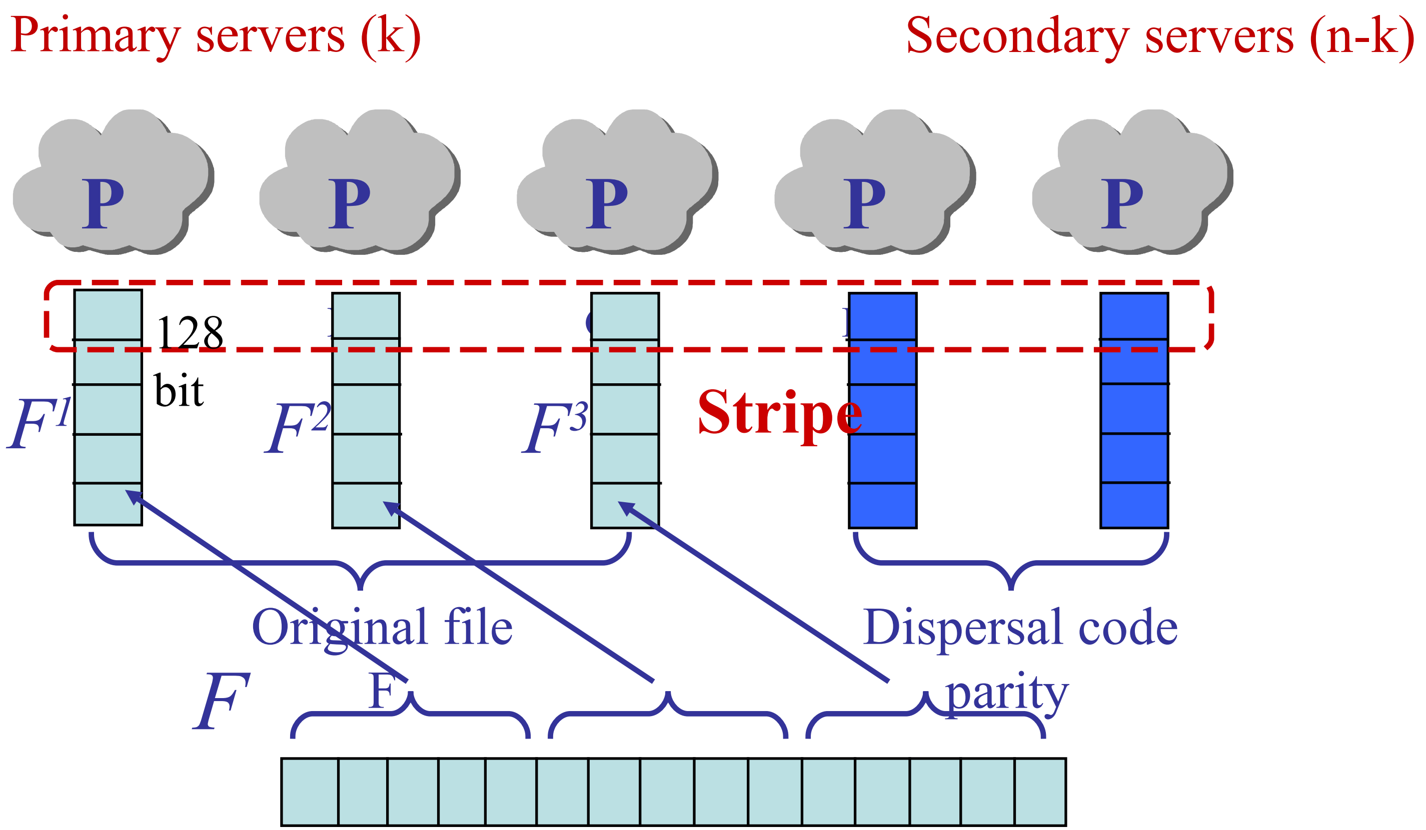
Replication with server code



- Still vulnerable to small-corruption attack, once corruption exceeds the error correction rate of server code
- Large storage overhead due to replication

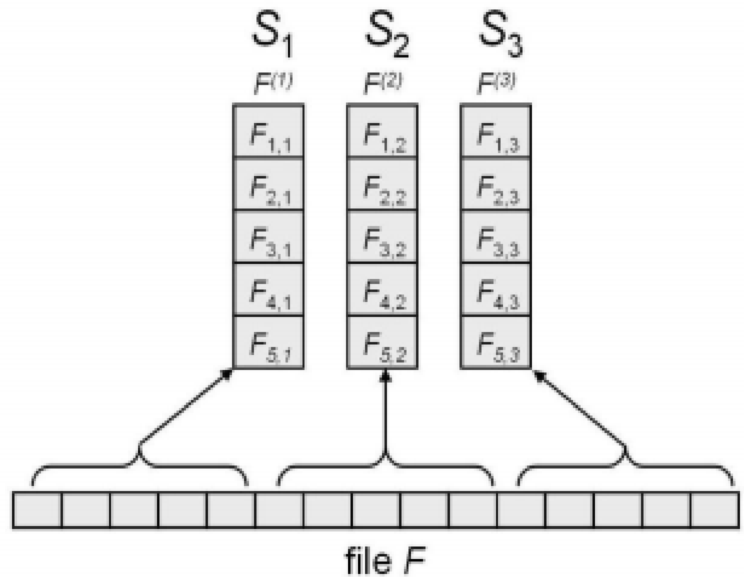
Client

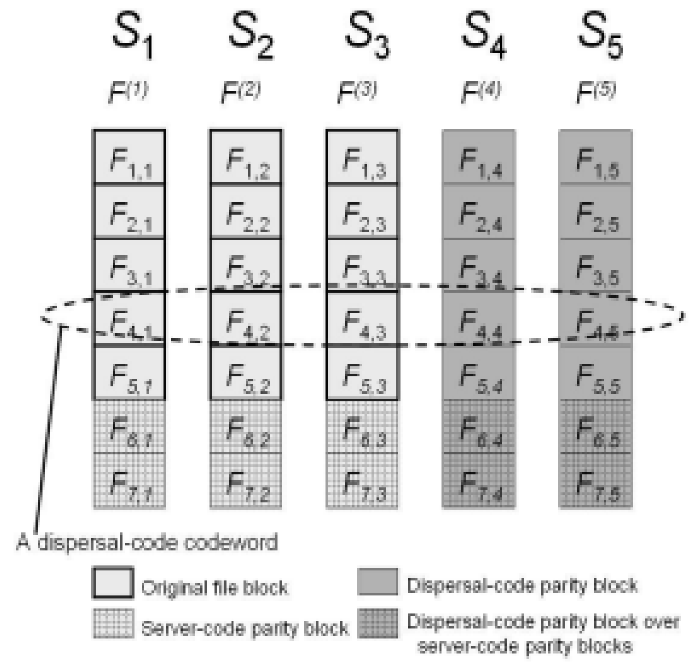
Dispersal erasure code



- File can be recovered from any k
- For encoding efficiency, use striping for 128-bit blocks

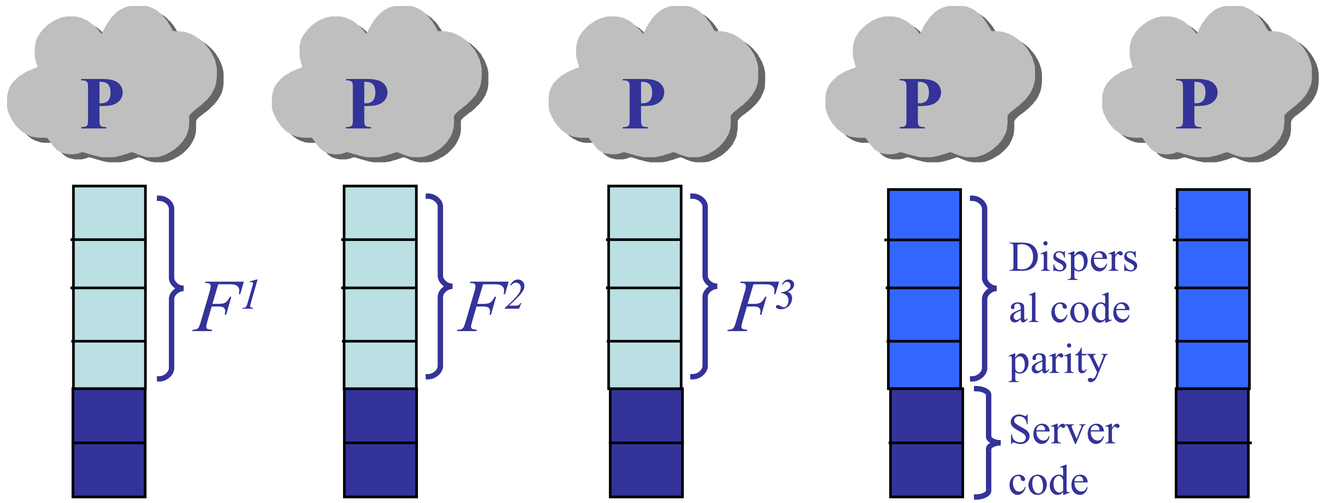
Dispersal erasure code





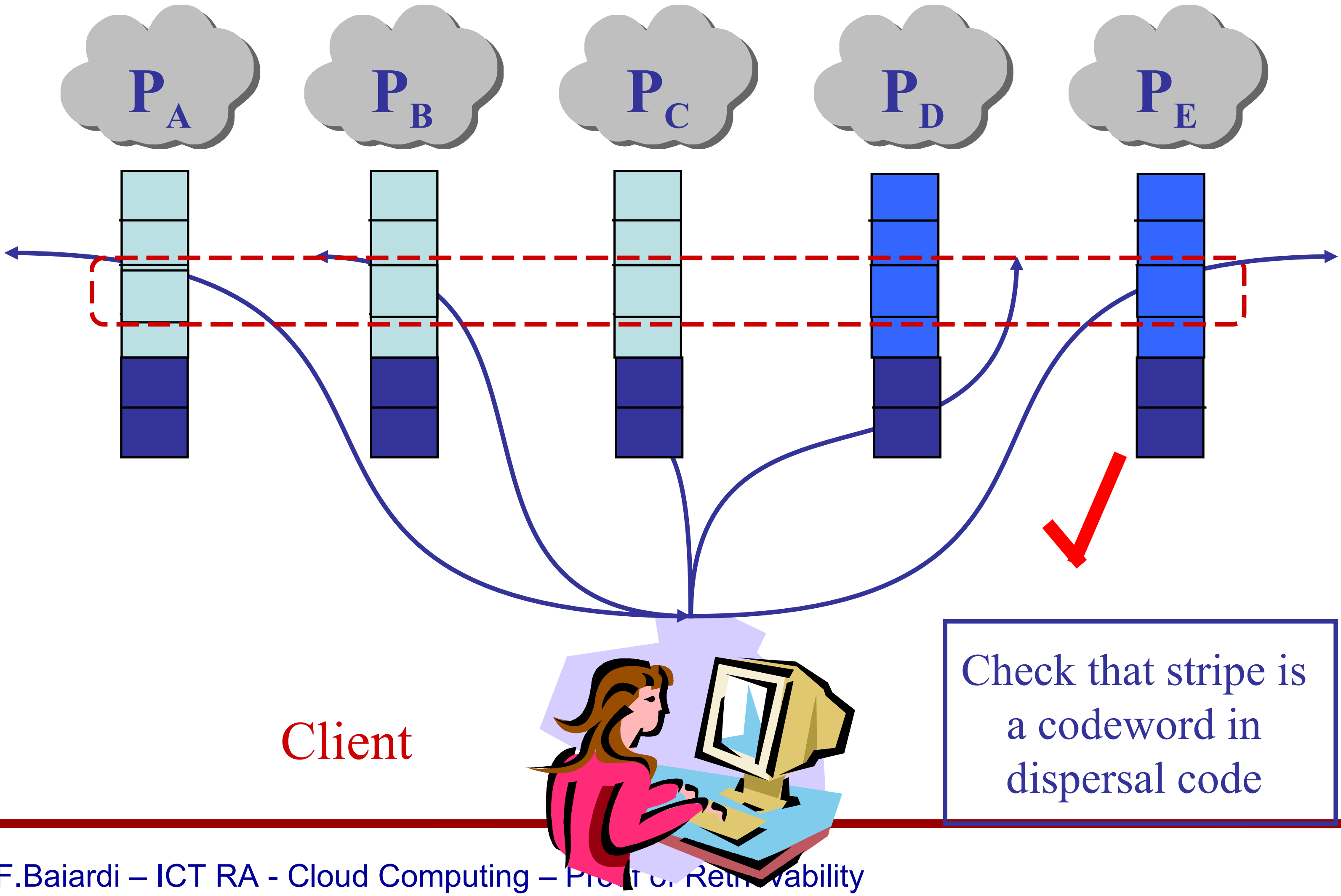
- File can be recovered from any k
- For encoding efficiency, use striping for 128-bit blocks

Two encoding layers



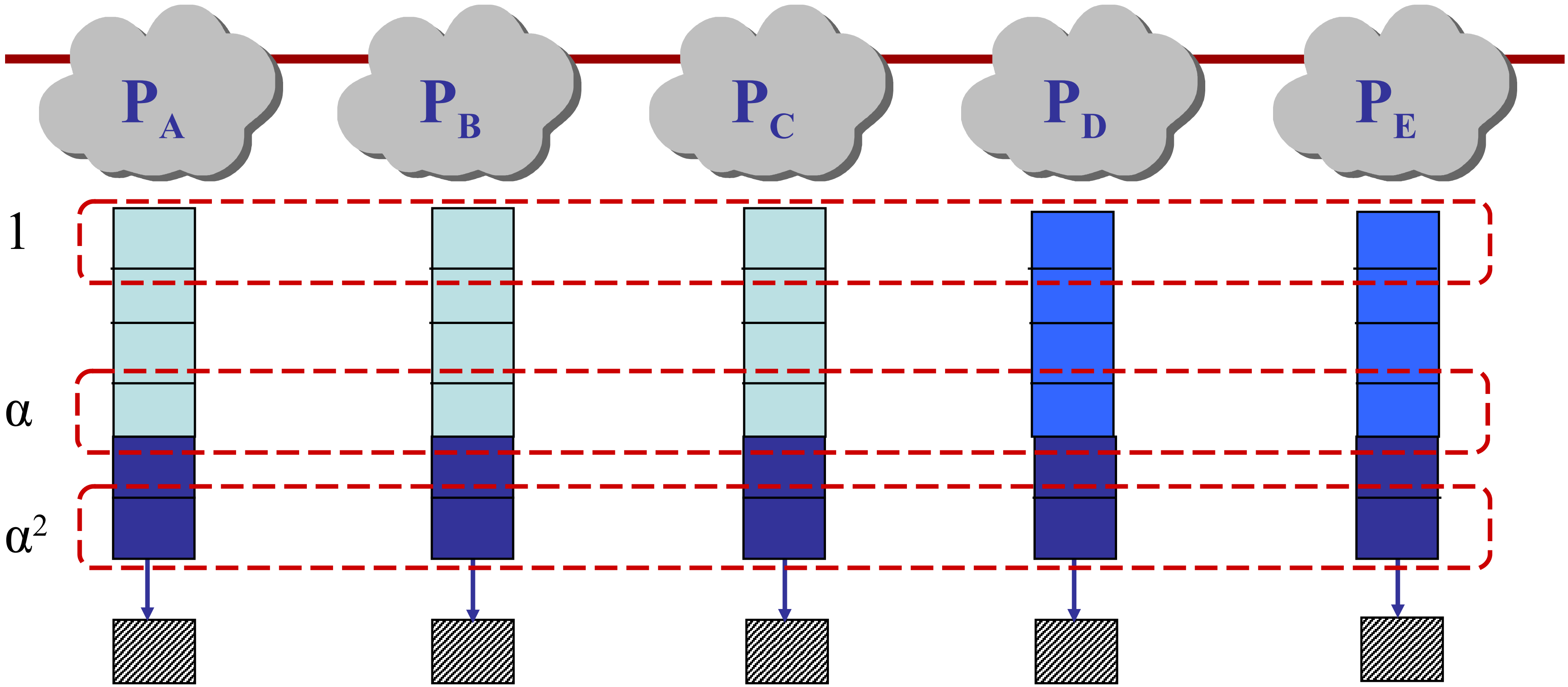
- Dispersal code reduces storage overhead of replication with similar availability guarantees
- Server code improves resilience to small-corruption attack it extend the “columns” of the encoded matrix by adding parity blocks.
- Then the dispersal code creates the parity blocks on the secondary servers.

Checking for correct encoding



Check that stripe is a codeword in dispersal code

Aggregation of stripes

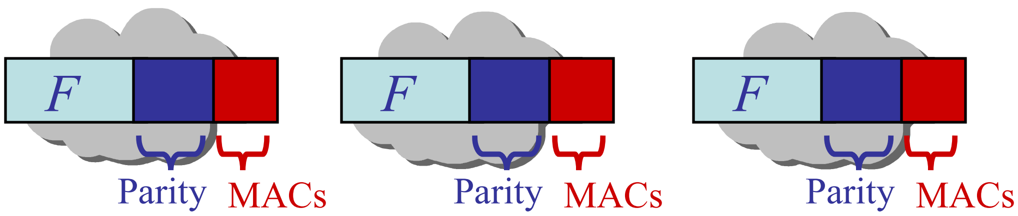


Client

Check that linear combination of stripes is a codeword

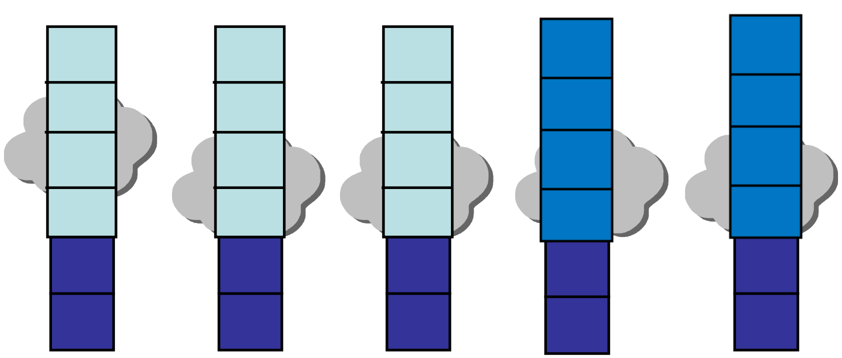
Comparison

File replication with POR



- Large storage overhead due to replication
- Redundant MACs for POR
- Large encoding overhead
- Verifiable by client only
+ Increased lifetime

HAIL:Two encoding layers (dispersal and server code)



+ Optimal storage overhead for given availability level
+ Uses cross-server redundancy for verifying responses
+ Reasonable encoding overhead
+ Public verifiability
- Limited lifetime

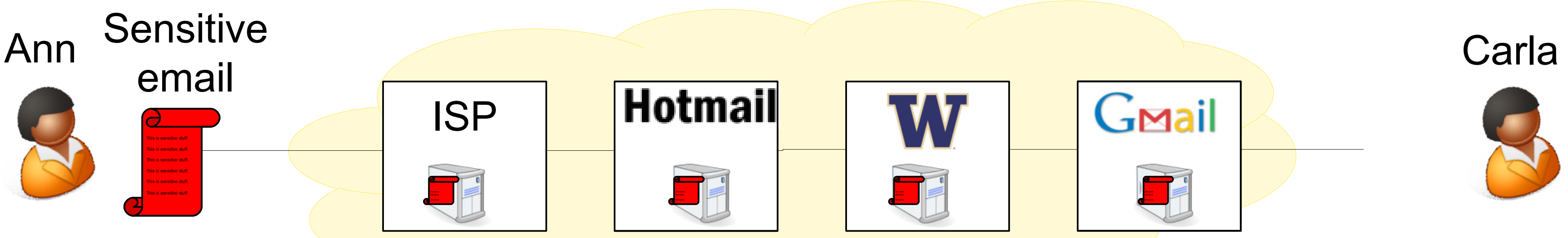
But we also have the inverse problem

- How can you be sure that data in the cloud has been erased?
- In general you cannot be sure if the data has been collected or created on the cloud
- But there are other solutions when data has been created outside and then stored in the cloud

Vanish: Increasing Data Privacy with Self-Destructing Data

R.Geambasu, T. Kohno, A. Levy, H.M. Levy.
Proceedings of the USENIX Security Symposium, Montreal, Canada, August 2009.

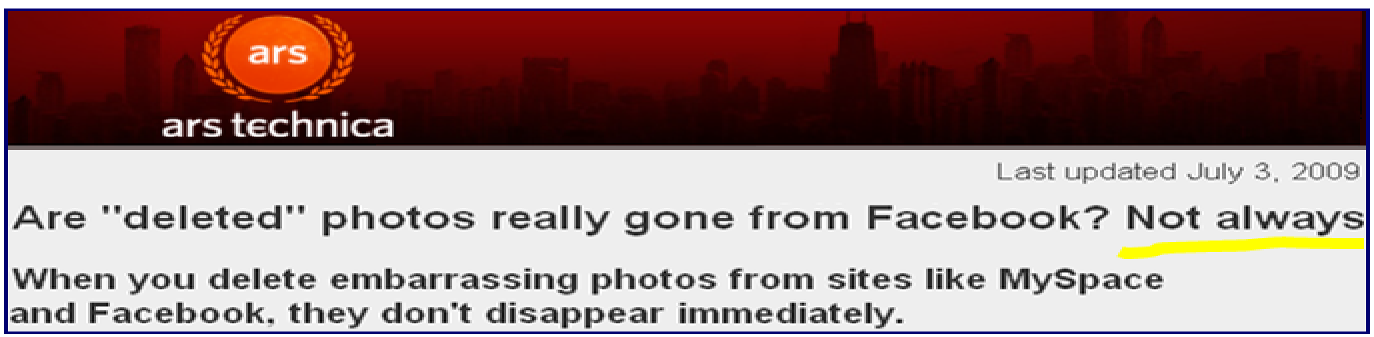
Motivating Problem: Data Lives Forever



How can Ann delete he…

She doesn't know where all the copies are

Services may retain data for long after user tries to delete

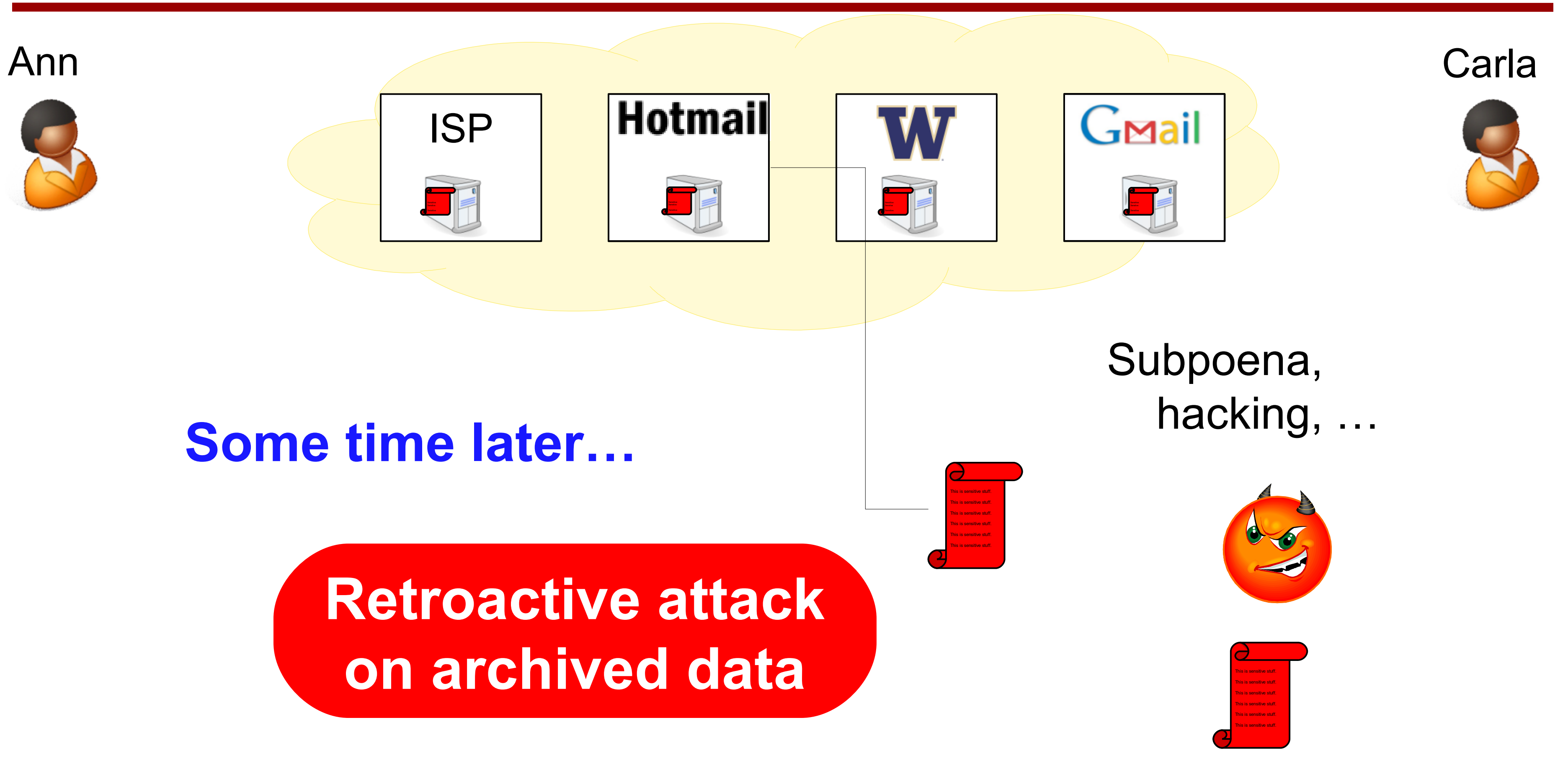

ars
ars technica

Last updated July 3, 2009

Are "deleted" photos really gone from Facebook? Not always

When you delete embarrassing photos from sites like MySpace and Facebook, they don't disappear immediately.

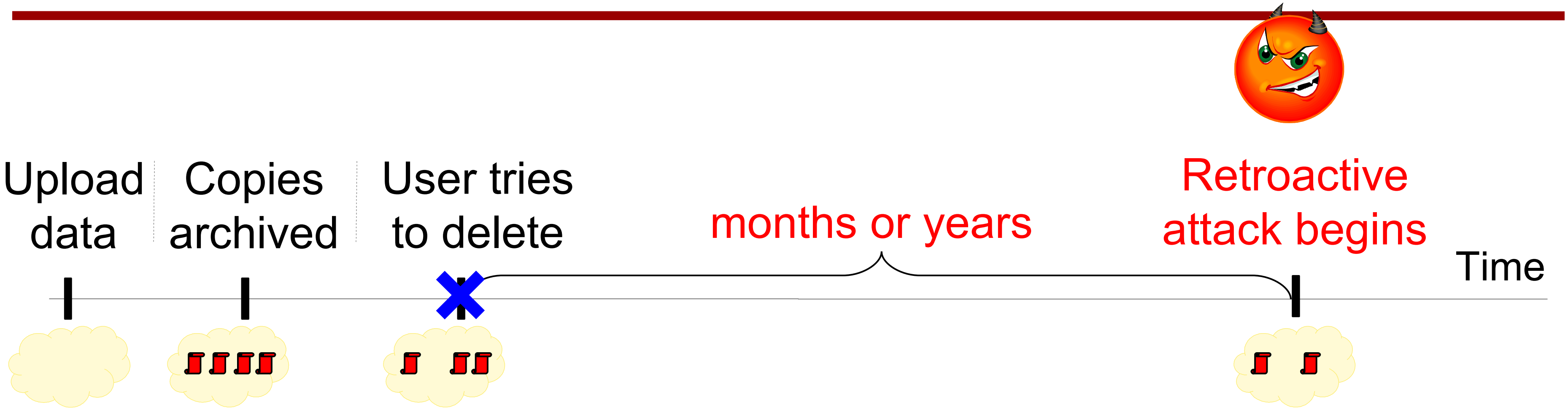
Archived Copies Can Resurface Years Later



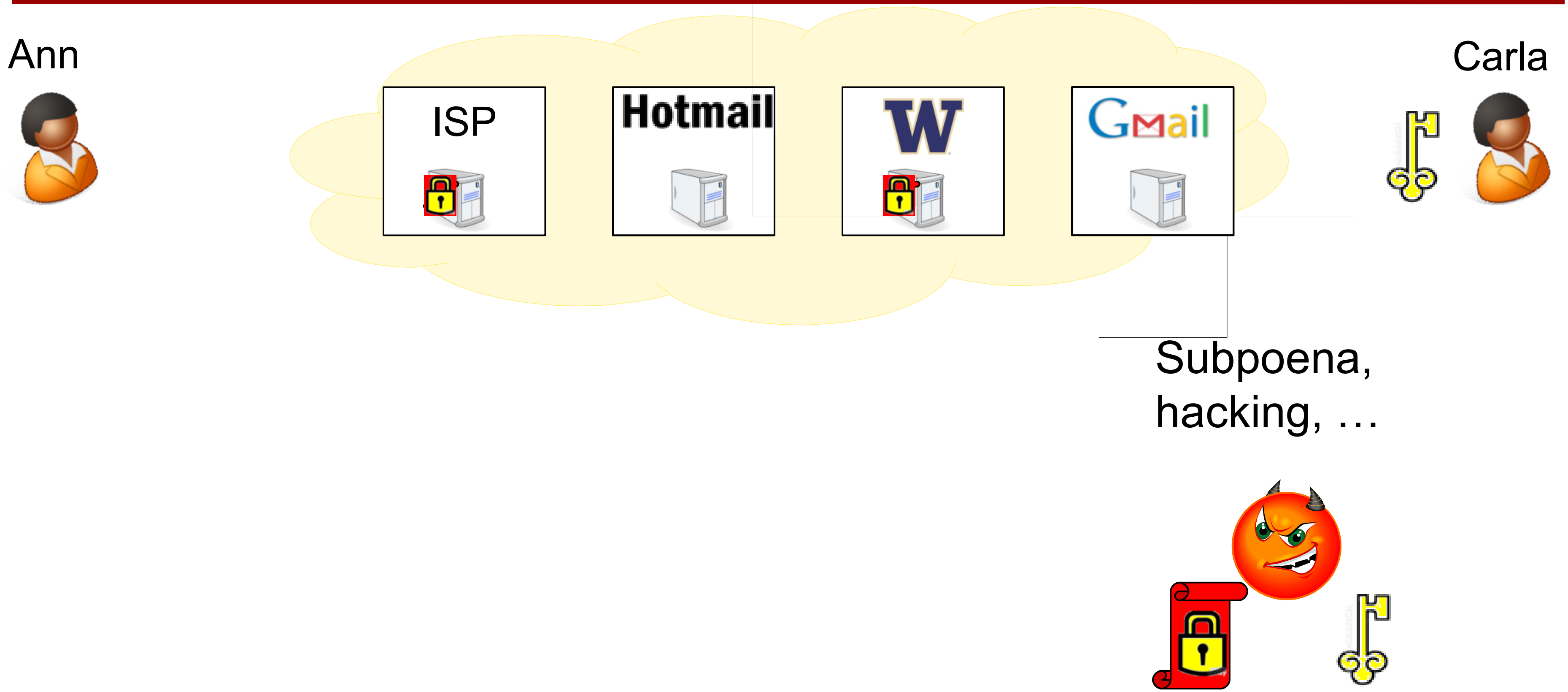
Some time later…

Retroactive attack on archived data

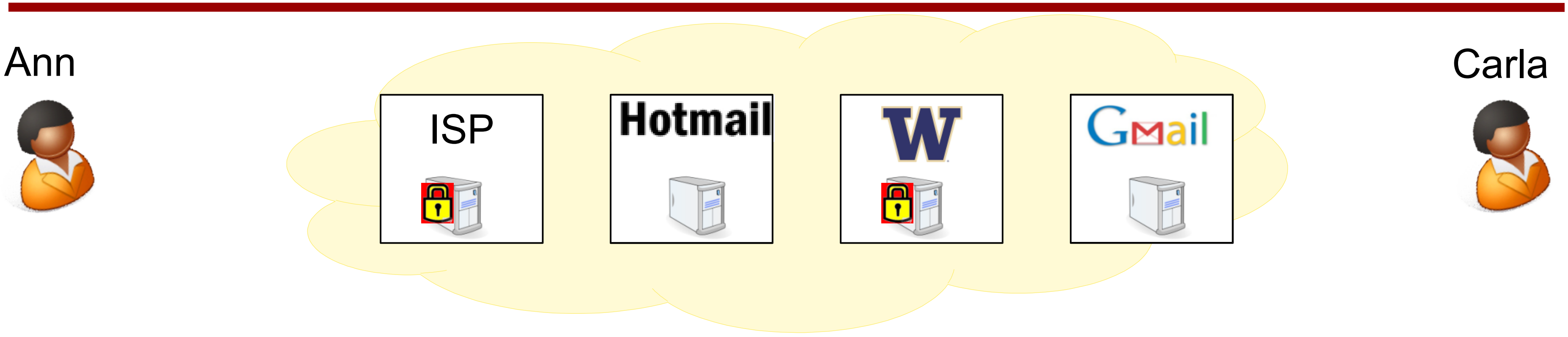
The Retroactive Attack



Why Not Use Encryption (e.g., PGP)?



Why Not Use a Centralized Service?



Centralized Service

“Trust us: we’ll help you delete your data on time.”

Backdoor agreement

The Problem: Two Huge Challenges for Privacy

Data lives forever

- On the web: emails, Facebook photos, Google Docs, blogs, …
- In the home: disks are cheap, so no need to ever delete data
- In your pocket: phones and USB sticks have GBs of storage

Retroactive disclosure of both data and user keys has become commonplace

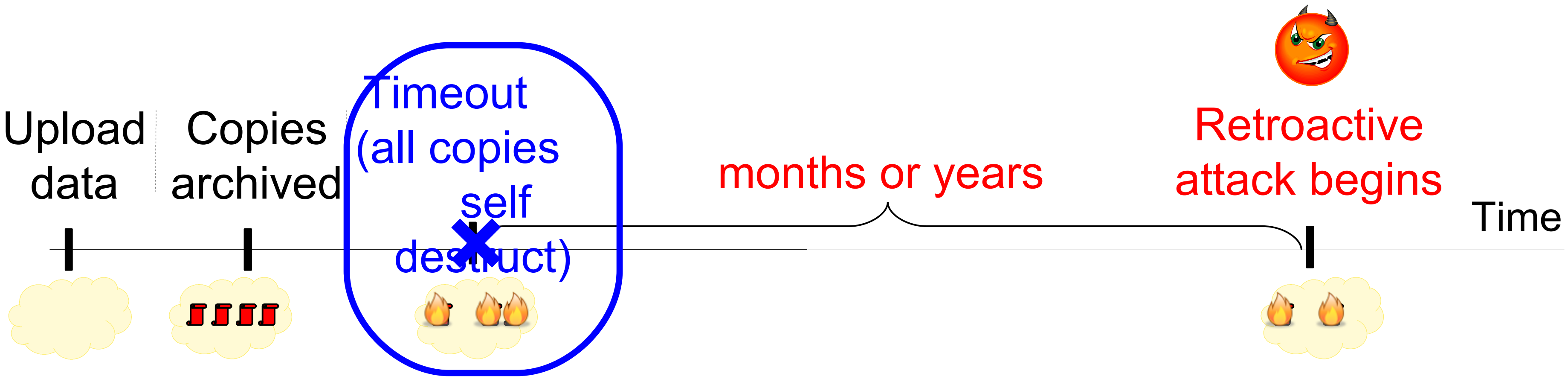
- Hackers
- Misconfigurations
- Legal actions
- Border seizing
- Theft
- Carelessness

Question:

Can we empower users with control of data lifetime?

Answer:

Self-destructing data



Self-Destructing Data Model

Goals

1. Until timeout, users can read original message
2. After timeout, all copies become permanently unreadable
   2.1. even for attackers who obtain an archived copy & user keys
   2.2. without requiring explicit delete action by user/services
   2.3. without having to trust any centralized services

A first solution by Disappearing Inc

- If Alice wishes to create an encrypted message for Bob, she contacts the ephemerizer, specifying an expiration time, and requesting a key.
- The ephemerizer chooses a random secret key K, assigns a key-ID IDK, tells Alice: (K, IDK), and remembers: (expiration time, K, IDK).
- Alice encrypts the message M with K (to obtain {M}K) and sends to Bob: ({M}K, IDK)
- When Bob wishes to decrypt the message,
  - he sends IDK to the ephemerizer
  - the ephemerizer replies with K,
  - Bob decrypt the message.

    Or
  - Sends ({M}K, IDK) to the ephemerizer that replies with M
- When expiration time occurs, the ephemerizer forgets K.

Vanish: Self-Destructing Data System

Traditional solutions are not sufficient for self-destructing data goals:

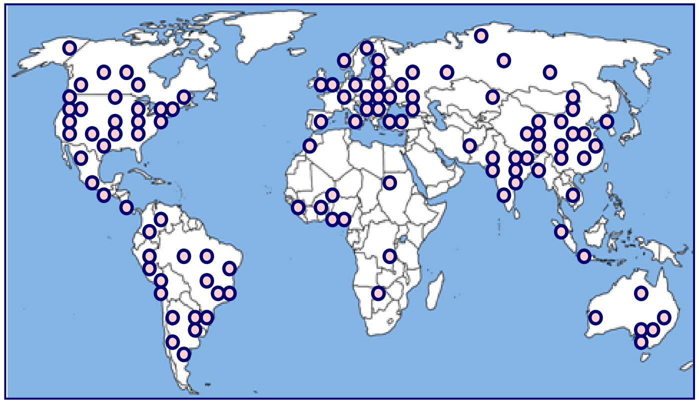
- PGP
- Centralized data management services
- Forward-secure encryption
- …

Let's try something completely new!

Idea:
Leverage P2P systems

P2P 101: Intro to Peer-To-Peer Systems

- A system of individually-owned computers that make a portion of their resources available directly to their peers without intermediary managed hosts or servers. [~wikipedia]



- Important properties (for Vanish):
- Huge scale – millions of nodes
- Geographic distribution – hundreds of countries
- Decentralization – individually-owned, no single point of trust
- Constant evolution – nodes constantly join and leave

Distributed Hashtables (DHTs)

Hashtable data structure implemented on a P2P network

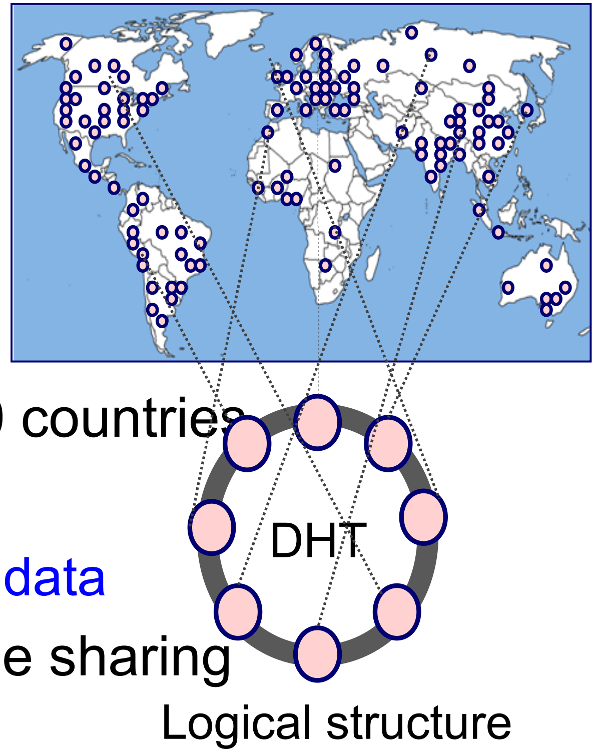
- Get and put (index, value) pairs
- Each node stores part of the index space

DHTs are part of many file sharing systems:

- Vuze, Mainline, KAD
- Vuze has ~1.5M simultaneous nodes in ~190 countries

Vanish leverages DHTs to provide self-destructing data

- One of few applications of DHTs outside of file sharing



Distributed Hashtables (DHTs)

Hashtable data structure implemented on a P2P network

Get and put (index, value) pairs

Each node stores part of the index space

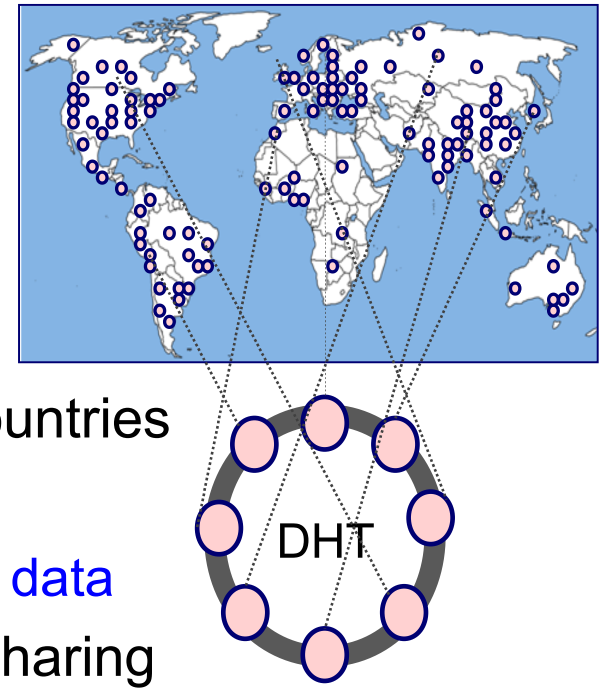
DHTs are part of many file sharing systems:

Vuze, Mainline, KAD

Vuze has ~1.5M simultaneous nodes in ~190 countries

Vanish leverages DHTs to provide self-destructing data

One of few applications of DHTs outside of file sharing



Logical structure

Shamir's (t, n)-threshold scheme:

a) D chooses prime p such that $p \leq n+1$, K in Z_p the group generated by p;

b) generates distinct, random, non-zero

x_i in Z_p, $i=1,\ldots,n$;

c) generates random $a_i \in Z_p$, $i=1, 2,\ldots, t-1$;

d) $a_0 = K$, the secret;

e) $f(x) = \sum_{i=0 \text{ to } t-1} a_i x^i \mod p$;

P_i's share is $(x_i, f(x_i))$.

Shamir's (t, n)-threshold scheme:

a) a_0 is the secret we want to share

b) generate a random polinomial of degree t-1 using a_0= generate t-1 random values a_1,..., a_{t-1}

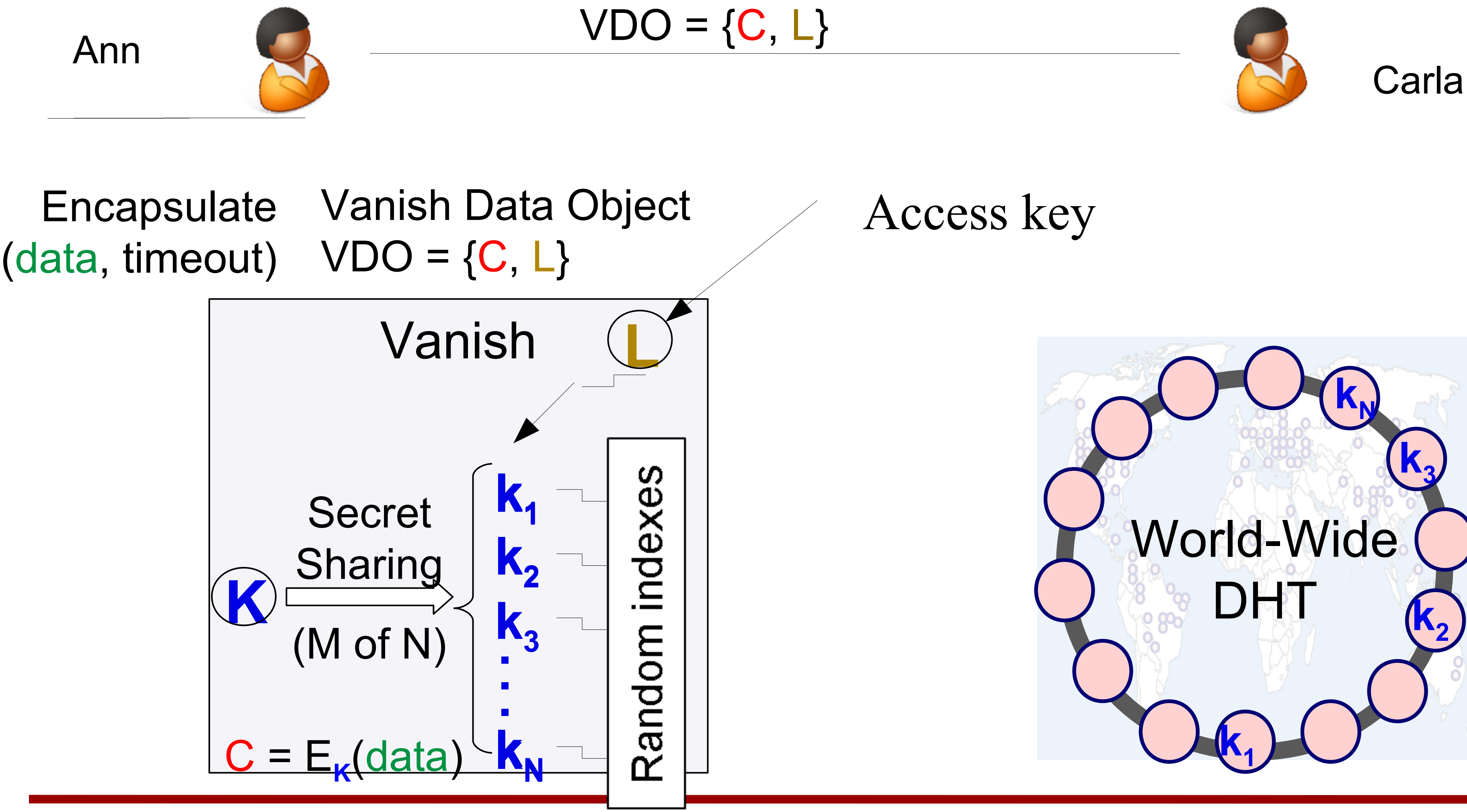
$$f(x) = a_{t-1}*x^{t-1}+a_{t-2}*x^{t-2}+.... + a_0$$

b) generate n non zero, distinct points x_1, …, x_n

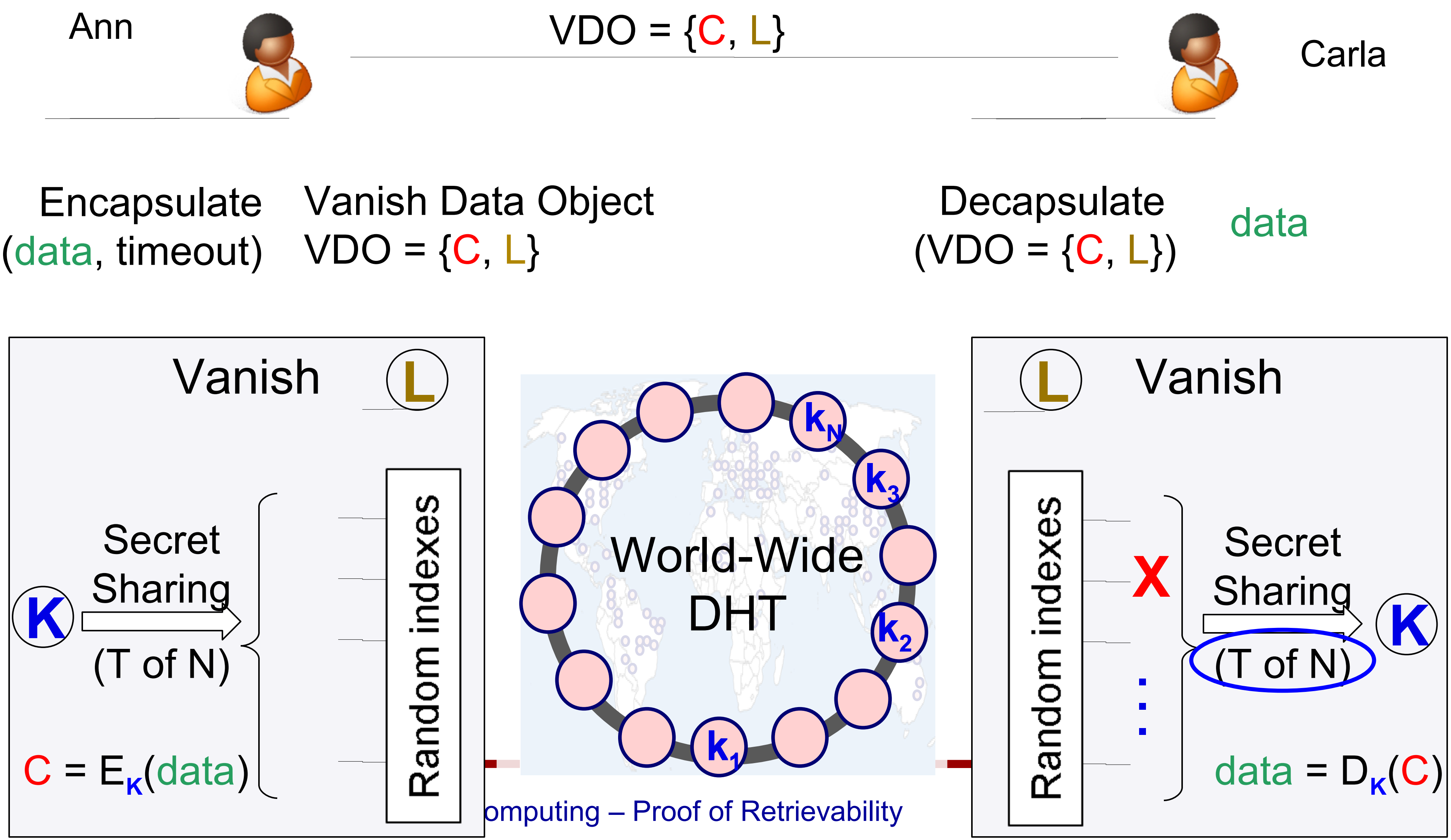
c) give to the i-th person the pair (x_i, $f(x_i)$)

d) if at least t out of n persons meet, they can interpolate the polinomial and discover a0

How Vanish Works: Data Encapsulation



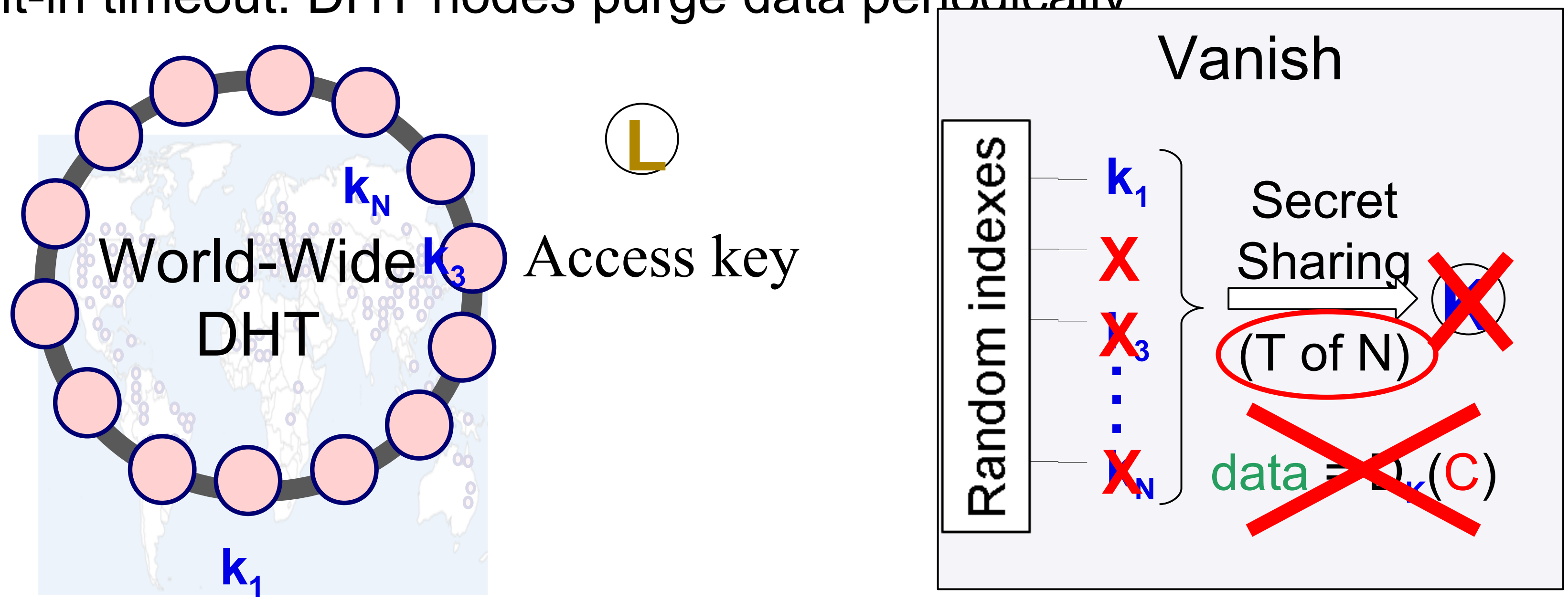
How Vanish Works: Data Decapsulation



How Vanish Works: Data Timeout

The DHT loses key pieces over time

- Natural churn: nodes crash or leave the DHT
- Built-in timeout: DHT nodes purge data periodically

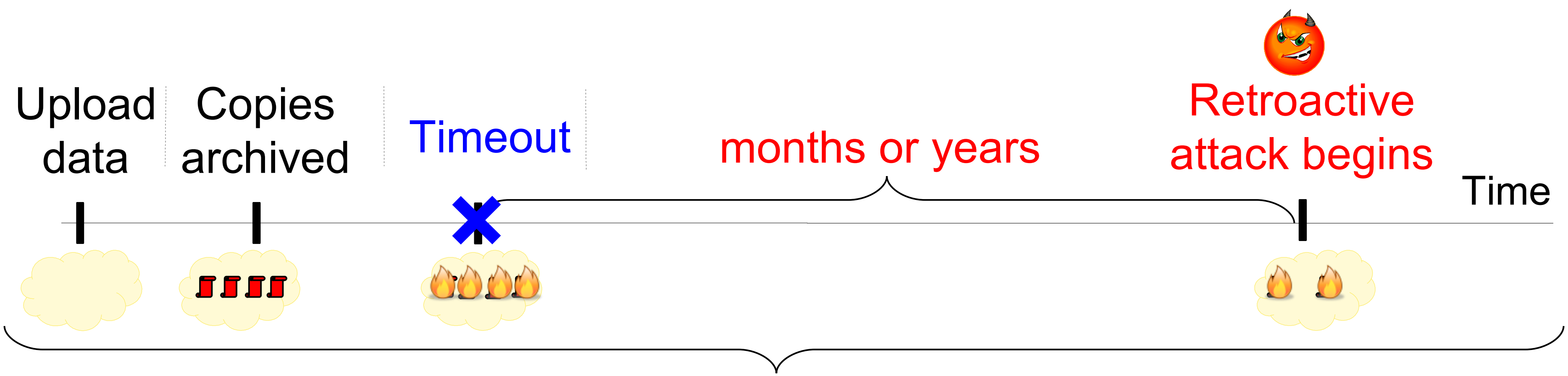


Key loss makes all data copies permanently unreadable

Threat Model

Goal: protect against retroactive attacks on old copies

- Attackers don’t know their target until after timeout
- Attackers may do non-targeted “pre-computations” at any time



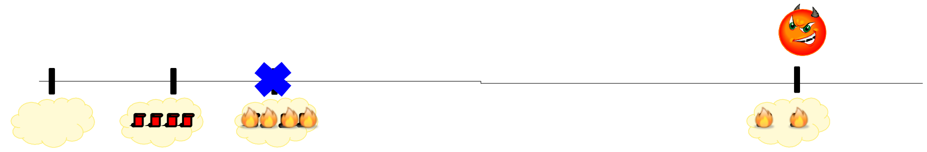
Communicating parties trust each other

- E.g., Ann trusts Carla not to keep a plain-text copy

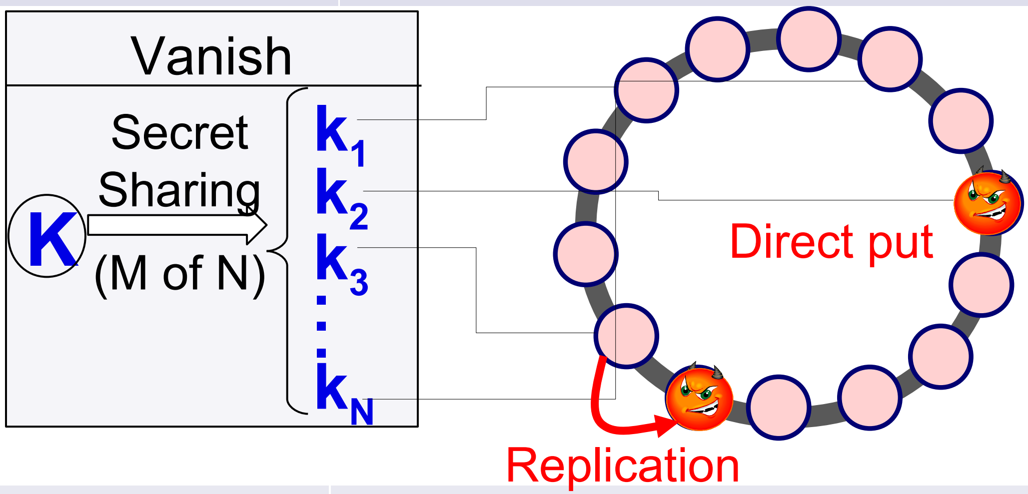
Attack Analysis

Retroactive Attack	Defense
Obtain data by legal means (e.g., subpoenas)	P2P properties: constant evolution, geographic distribution, decentralization
Gmail decapsulates all Vanish Data Object emails	Compose with traditional encryption (e.g., PGP)
ISP sniffs traffic	Anonymity systems (e.g., Tor)
DHT eclipse, routing attack	Defenses in DHT literature (e.g., constraints on routing table)
DHT Sybil attack	Defenses in DHT literature; Vuze offers some basic protection
Intercept DHT "get" requests & save results	Vanish obfuscates key share lookups
Capture key pieces from the DHT (pre-computation)	P2P property: huge scale
More (see paper)	

Retroactive Attacks



Attack	Defense
Obtain data by legal means (e.g., subpoenas)	P2P properties: constant evolution, geographic distribution, decentralization
Capture any key pieces from the DHT (pre-computation)	P2P property: huge scale

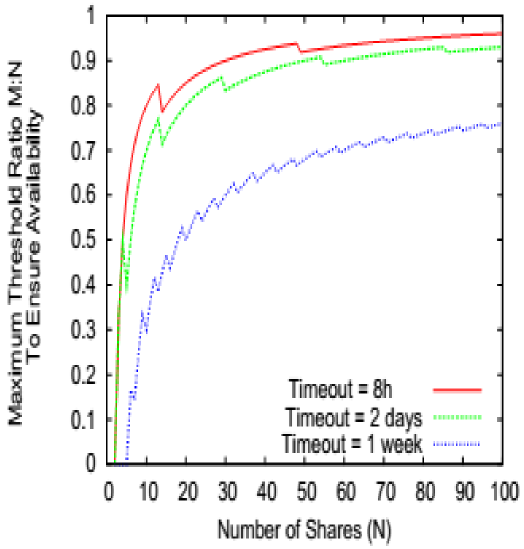


Given the huge DHT scale, how many nodes does the attacker need to be effective?

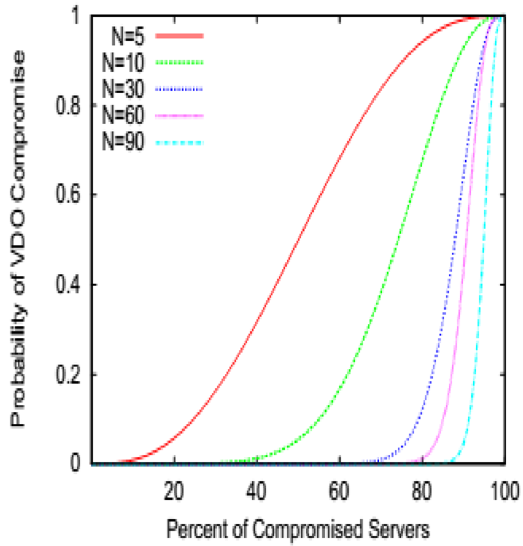
Current estimate: Attacker must join with ~8% of DHT size, for 25% capture

There may be other attacks (and defenses)

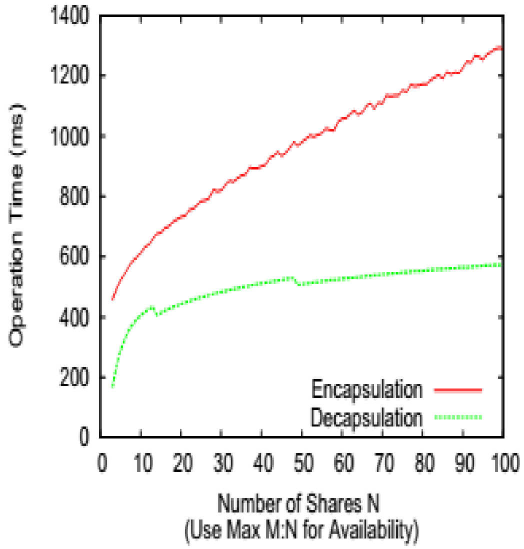
Performances



(a) Availability.



(b) Security.



(c) Performance.

Vanish Applications

Self-destructing data & Vanish support many applications

Example applications:

Firefox plugin

- Included in our release of Vanish

Thunderbird plugin

- Developed by the community two weeks after release ☺

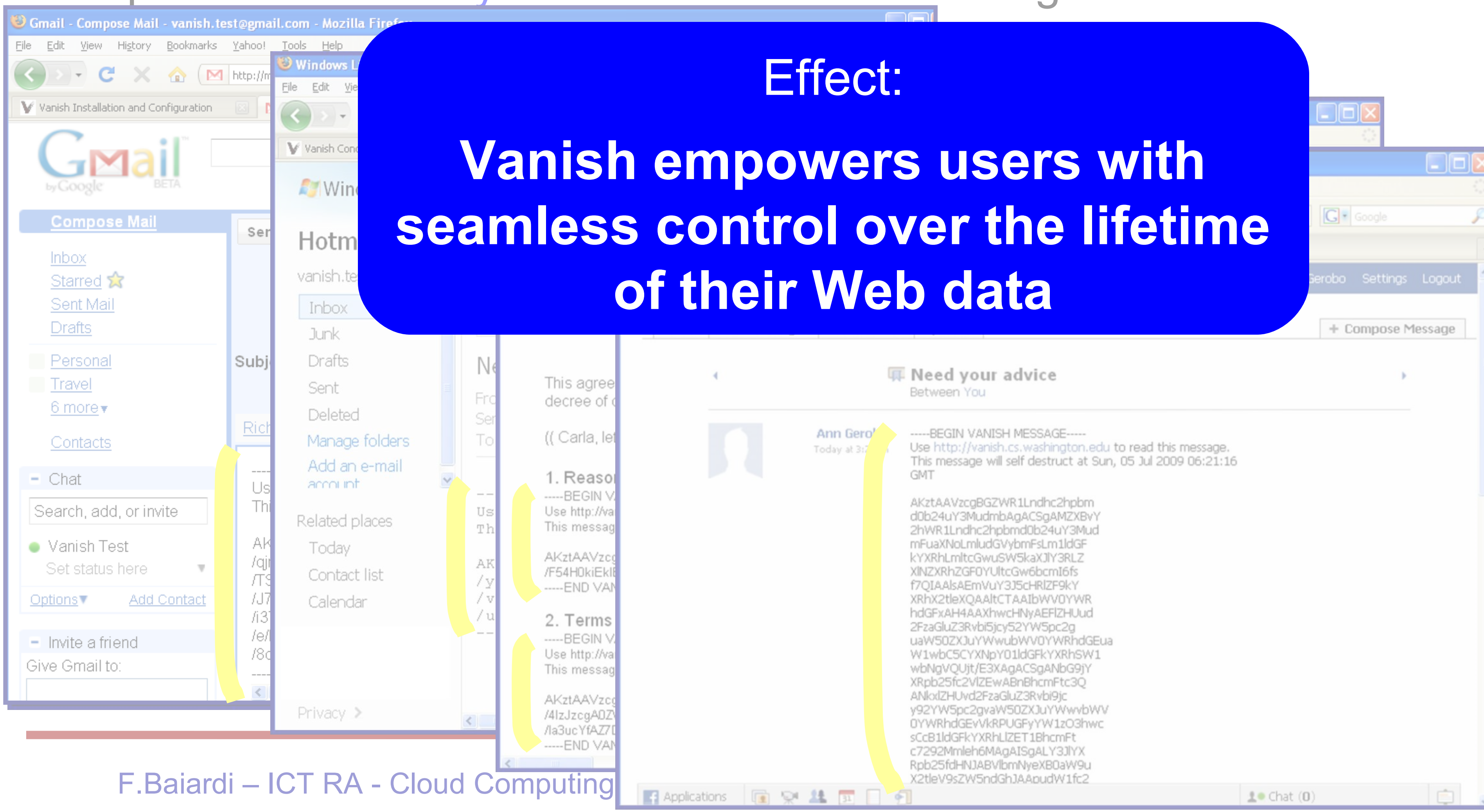
Self-destructing files

Self-destructing trash-bin

…

Firefox Plugin For Vanishing Web Data

Encapsulate text in any text area in self-destructing VDOs



Effect:

Vanish empowers users with seamless control over the lifetime of their Web data

Conclusions

Two formidable challenges to privacy:

http://vanish.cs.washington.edu/

- Data lives forever
- Disclosures of data and keys have become commonplace

Self-destructing data empowers users with lifetime control

Vanish:

- Combines global-scale DHTs with secret sharing to provide self-destructing data
- Firefox plugin allows users to set timeouts on text data anywhere on the web

Vanish ≠ Vuze-based Vanish

- Customized DHTs, hybrid approach, other P2P systems
- Further extensions for security in the paper

Attacking Vanish

Defeating Vanish with Low-Cost Sybil Attacks Against Large DHTs

Scott Wolchok†[1], Owen S. Hofmann†[2], Nadia Heninger[3], Edward W. Felten[3],
J. Alex Halderman[1], Christopher J. Rossbach[2], Brent Waters*[2], and Emmett Witchel[2]

[1]The University of Michigan
{swolchok,jhalderm}@eecs.umich.edu

[2]The University of Texas at Austin
{osh,rossbach,bwaters,witchel}@cs.utexas.edu

[3]Princeton University
{nadiah,felten}@cs.princeton.edu

September 18, 2009

The Sybil attack

One entity presents multiple identities for malicious intent.

Disrupt geographic and multi-path routing protocols by “being in more than one place at once” and reducing diversity.

Relevant in many context:

- P2P network
- Ad hoc networks
- Wireless sensor networks

Existing Work: Is Preventing Sybil Attacks Possible?

- John Douceur, Microsoft Research The Sybil Attack", IPTPS '01 (First International Workshop on Peer-to-Peer Systems (revised paper 2002))
- named and introduced problem
- strong negative theoretical results for networks without a centralized authority

Validation

Goal: accept all legitimate identities, but no counterfeits.

Verify identities:

- Direct validation
- Indirect validation

Direct validation

Validate the distinctness of two entities by asking them to perform task that one entity can not do:

If the communication resource is restricted, the verifier broadcasts a request for identities and then only accepts replies that occur within a given time interval.

If the storage resource is restricted, the verifier can challenges each identity to store large amount of unique data. The verifier verifies by keeping small excerpts of the data (sentinel).

If the computation resource is restricted, the verifier challenges each identity to solve a unique computational problem.

Direct validation

Assumption:

- all entities have identical resource constraints.
- all involved entities are verified simultaneously.

Extreme and unrealistic!

Indirect validation

Accept identities that have been validated by a sufficient count of other identities that it has already accepted.

Danger: a group of faulty entities can vouch for counterfeit identities.