Cloud Security: Remote Property Testing
Data Center
Who knows what evil lurks in the heart of the cloud?
Who knows what evil lurks in the heart of the cloud?

- Trojans
- viruses
- corner-cutting cloud providers
- faulty hardware
- faulty software
Who knows what a mess lurks in the heart of the cloud?
In cloud storage...

One day, Alice’s machine crashes, so she contacts MegArchive...
Financially motivated service degradation

- MegArchive moves Alice’s file to a slow disk array
  - File is there, but retrieval is unacceptably slow
- MegArchive uses cheap storage that degrades over time
- MegArchive throws away a portion of Alice’s file to save space
  - Most users don’t retrieve their backup files anyway

Question: How can Alice be sure that she can retrieve her file in its entirety?
Remote Property Testing

Idea: Check that a given property holds without trusting Cloud itself
What properties $Z$ might we test? (General)

- $Z = \text{“Are my resources properly isolated from other tenants’?”}
  - E.g., VM isolation (due to side channels)
- $Z = \text{“Is my cloud provider’s infrastructure adequately secured?”}
  - E.g., Are its firewalls correctly configured?
- $Z = \text{“Is my cloud provider billing me fairly?”}
What properties Z might we test? (Data)

• Z = “Is the Cloud encrypting my files at rest?”
  – Common regulatory requirement (e.g., HIPAA)

• Z = Geolocation, e.g., “Is my data accessible only in the United States?”
  – Huge commercial demand
What properties $Z$ might we test? (Data)

- $Z = "Can my files survive a drive crash?"
- $Z = "Can my files can survive a service-provider failure?"
- $Z = "Are my files intact right now?"
Proofs of Storage

Proofs of Retrievability (PORs) +
Proofs of Data Possession (PDPs)
Setup

• Alice has a file $F = f_1 \parallel f_2 \parallel f_3 \ldots \parallel f_n$
• Alice would like to ensure that $F$ is intact
• Why is this not enough?
• She also wants to ensure that she can retrieve it!
• Alice would like:
  – A way to make periodic, efficient checks of file intactness
  – Not to store very much herself
• She may also want to ensure compliance with a Service-Level Agreement (SLA)
Strawman approaches

• Approach 1: Alice periodically downloads $F$
  – Why not?
  – Resource-intensive!

• Approach 2: Alice stores a hash $H(F)$, asks cloud to send her valid hash
  – Why not?
  – Cloud can easily cheat
Strawman approaches

• Approach 3: Alice stores \(\{H(F||r_i)\}_i\), challenges cloud with \(r_i\)
  – Why not?
  – Alice’s storage linear in number of queries
  – Cloud must process *whole file* to respond

• What about spot-checking instead?
Functions

- **KeyGen**(1^k) → K
  - Alice generates a secret key
- **Setup**(F, K) → F’
  - Alice processes file for upload
- **Challenge**(K, [B]) → c
  - Alice generates a challenge and sends to Cloud
- **Respond**(F, c) → r
  - Cloud responds to challenge against F
- **Verify**(K, [B], r) → {“accept”, “reject”}
  - Alice checks Cloud’s response
- **Extract**(K)
  - Alice recovers F (if normal interface doesn’t respond)
Spot checking: Preparation

Cloud

\[ f_2 f_1 f_3 \]

Alice
Spot checking: Verification

Cloud

\[ \tilde{f}_2 \tilde{f}_1 \tilde{f}_3 \]

\[ \tilde{F} \]

Alice

\[ f_1 f_2 f_3 \]
Spot checking: Verification

Cloud

\[ f_2 \quad \times \quad f_3 \]

\[ \tilde{F} \]

Alice

\[ f_1 \quad f_2 \quad f_3 \]
Spot checking

Pros:
- Alice needn’t download all of $\tilde{F}$
- Can detect large erasure / corruption

Cons:
- Alice must store chunks of $\tilde{F}$
- Can’t detect small erasure / corruption
Message Authentication Code (MAC) in PORs [JK ’07]
MAC approach: Preparation

Cloud

$F$

$\text{MAC}_k[f_1]$

Alice

$k$
MAC approach: Preparation

Cloud

Cloud:

\[
\begin{array}{c}
\text{Alice} \\
\end{array}
\]

\[
\begin{array}{cccc}
& f_2 & f_1 & f_3 \\
\end{array}
\]

\[
\begin{array}{c}
\text{Alice} \\
\end{array}
\]

\[
\begin{array}{c}
k \\
\end{array}
\]

\[
\begin{array}{c}
\text{Cloud} \\
\end{array}
\]

\[
\begin{array}{cccc}
& c_1 & c_2 & c_3 \\
\end{array}
\]

\[
\begin{array}{c}
F \\
\end{array}
\]
MAC approach: Verification

Pros:
• Alice needn’t store any of $F$
• Can detect large erasure / corruption

Cons:
• Can’t detect small erasure / corruption
The PDP approach

[ABCHKPS ‘07]

- Uses homomorphic “tags”
  - Basically function like “block-specific” MACs
  - (Extensible to public verification)

- KeyGen
  - Public:
    - RSA modulus $N = (2p’+1)(2q’+1)$,
    - $g$ generator of $Q_N$
  - Secret:
    - Large primes $(e,d)$ s.t. $ed = 1 \mod (p’q’)$
    - Secret key $v$ (l-bit random value)

- Computations are mod $N$
Hardness assumptions

• RSA assumption:
  – Given \((N,e)\), computing \(e\)th roots is hard

• KEA-r (Knowledge of Exponent Assumption, RSA ring)
  – For any adversary \(A\) that takes input \((N, g, g^s)\) and returns \((C, Y)\) s.t. \(Y = C^s\), there exists an extractor \(A^*\) which, given same inputs as \(A\), returns \(x\) s.t. \(C = g^x\).
  – I.e., if you can produce \((g^m, g^{sm})\), you must “know” \(m\)
The PDP approach: Basic tag construction

- $T_i = (w_i g^{m_i})^d = (H(v||i) g^{m_i})^{1/e} \mod N$
  - $w_i = H(v||i)$
  - $H: \{0,1\}^* \rightarrow Q_N$

- To verify $m_i$, $T_i$:
  - Verifier computes $w_i$
  - Verifier checks $(T_i)^e = w_i g^{m_i}$
  - Why is $w_i$ necessary?

- Expensive to verify for many blocks
- Can we aggregate?
The PDP approach: Aggregation

• **Observe homomorphic property:**
  
  – \( T_i T_j = w_i w_j g^{m_i + m_j} \)
  
  – \( T_i^{a_i} T_j^{a_j} = w_i^{a_i} w_j^{a_j} g^{a_i m_i + a_j m_j} \)

• **Challenge**
  
  – \( c = \{i_1, i_2, \ldots, i_k\}, g_s = g^s \)
    
    • (Plus (pseudo)random \( a_{i_1}, a_{i_2}, \ldots, a_{i_k} \))

• **Response**
  
  – \( T = (T_{i_1}^{a_{i_1}} T_{i_2}^{a_{i_2}} \ldots T_{i_k}^{a_{i_k}}), \ H(g_s^{a_{i_1} m_{i_1} + a_{i_2} m_{i_2} \ldots a_{i_k} m_{i_k}) \)
The PDP approach: Aggregation

• Response
  
  \[ T = (T_{i1}^{ai1} T_{i2}^{ai2} \ldots T_{ik}^{aik}), \quad \rho = H(g_s^{a_i m_i_1 + a_{i_2} m_{i_2} \ldots a_{i_k} m_{ik}}) \]

• Verification?
  
  – Compute \( u = T^e / (w_{i1}^{ai1} w_{i2}^{ai2} \ldots w_{ik}^{aik}) \)
  
  – Recall \( T_i = (w_i g^{m_i})^d \)
  
  – Thus \( u = g^{a_i m_{i_1} + a_{i_2} m_{i_2} \ldots a_{i_k} m_{ik}} \)
  
  – Check \( u^s = \rho \)
  
  – Intuition:
    
    • Tags show \( \rho \) hash of \( g_s^{a_i m_{i_1} + a_{i_2} m_{i_2} \ldots a_{i_k} m_{ik}} \)
    
    • Why hash?
    
    • KEA-r implies cloud “knows” \( a_i m_{i_1} + a_{i_2} m_{i_2} \ldots a_{i_k} m_{ik} \)
What good is knowing a linear function of blocks?

• “Knowing” \( a_{i1}m_{i1} + a_{i2}m_{i2} \ldots + a_{ik}m_{ik} \) does not imply “knowledge” of \( F \! \)
• How to obtain “knowledge” of individual blocks?
• Observe that (in finite field) for random \( r_1, r_2 \), you can’t compute \( m_1 \) or \( m_2 \) from \( r_1m_1 + r_2m_2 \)
• But you can, w.h.p., from
  \[- r_1m_1 + r_2m_2 \]
  \[- r'_1m_1 + r'_2m_2 \]
What good is knowing a linear function of blocks?

- Extraction idea:
  - Extractor obtains $a_{i1}m_{i1} + a_{i2}m_{i2} \ldots + a_{ik}m_{ik}$ from RO for $H$ and KEA-r
  - Extract many linear functions of blocks by repeating protocol
  - Solve system of equation for blocks
  - Random $a$ values help achieve linear independence
    » (Can do better with error-correcting codes [BJO ‘09])
Benefits of PDPs

• Arbitrarily many challenges
  – Unlike simplest JK ‘07 construction
• Publicly verifiable variant
Homomorphic MAC idea

[SW’08]

- $T_i = \text{PRF}_k(i) + \alpha m_i$
  - Pseudorandom function + universal hash = MAC[$m_i$]
  - (Not observed by SW, but implicit in construction [BJO ‘09])

- Homomorphism:
  - $a_i T_i + a_j T_j = \text{PRF}_k(i) + \text{PRF}_k(j) + \alpha (a_i m_i + a_j m_j)$

- Roughly same idea as PDP
  - Challenge: $(a_1, a_2... a_k)$
  - Proof: $(T = \Sigma_i a_i T_i, \rho = \Sigma_i a_i m_i)$
  - $T - \Sigma_i a_i \text{PRF}_k(i) = \alpha \rho$
  - Can extract from many $\rho$ response

- Allows arbitrarily many challenges
  - (Still, more storage than basic POR in many practical settings)

- Publicly verifiable variant
Problems with PDPs

1. What’s cost of computing tags, i.e., Setup?
   – One modular multiplication per bit!

2. What if cloud deletes a single block?
   – Low probability of detection!

3. How does Extract work?
   – “Knowledge” of cloud does not equate with knowledge by client!

• PORs better for 1-3
• Now let’s examine 2 and 3
• With ECC to encode $F \rightarrow F^*$, big error in $F^*$ needed to induce any error in $F$
• In effect, we can amplify errors in stored file
ECC + (stored) MAC

Cloud

\[ f_2 \quad f_1 \quad f_3 \quad \text{parity bits} \quad c_1 \quad c_2 \quad c_3 \quad c_4 \]

Alice

\[ k \]

\[ F^* \]

Pros:
- Alice needn’t store any of \( F \)—only key \( k \)
- Alice can detect any corruption in \( F \) w.h.p.
  (= large corruption in \( F^* \) )
ECC + MAC: Verification

Example:
- ECC corrects up to 10% corruption
- Alice checks 30 positions
- Each check detects corruption with probability $\geq 0.9$
- Alice detects insurmountable corruption with prob. $\geq 1 - 0.9^{30} > 95\%$
Another problem

- Classical error-correcting codes (e.g., Reed-Solomon) operate on small blocks.
- “Striping” allows efficient operation on large blocks, but only a small number of them (e.g., 256).
- Introduces structure into code.
- Adversary can surgically target single codeword.
  - I.e., take advantage of structure.
Another problem

- Classical error-correcting codes (e.g., Reed-Solomon) operate on small blocks
- “Striping” allows efficient operation on large blocks, but only a small number of them (e.g., 256)
- Introduces structure into code
- Adversary can surgically target single codeword
Idea

• Permute and encrypt blocks
• Note: We only need to encrypt parity blocks
  – Why?
• Problem: file updates are now hard!
  – Why?
Another idea

- Use error-correcting code with massive block size
  - Eliminates local structure!
  - (Maximum distance separable (MDS))
  - (Technically erasure code + data authentication)

- [SSP ’13]
  - Fourier-like construction

- $O(n \log n)$
  - Practical constants
Another challenge: Extraction

“May I have my file now?”

“Not from me, sucker!”
What can we do about this?

- Allow Alice to retrieve file from **responses to challenges** ("extraction")
  - POR approach
- But won’t a flurry of challenges tip off the server?
- Assume **memoryless** server, e.g., virtual machine that can be replayed or
- Assume rational adversary, e.g., service provider that wants to save space, but otherwise cooperates
  - Needed for PDP
Someone may have noticed…
POSs seem not to solve this problem!

• But what about *speed* of a response?
• POSs can check *quality of service*
Where to use Proofs of Storage?

• Where is a quickly detected failure tolerable?
• Backup!
• Idea:
  – Use POSs to verify backed up files
  – In case of failure, change providers or restore from primary copy
• Another application: Quality of storage in backup
  – E.g., are my files on disk or on tapes conveyed by mule from a mountain bunker?
Where POSs blow up

What’s Alice to do???

E.g., T-Mobile / Microsoft
RAID
(Redundant Array of Inexpensive Disks)

File block

File block

File block

Parity block

\[ F \rightarrow F_1 \quad F_2 \quad F_3 \quad F_1 \oplus F_2 \oplus F_3 \]
May not work in Cloud

• What if service providers lose data but... don’t tell you until file is lost?
A mobile adversary moves from device to device, corrupting as it goes—potentially silently.

Mobile adversary models, e.g., system failures / corruptions over time, malware propagation:
- In the long run, this is how things really happen!

RAID isn’t designed for this kind of adversary:
- Designed for limited, readily detectable failures in devices you own—the benign case.
Mobile adversary

- In cryptography, usual approach to mobile adversary is *proactive*
• In cryptography, usual approach to mobile adversary is *proactive*.

• Another, cheaper possibility is *reactive*: We *detect and remediate*
  – Like whack-a-mole!

• POSs can provide detection here…
Applying POSs
Applying POSs
Applying POSs

Alice
HAIL: High Availability and Integrity Layer

[BJO ‘09]

- RAID-type redundancy tolerates \( t \) drive failures
- POS helps bound number of failures thanks to “whack-a-mole”
  - Needs only efficient, periodic checks
HAIL:

Checking a row of blocks
HAIL:
Checking a row of blocks

\[ M_1 \quad M_2 \quad M_3 \quad P_1 \quad P_2 \]

\[ = \text{ECC}_1(M_1, M_2, M_3) \]

\[ = \text{ECC}_2(M_1, M_2, M_3) \]
HAIL:

Checking a row of blocks

\[ M_1 \quad M_2 \quad \overbrace{M_3} \quad P_1 \quad P_2 \]

\[ = \text{ECC}_1(M_1, M_2, M_3) \]

\[ = \text{ECC}_2(M_1, M_2, M_3) \]

- Remember in original POR, we need to store check values \( c_1, c_2, c_3 \)

- In HAIL, Alice probes a whole row

- So row redundancy enables check: No need to store special check values!

- Encrypting parity chunks turns them into MACs…
HAIL: Putting it all together

- HAIL offers an abstraction of high-integrity file system
- Robust to Byzantine, mobile adversary
- Modest storage / server overhead, thanks to “free MACs”
- Minimal bandwidth checking, thanks to POR techniques, e.g., aggregation
Another challenge:
The physical layer
Another challenge:
The physical layer

- Amazon claims to store three distinct copies of my file for resilience. Can they prove it?
  - POS won’t do the trick, nor will downloading!
Another challenge: The physical layer

- Ideas?
The Pizza Oven Protocol
[BvDJOR ‘11]

Eeta Pizza Pi

“Six pizzas!”

Cheapskate Pizza
The Pizza Oven Protocol

Six pizzas!

Eeta Pizza Pi

“Six pizzas!”

Cheapskate Pizza
The Pizza Oven Protocol

Cheapskate now claims it can survive an oven failure! How can Eeta Pizza Pi verify without visiting???
The Pizza Oven Protocol

Suppose that:

- A pizza oven bakes one pizza at a time, and takes 10 minutes
- The Cheapskate truck takes 15 minutes to deliver to Eeta Pizza Pi

\[ T_1 - T_0 = 45 \text{ mins?} \]

Eeta Pizza Pi

Cheapskate Pizza
How to apply to physical storage

- Ovens = rotational hard drives
- Truck = network
- Verifier challenges prover to serve randomly selected blocks from file...