Secure content distribution using untrusted servers

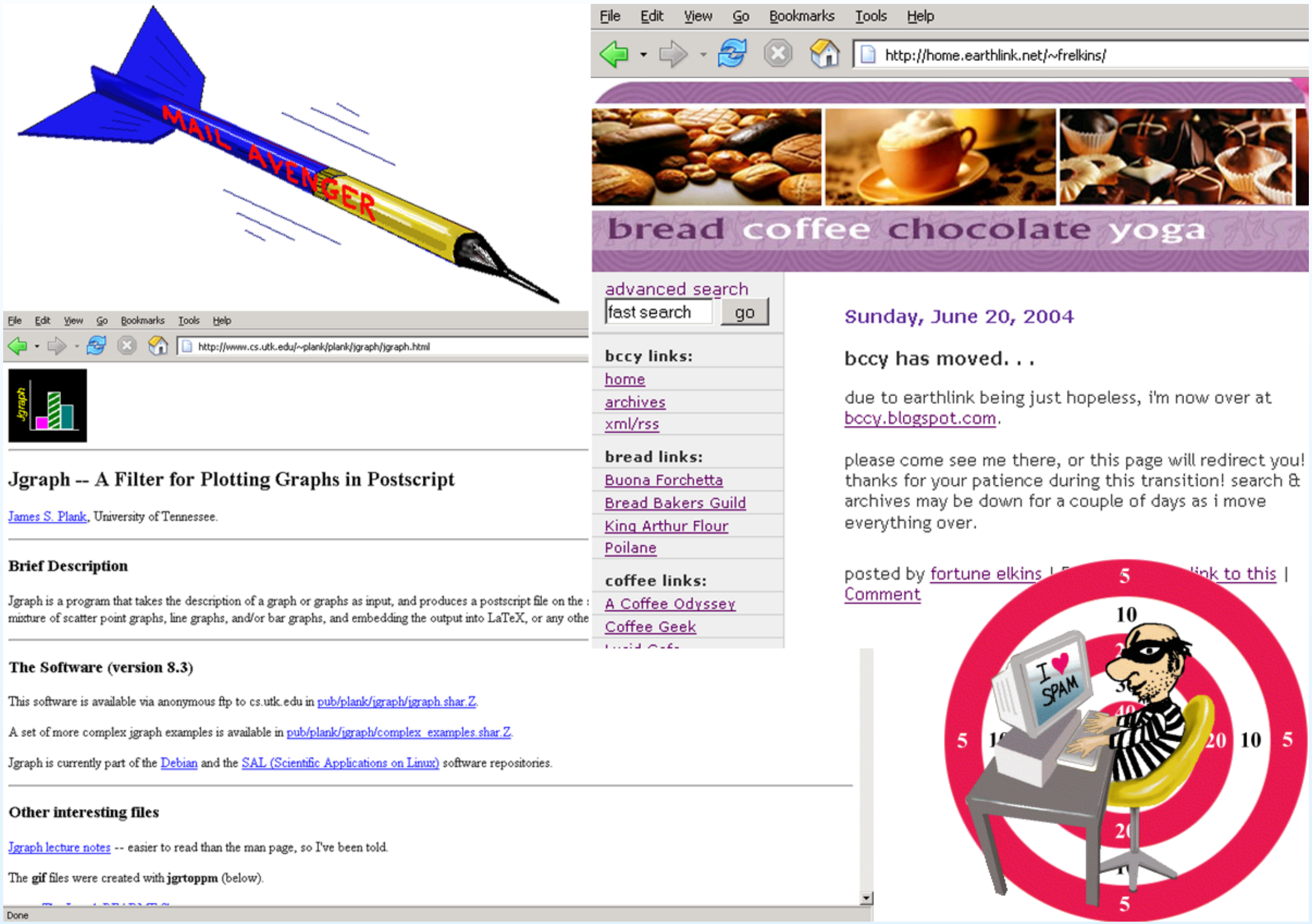
Kevin Fu

MIT Computer Science and Artificial Intelligence Lab

in collaboration with M. Frans Kaashoek (MIT),
Mahesh Kallahalla (DoCoMo Labs), Seny Kamara (JHU),
Yoshi Kohno (UCSD), David Mazières (NYU), Raj Rajagopalan (HP Labs),
Ron Rivest (MIT), Ram Swaminathan (HP Labs)

January-April 2005

How do we distribute content?



We pay services

Akamai

mirror image®

RSS cache.com

The bandwidth saver solution for your RSS feeds!

We coerce friends

We coerce friends

We enlist volunteers

BitTorrent

CoDeeN

CORAL

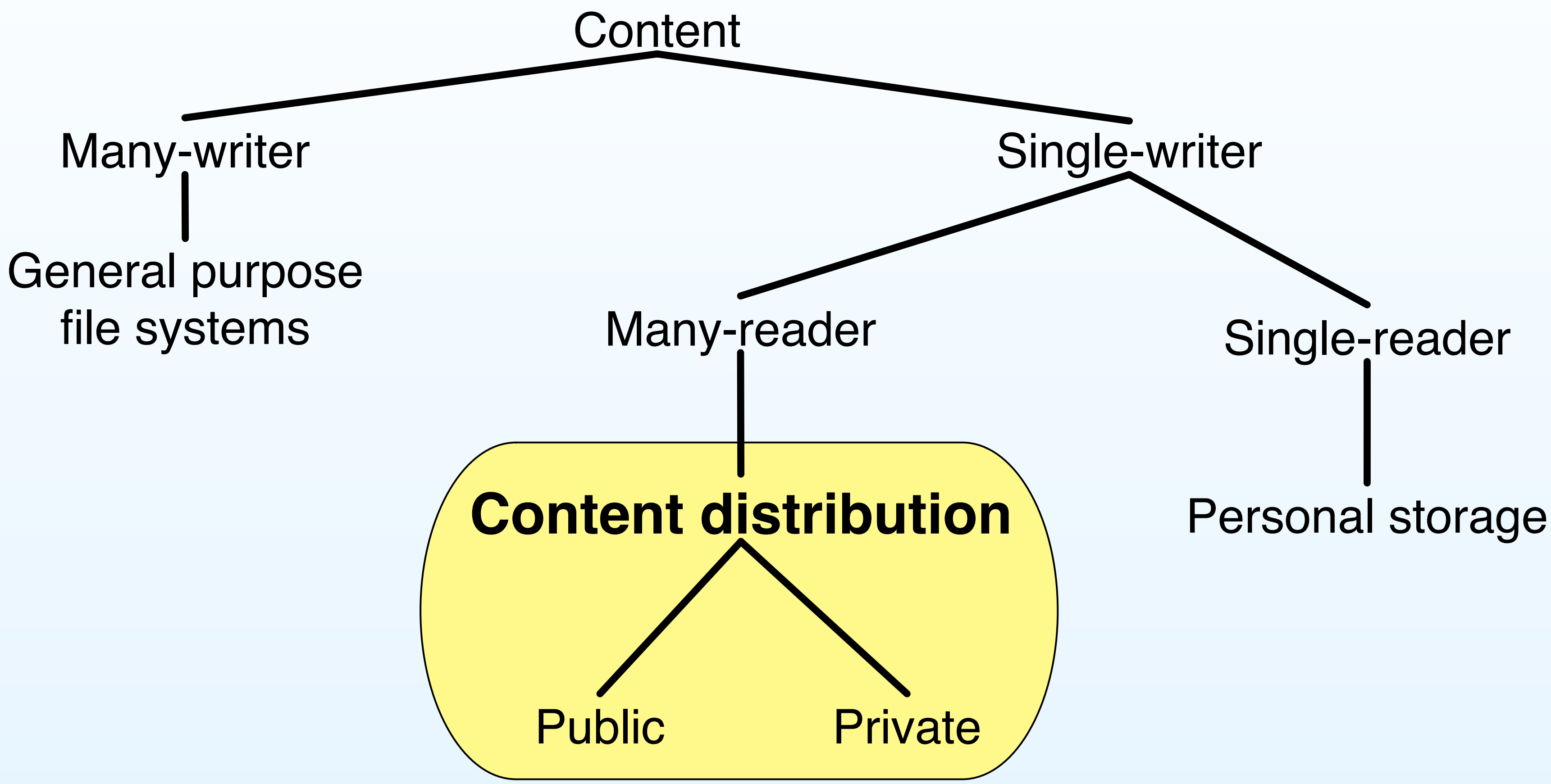
Fast content distribution, so what's left?

- Clients want
  - Authenticated content
  - Example: software updates, virus scanners
- Publishers want
  - Access control
  - Example: online newspapers

But what if

- Servers are untrusted
- Malicious parties control the network

Taxonomy of content



Framework

- Publishers ←write content, manage keys
- Clients ←read/verify content, trust publisher
- Untrusted servers ←replicate content
- File system ←protects data and metadata

Contributions

- Authenticated content distribution ←SFSRO
  - Self-certifying File System Read-Only
  - Public content distributed by untrusted servers
- Decentralized access control
  - Private content distributed by untrusted servers
  - Efficient client eviction
  - Efficient key distribution
- Implementation and performance measurements

Contributions

- Authenticated content distribution ←SFSRO
  - Self-certifying File System Read-Only
  - Public content distributed by untrusted servers
- Decentralized access control ←Chefs
  - Private content distributed by untrusted servers
  - Efficient client eviction
  - Efficient key distribution
- Implementation and performance measurements

Contributions

- Authenticated content distribution ←SFSRO
  - Self-certifying File System Read-Only
  - Public content distributed by untrusted servers
- Decentralized access control ←Chefs
  - Private content distributed by untrusted servers
  - Efficient client eviction ←Lazy revocation
  - Efficient key distribution ←Key regression
- Implementation and performance measurements

Contributions

- Authenticated content distribution ←SFSRO
  - Self-certifying File System Read-Only
  - Public content distributed by untrusted servers
- Decentralized access control ←Chefs
  - Private content distributed by untrusted servers
  - Efficient client eviction ←Lazy revocation
  - Efficient key distribution ←Key regression
- Implementation and performance measurements ←It works too!

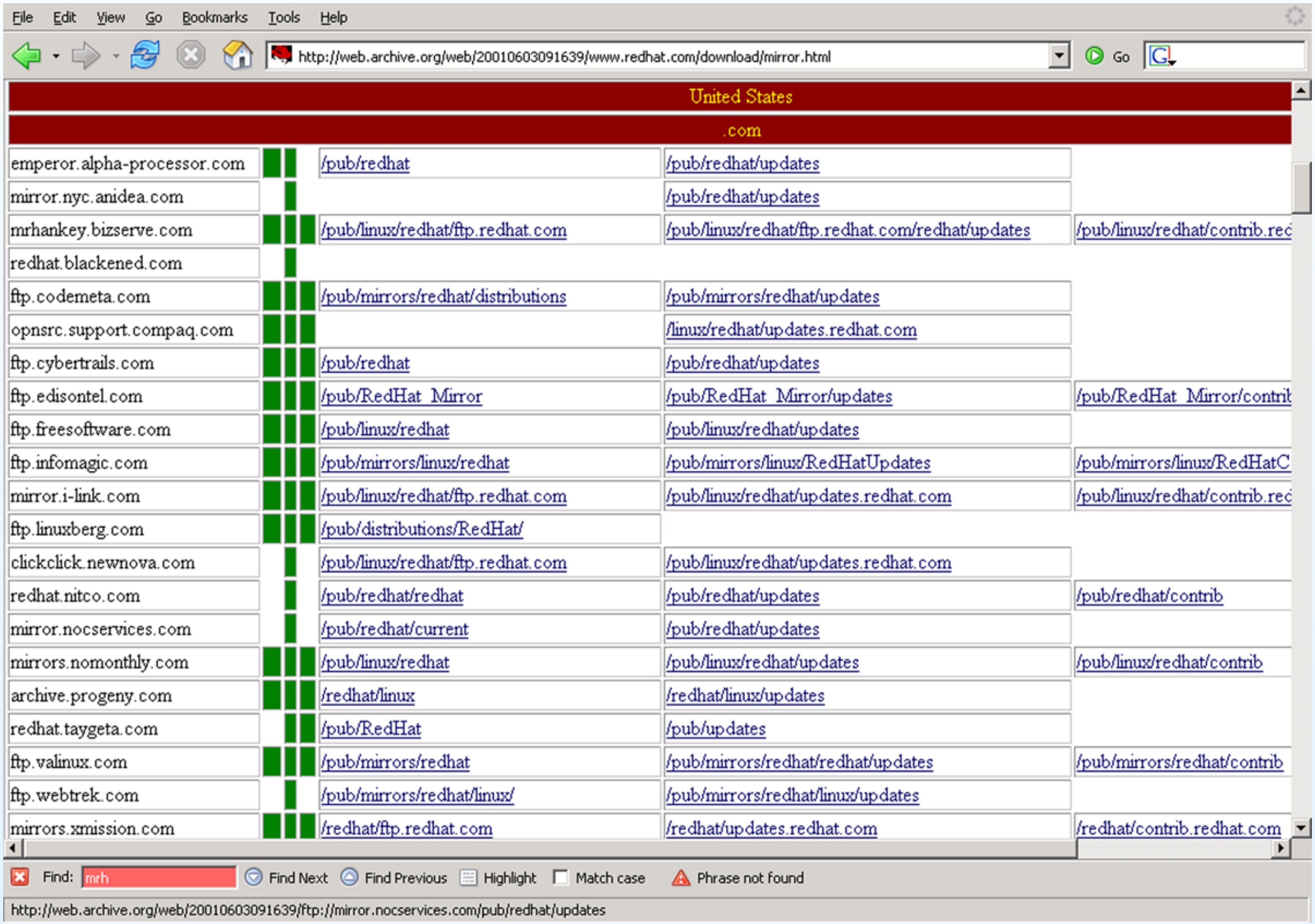
SFSRO

SFSRO challenges

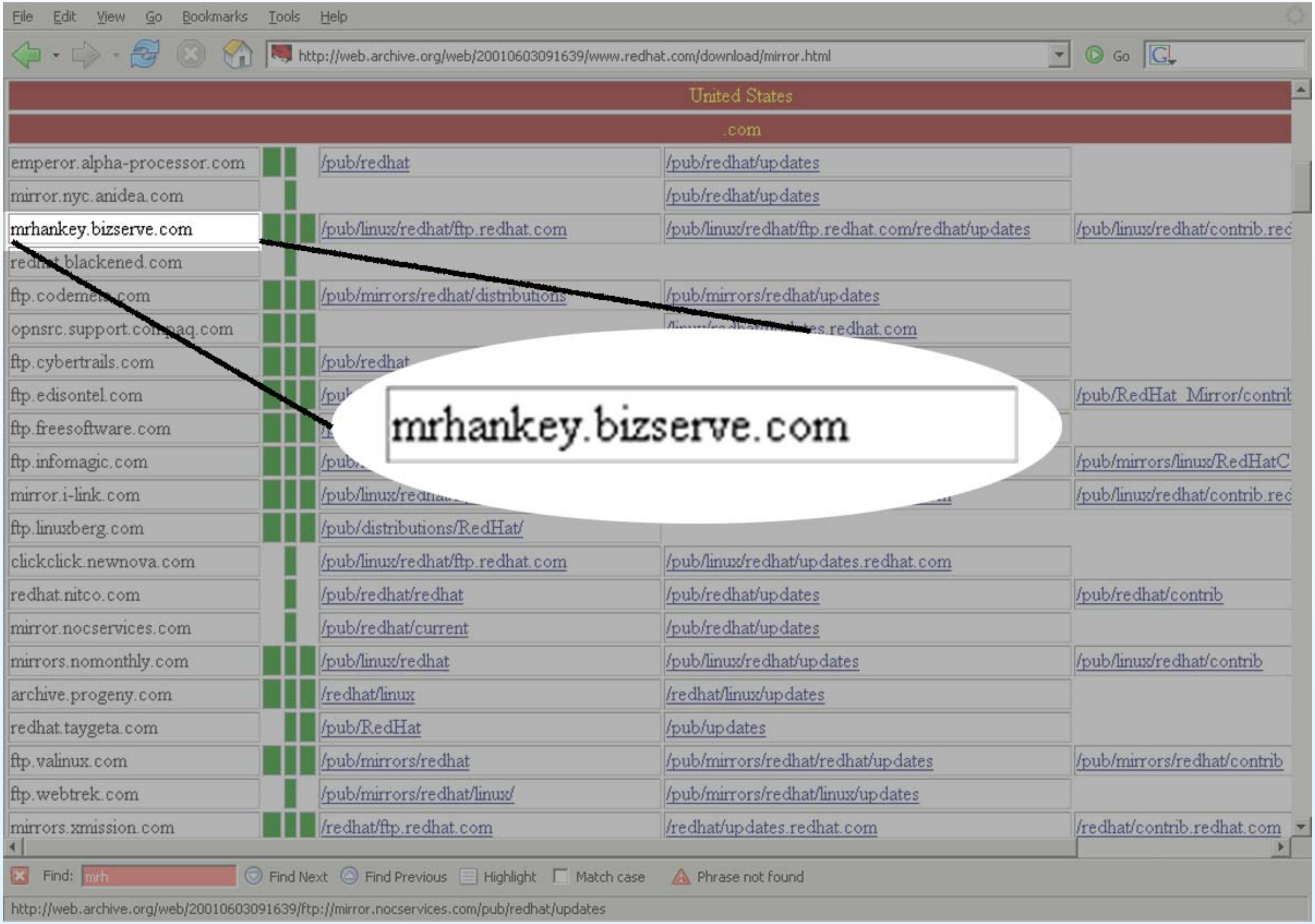
How can we authenticate content and also

- Provide incremental updates?
- Authenticate partial downloads?
- Scale servers to many clients?

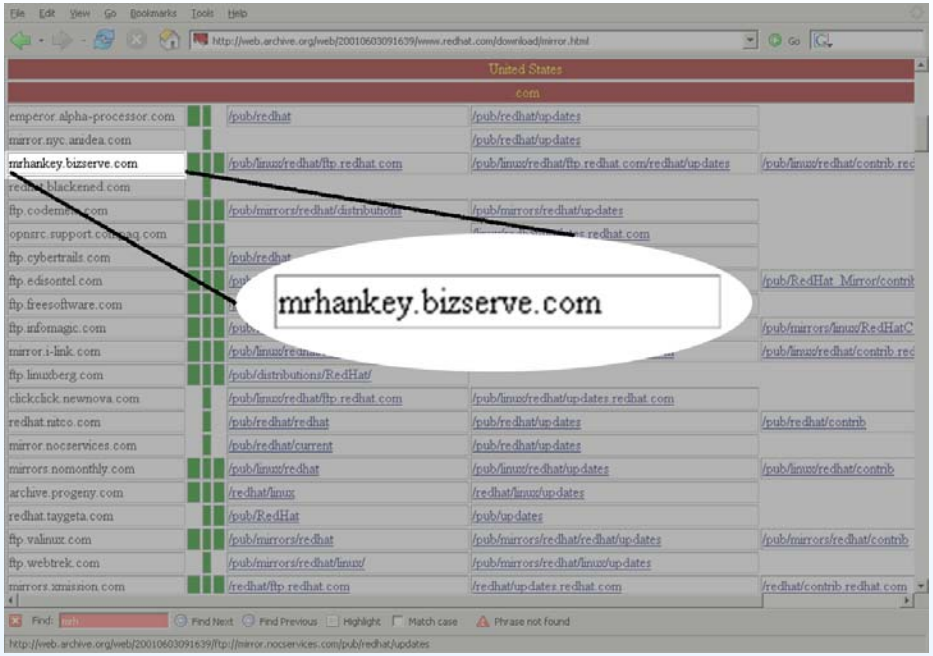
Signed software packages: part of your complete breakfast



Signed software packages: part of your complete breakfast

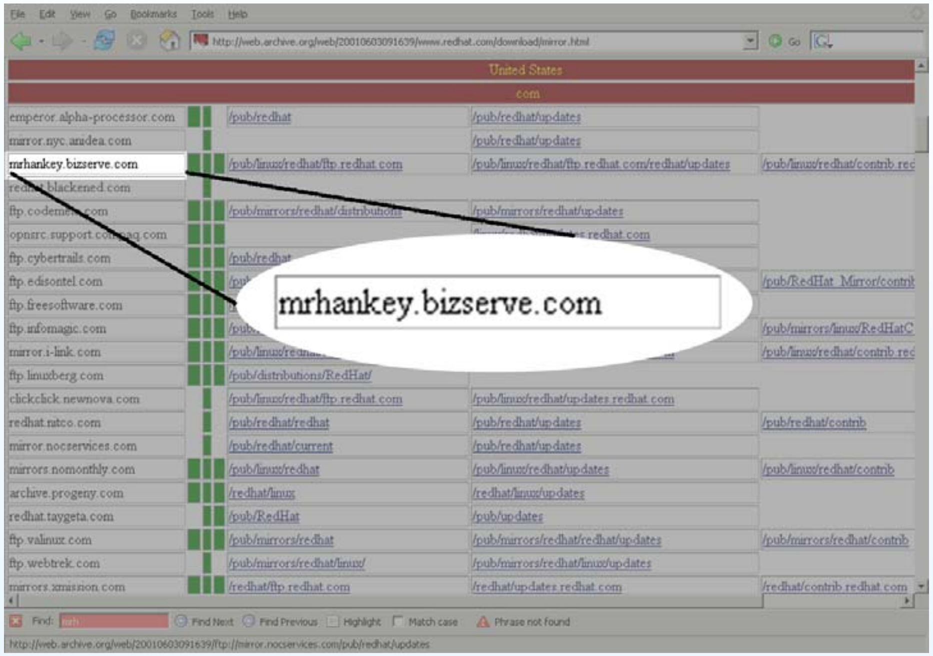


Signed software packages: part of your complete breakfast



- Authenticated

Signed software packages: part of your complete breakfast

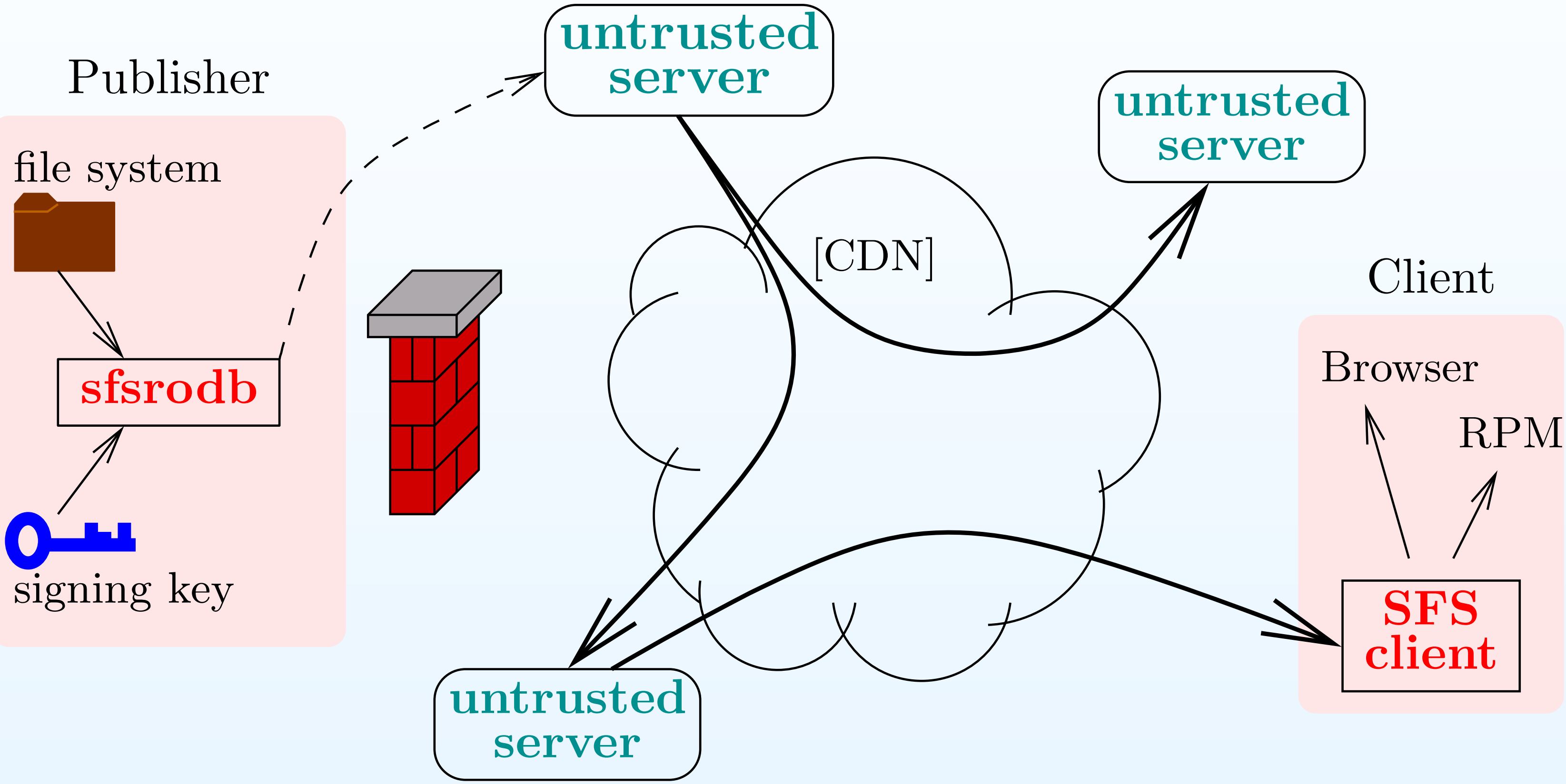


- Authenticated
- No revocation ✘
- No incremental updates ✘
- No integrity of file collections ✘

Is your collection of software authentic?

- Is the collection as a whole authentic? Rolled back?

SFSRO architecture

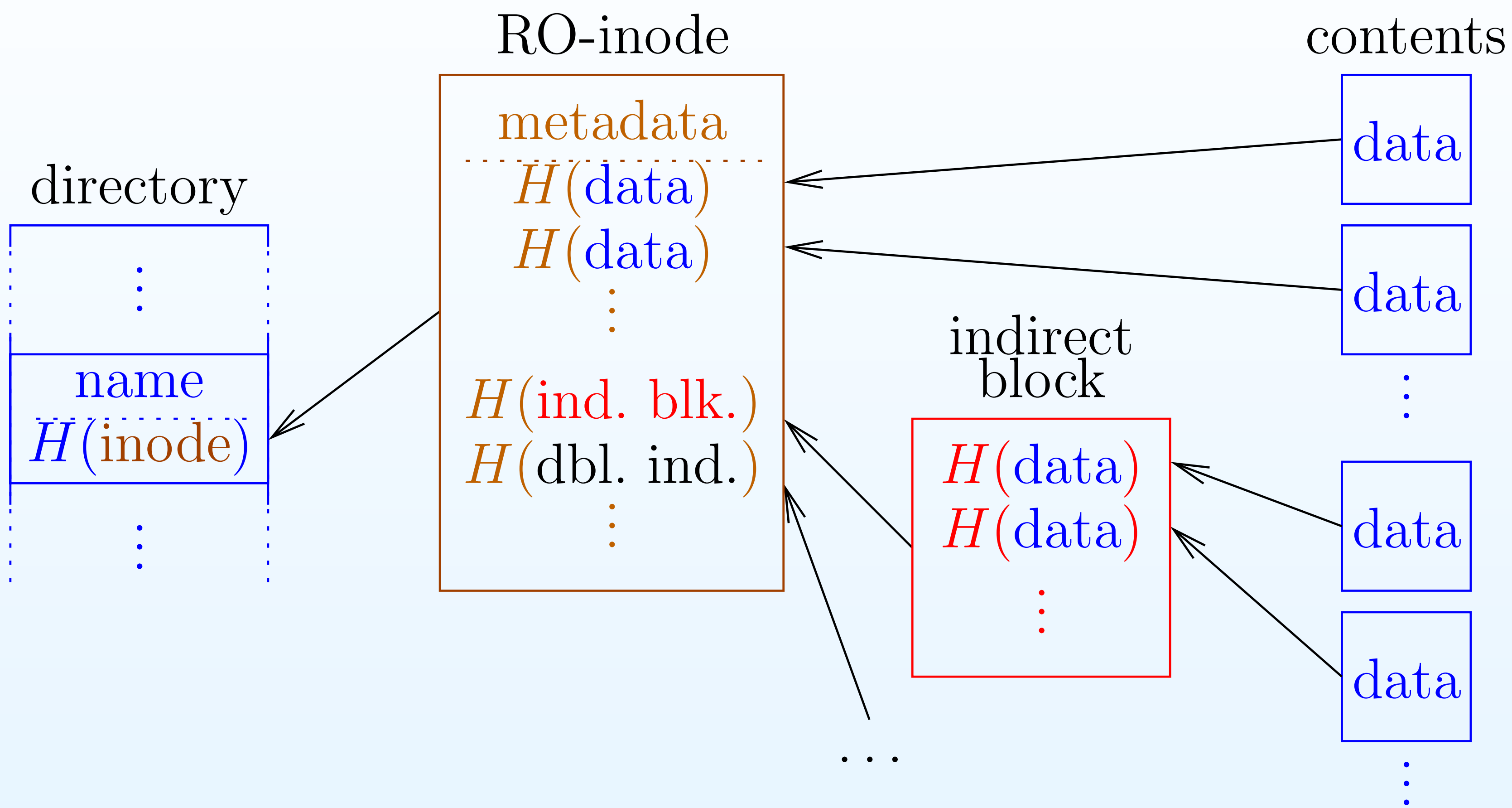


- SFSRO signs complete file system (data and metadata)
- Publisher stores files in replicated database ($\sim$ a disk image)
- Clients verify files without trusting servers

Authenticity via hash trees [Merkle:79]

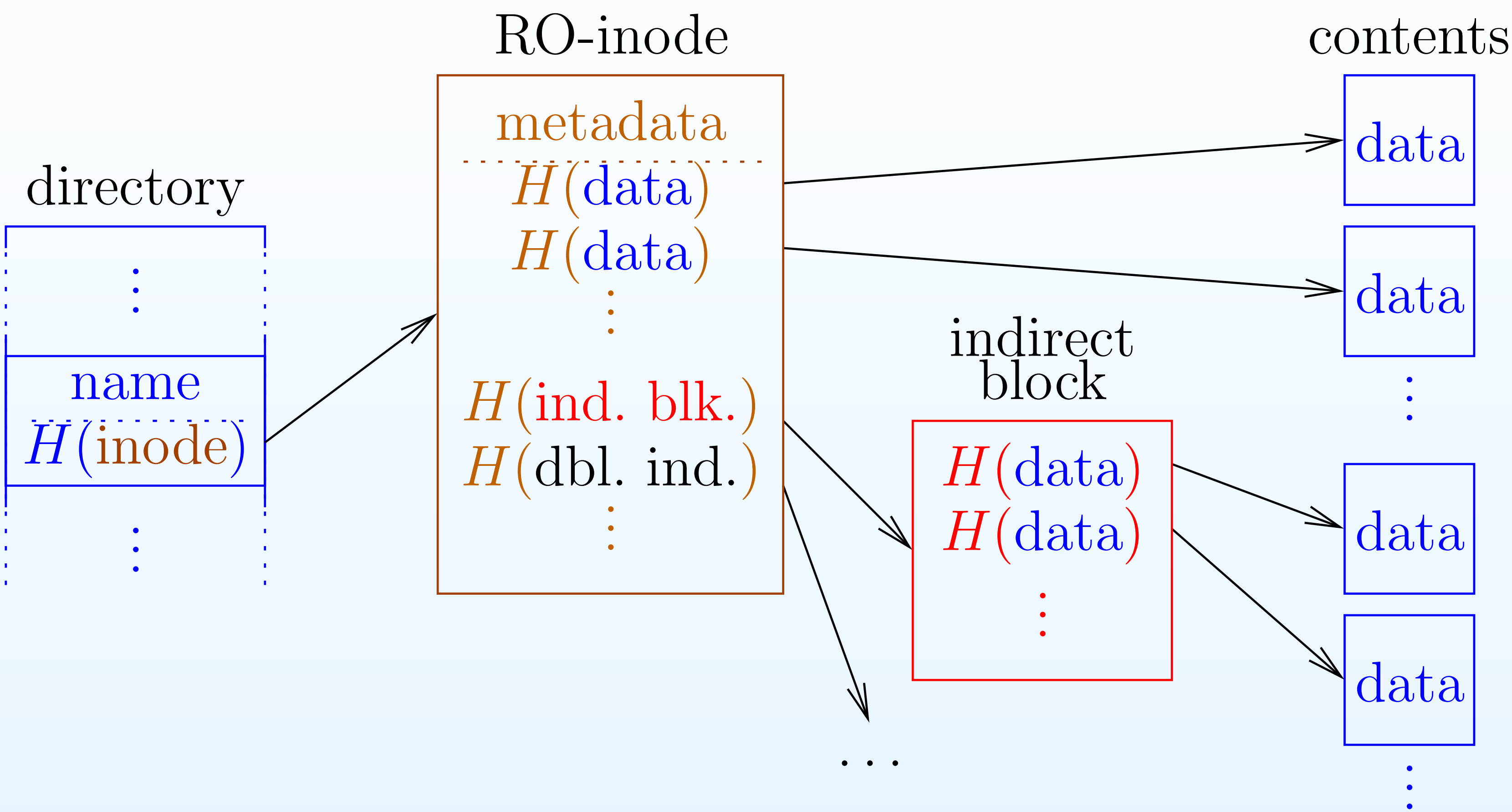
- Proves membership of a leaf in an n-node tree with O(log n) hashes
- Matches structure of a file system directory tree
- Ideal performance for incremental updates

Merkle hash tree mapped over directory tree



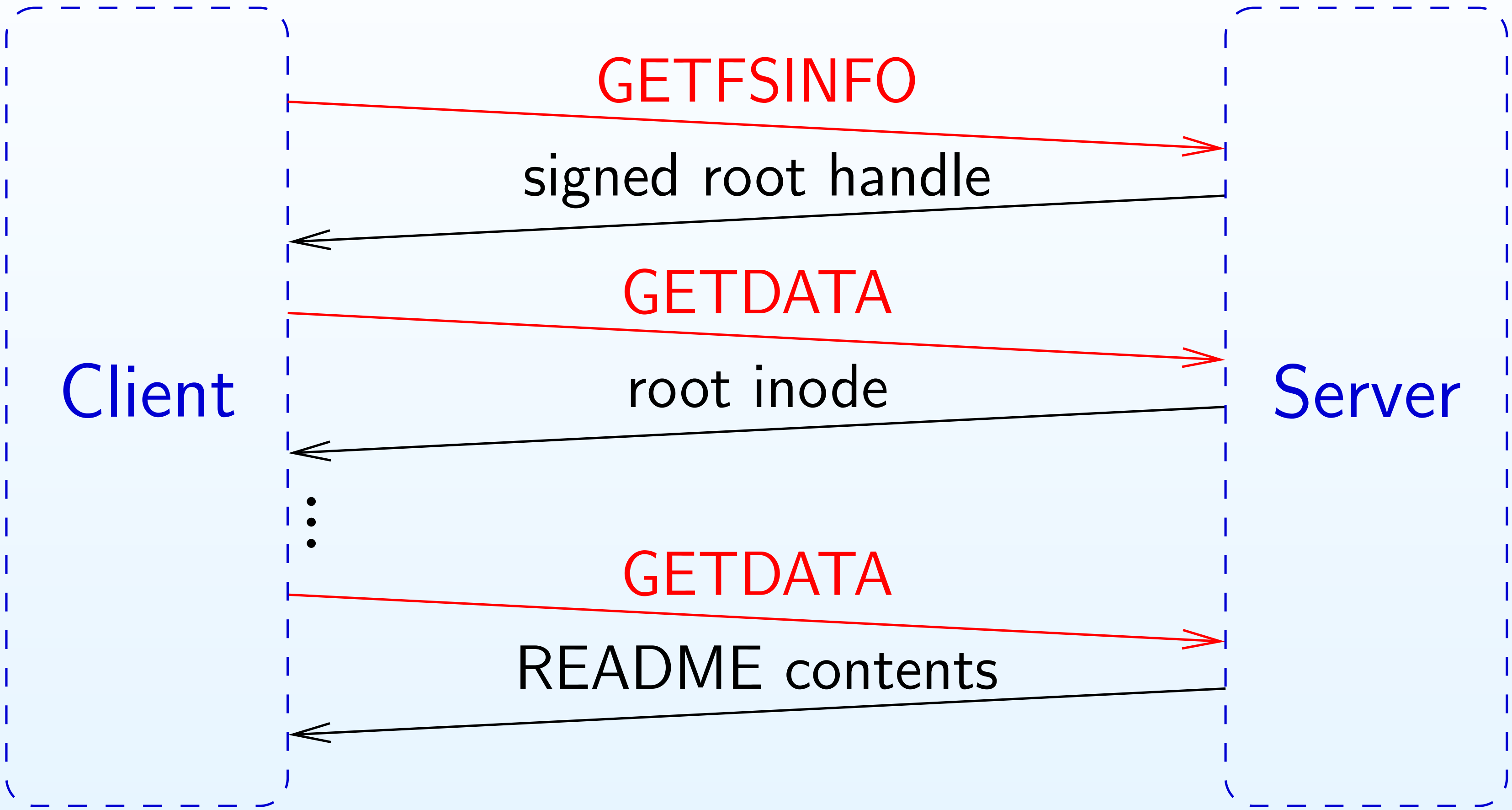
- Merkle hash tree of the file system [Haber:91, Devanbu:02]
- One public key operation and O(log n) hashes to authenticate

Merkle hash tree mapped over directory tree



- Merkle hash tree of the file system [Haber:91, Devanbu:02]
- One public key operation and O(log n) hashes to authenticate
- SFSRO protocol designed to walk Merkle trees

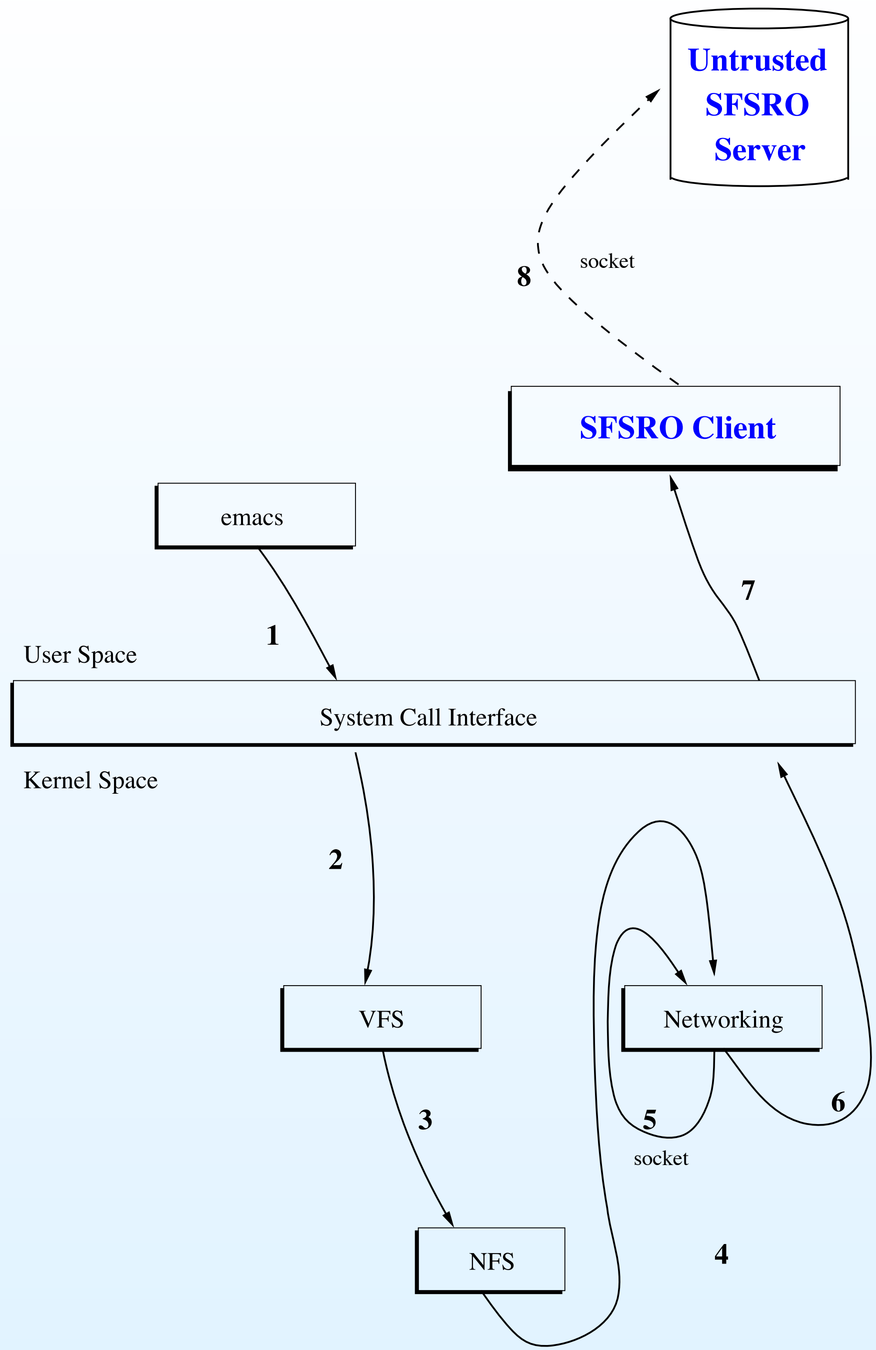
Example of reading `/shome/sfs.fs.net/README`



- SFSRO servers perform no online cryptography

Implementation of SFSRO

- Publisher
  - SHA-1 Merkle hash tree
  - Rabin-Williams signature
- Block server
  - Uses sleepycat database
  - Incremental updates
  - Influenced CFS [Dabek:01]
- Client
  - Transparent integrity checking
  - Implemented as NFS loopback



Chefs

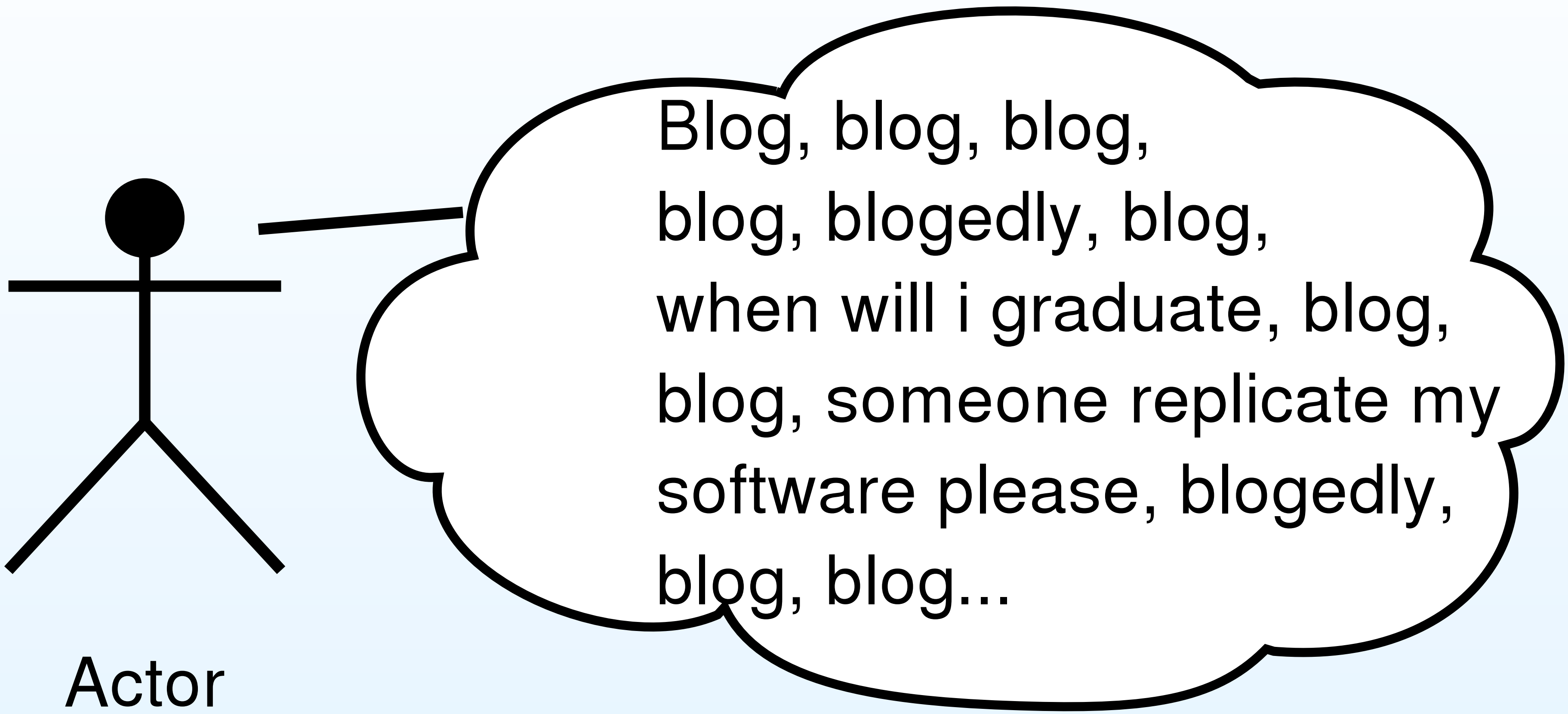
A brief timeline of SFS

- Read-write security in SFS [Mazieres:99]
- Read-only dialect [Fu:00] ←SFSRO
- Decentralized access control

A brief timeline of SFS

- Read-write security in SFS [Mazieres:99]
- Read-only dialect [Fu:00] ←SFSRO
- Decentralized access control ←Chefs
  - Servers remain untrusted
  - Clients with key can read content
  - Problem: Reduce key distribution

Access control using untrusted servers



A private blog

Potential approaches

- Proxy SSL Web server? [Laas:03]
  - Untrusted servers cannot replicate confidential content ✘

Potential approaches

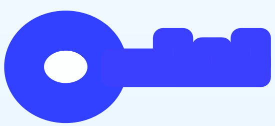
- Proxy SSL Web server? [Laas:03]
  - Untrusted servers cannot replicate confidential content ✘
- File encryption (e.g., PGP [Zimmermann:91])
  - Access controlled
  - Not transparent ✘
  - Ciphertext linear in number of clients ✘
  - No incremental updates ✘

Chefs approach extends SFSRO

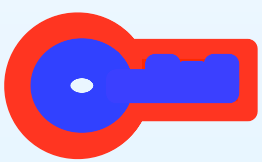
- Content encrypted for confidentiality [Swallow:81, Blaze:93, Waldman:00] ←decentralized access control
- Efficient client eviction ←lazy revocation
- Efficient key distribution ←key regression

Decentralized access control

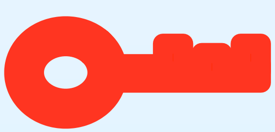
- Clients download content encrypted with content keys
- Encrypted content tagged with lockbox
- Open lockbox with the group key



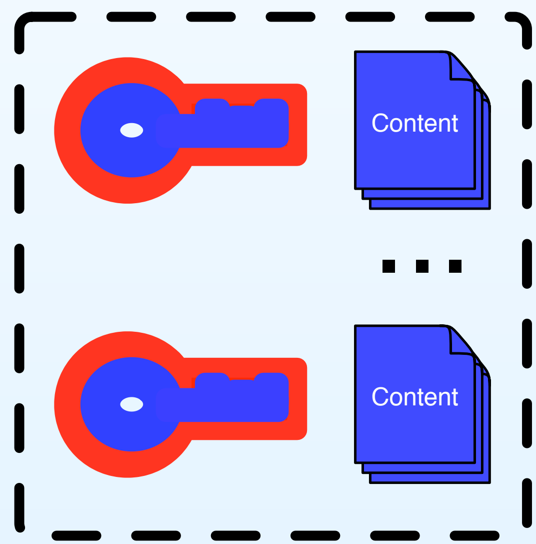
Content keys protect blocks



A lockbox contains a content key



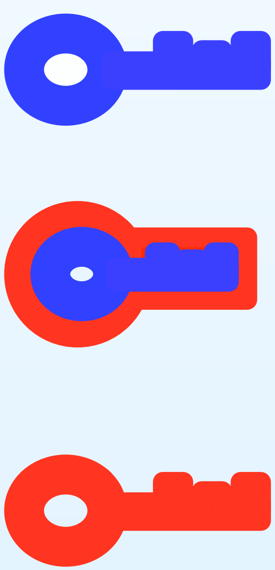
Group key opens lockboxes



Database of encrypted content + name of group key

Decentralized access control

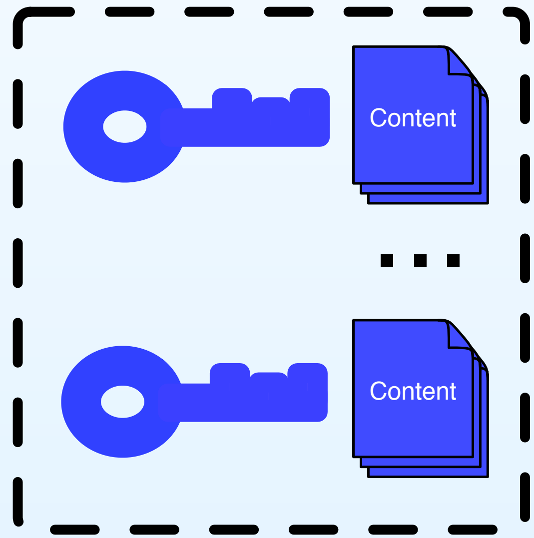
- Clients download content encrypted with content keys
- Encrypted content tagged with lockbox
- Open lockbox with the group key



Content keys protect blocks

A lockbox contains a content key

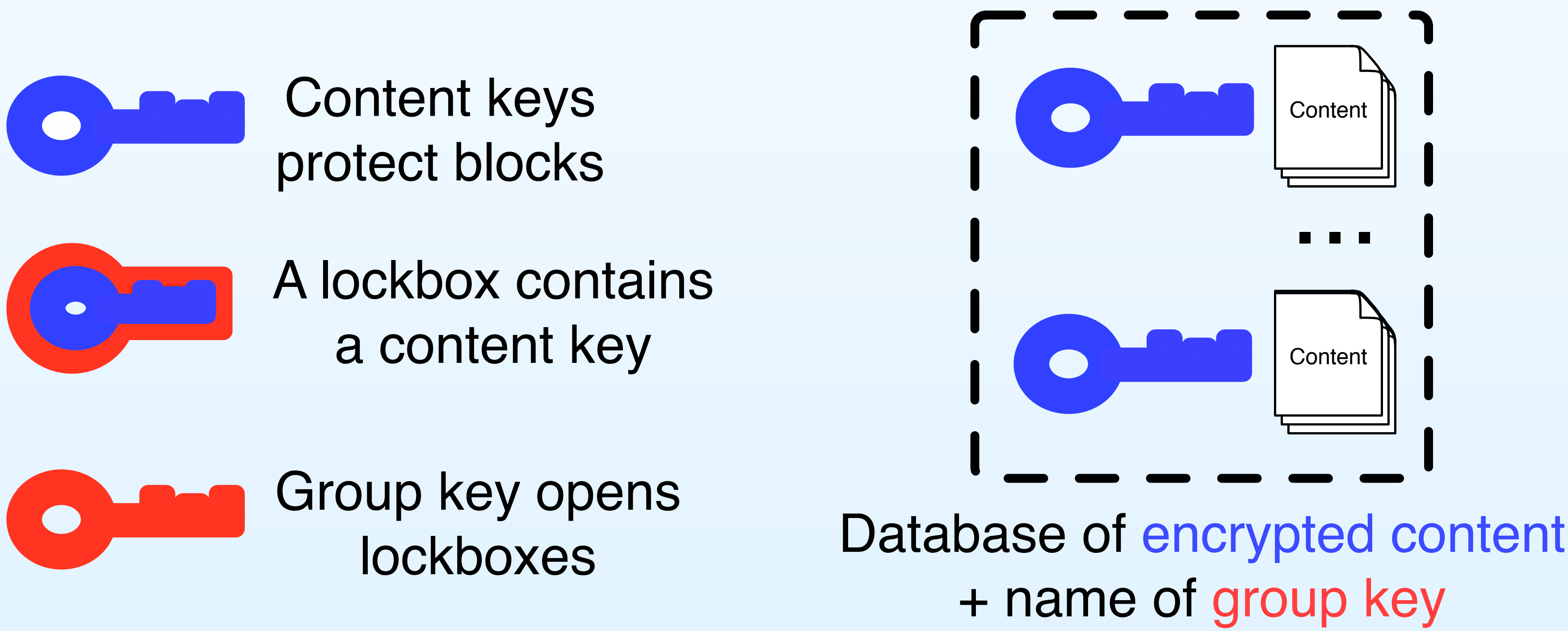
Group key opens lockboxes



Database of encrypted content + name of group key

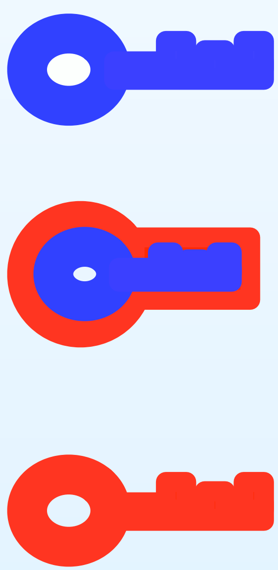
Decentralized access control

- Clients download content encrypted with content keys
- Encrypted content tagged with lockbox
- Open lockbox with the group key



Decentralized access control

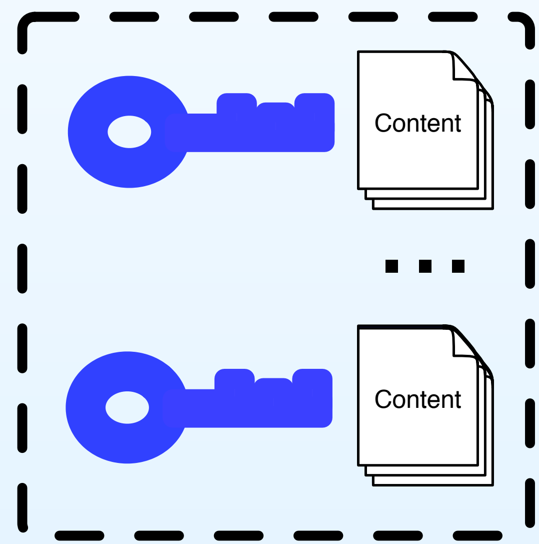
- Clients download content encrypted with content keys
- Encrypted content tagged with lockbox
- Open lockbox with the group key
- No key distribution required to add new content!



Content keys protect blocks

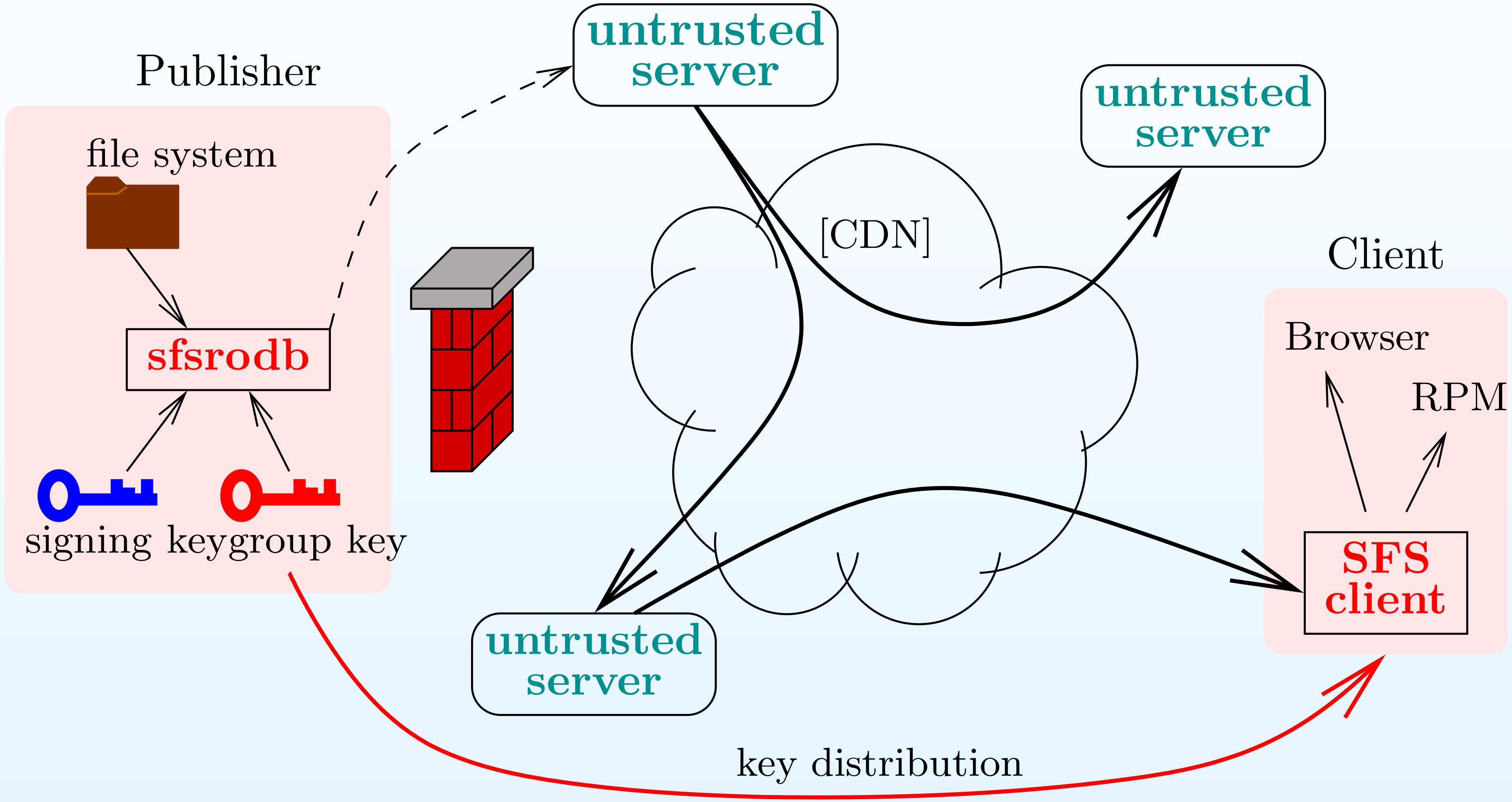
A lockbox contains a content key

Group key opens lockboxes



Database of encrypted content + name of group key

Overview of Chefs



Chefs = SFSRO + access control

Costly approach to coping with eviction

- Re-encrypt content after eviction
- Distribute new key to remaining clients

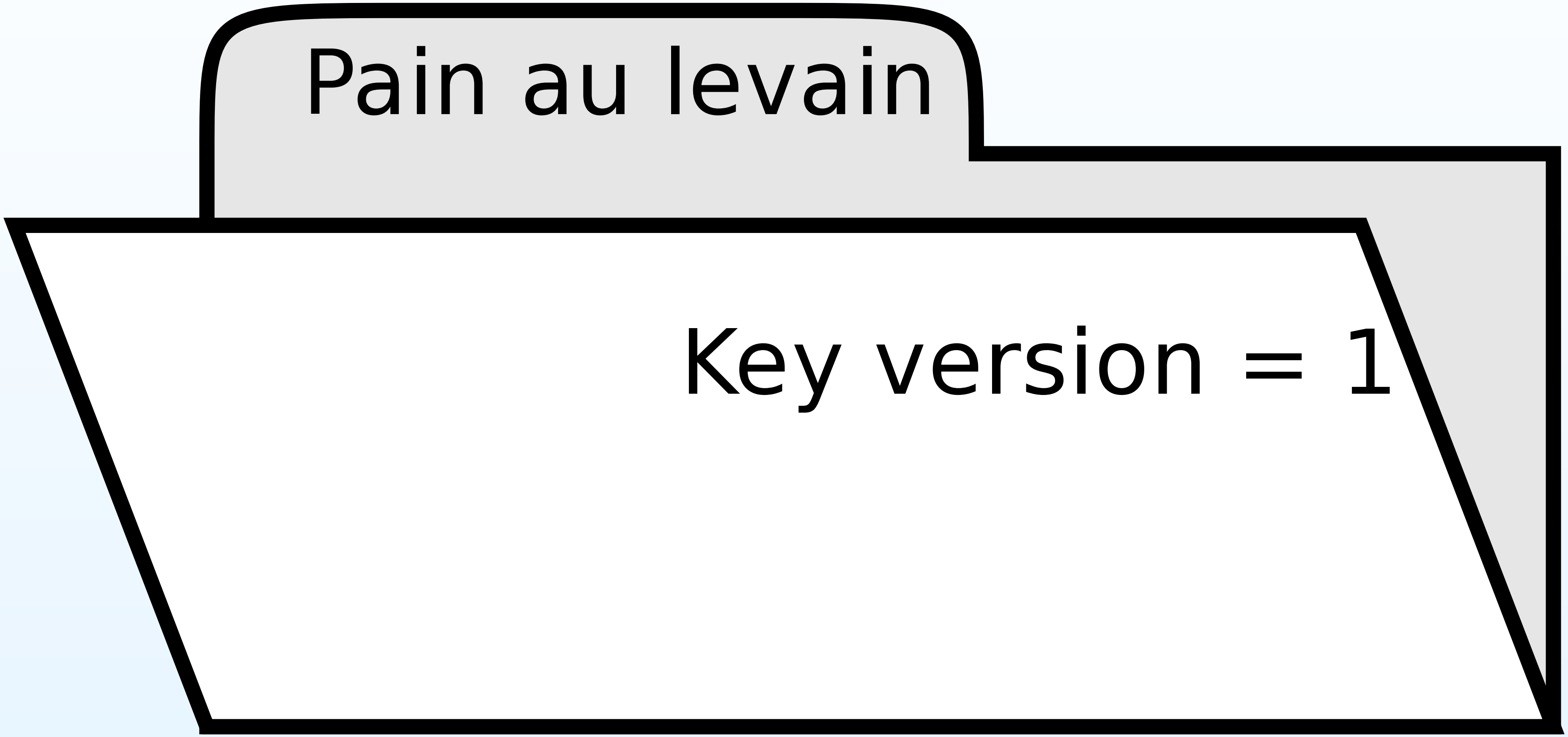
Costly approach to coping with eviction

- Re-encrypt content after eviction ←Unnecessary
- Distribute new key to remaining clients

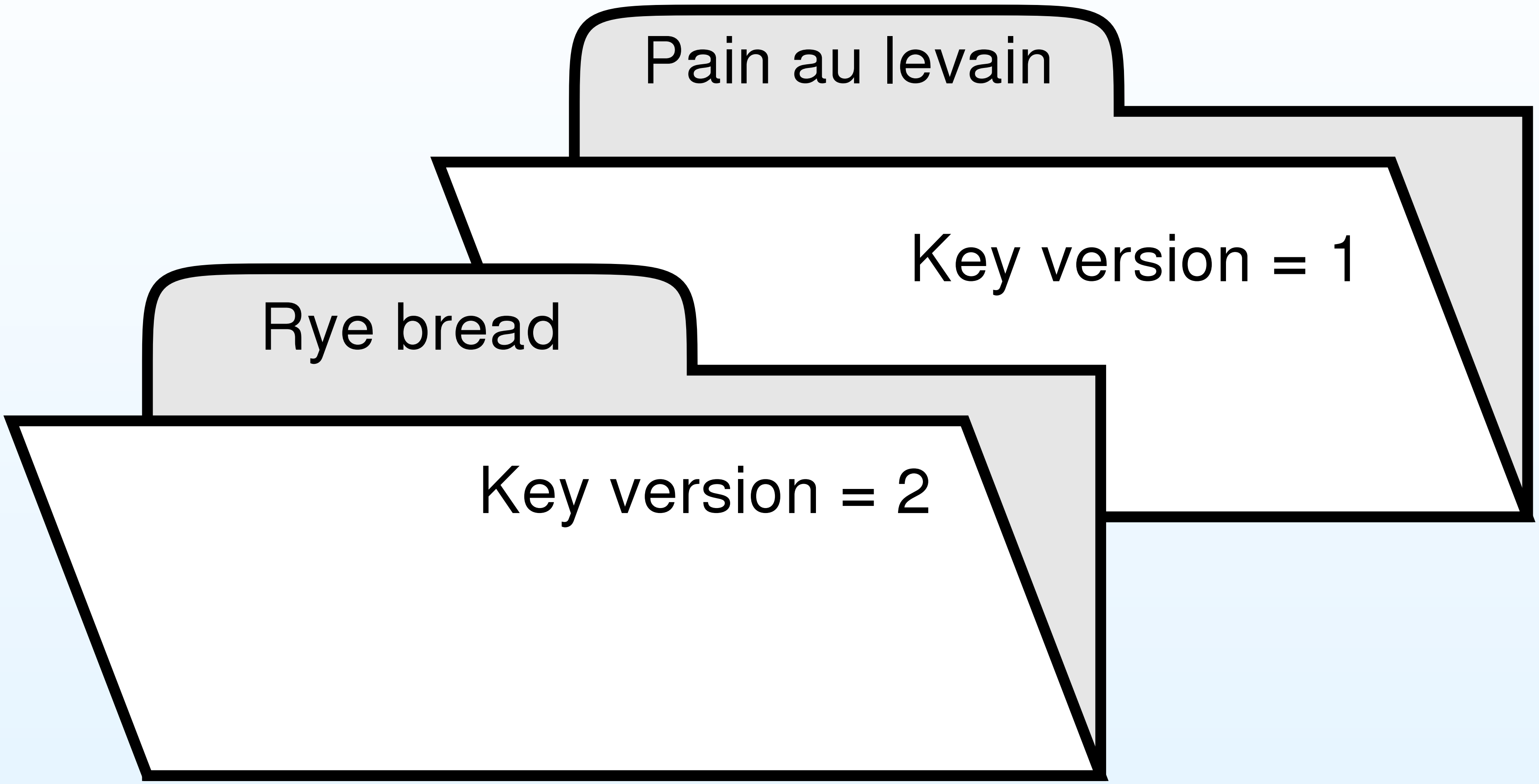
Chefs solution: lazy revocation

- Guarantees evicted client cannot access new content
- After eviction, generate a new key for future updates
- Matches semantics of untrusted storage

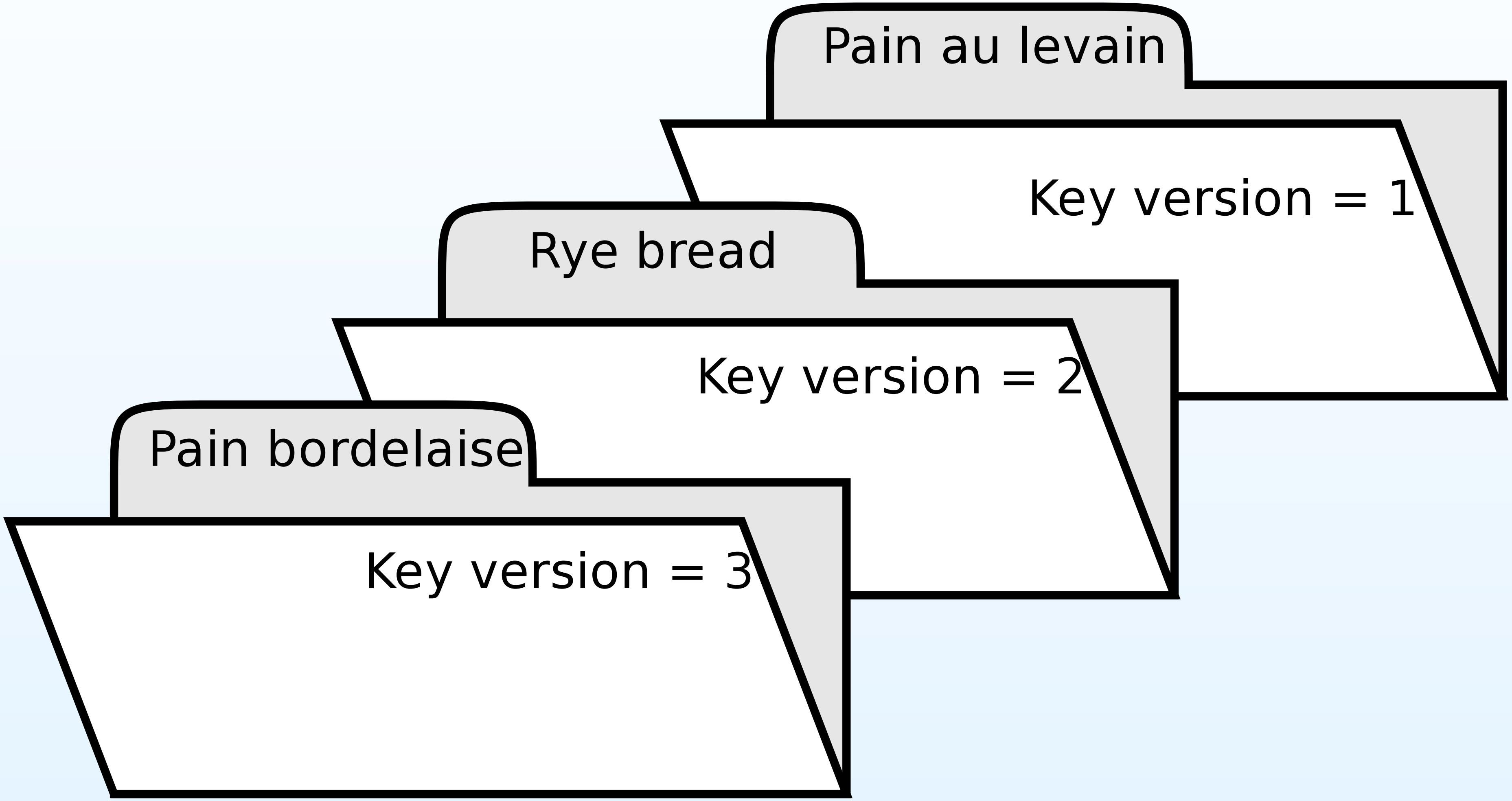
Lazy revocation results in many keys



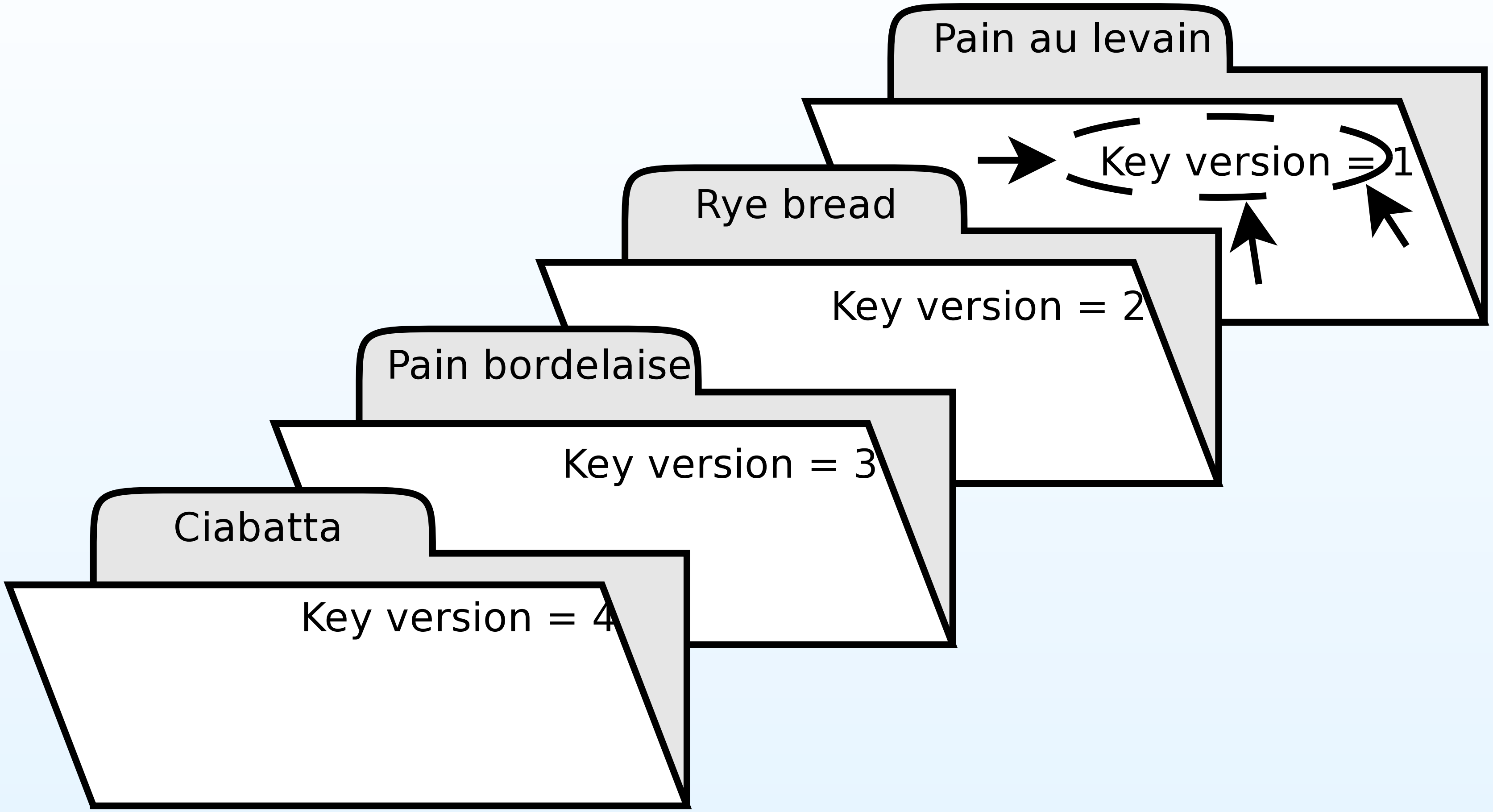
Lazy revocation results in many keys



Lazy revocation results in many keys



Lazy revocation results in many keys



- How can a client coalesce group key versions?

Key regression: coping with many keys

- Guarantees clients
  - Can access old content
  - Cannot yet access future content
- Clients derive past keys from current key
- Low-bandwidth publishers make new keys available

Downloading all the keys can be costly

- Searching encrypted content
  - Client must perform search, not untrusted server
  - Client downloads all encrypted recipes and keys
- Scenarios
  - 60,000 membership events/year on Salon.com online journal
  - Offline publisher

What does "secure" key regression mean?

- Only clients can unwind keys
  - $K_i = \text{unwind}(K_{i+1})$
- Only publisher can wind key forward
  - $K_{i+1} = \text{wind}(K_i)$
- Should behave like randomly selected keys

Simplest way to use key regression

- Publisher initialization:
  - Generate a random K_{t-1}

Publisher:

K_{t-1}

Simplest way to use key regression

- Publisher initialization:
  - Generate a random K_{t-1}
  - Compute $K_0, \ldots, K_{t-2}$ by unwinding

Publisher:

K_{t-1}

Simplest way to use key regression

- Publisher initialization:
  - Generate a random K_{t-1}
  - Compute $K_0, \ldots, K_{t-2}$ by unwinding

Publisher:

$$K_{t-2} \xleftarrow{U(K_{t-1})} K_{t-1}$$

Simplest way to use key regression

- Publisher initialization:
  - Generate a random K_{t-1}
  - Compute $K_0, \ldots, K_{t-2}$ by unwinding

Publisher:

$$\cdots \xleftarrow{U(K_{t-2})} K_{t-2} \xleftarrow{U(K_{t-1})} K_{t-1}$$

Simplest way to use key regression

- Publisher initialization:
  - Generate a random K_{t-1}
  - Compute $K_0, \ldots, K_{t-2}$ by unwinding

Publisher:

$$K_1 \xleftarrow{U(K_2)} \cdots \xleftarrow{U(K_{t-2})} K_{t-2} \xleftarrow{U(K_{t-1})} K_{t-1}$$

Simplest way to use key regression

- Publisher initialization:
  - Generate a random K_{t-1}
  - Compute $K_0, \ldots, K_{t-2}$ by unwinding

Publisher:

$$K_0 \xleftarrow{U(K_1)} K_1 \xleftarrow{U(K_2)} \cdots \xleftarrow{U(K_{t-2)}} K_{t-2} \xleftarrow{U(K_{t-1})} K_{t-1}$$

Simplest way to use key regression

- Publisher initialization:
  - Generate a random K_{t-1}
  - Compute $K_0, \ldots, K_{t-2}$ by unwinding
  - Distribute K_0 to clients

Publisher:

$$K_0 \xleftarrow{U(K_1)} K_1 \xleftarrow{U(K_2)} \cdots \xleftarrow{U(K_{t-2})} K_{t-2} \xleftarrow{U(K_{t-1})} K_{t-1}$$

Simplest way to use key regression

- Publisher initialization:
  - Generate a random K_{t-1}
  - Compute $K_0, \ldots, K_{t-2}$ by unwinding
  - Distribute K_0 to clients

- Client joining at time i
  - Receive K_i from publisher
  - To read content encrypted with K_j for $j < i$, unwind K_i

Publisher:

$$K_0 \xleftarrow{U(K_1)} K_1 \xleftarrow{U(K_2)} \cdots \xleftarrow{U(K_{t-2})} K_{t-2} \xleftarrow{U(K_{t-1})} K_{t-1}$$

Client:

$$K_i$$

Simplest way to use key regression

- Publisher initialization:
  - Generate a random K_{t-1}
  - Compute $K_0, \ldots, K_{t-2}$ by unwinding
  - Distribute K_0 to clients

- Client joining at time i
  - Receive K_i from publisher
  - To read content encrypted with K_j for $j < i$, unwind K_i

Publisher:

$$K_0 \xleftarrow{U(K_1)} K_1 \xleftarrow{U(K_2)} \cdots \xleftarrow{U(K_{t-2})} K_{t-2} \xleftarrow{U(K_{t-1})} K_{t-1}$$

Client:

$$K_{i-1} \xleftarrow{U(K_i)} K_i$$

Simplest way to use key regression

- Publisher initialization:
  - Generate a random K_{t-1}
  - Compute $K_0, \ldots, K_{t-2}$ by unwinding
  - Distribute K_0 to clients

- Client joining at time i
  - Receive K_i from publisher
  - To read content encrypted with K_j for $j < i$, unwind K_i

Publisher:

$$K_0 \xleftarrow{U(K_1)} K_1 \xleftarrow{U(K_2)} \cdots \xleftarrow{U(K_{t-2})} K_{t-2} \xleftarrow{U(K_{t-1})} K_{t-1}$$

Client:

$$\cdots \xleftarrow{U(K_{i-1})} K_{i-1} \xleftarrow{U(K_i)} K_i$$

Simplest way to use key regression

- Publisher initialization:
  - Generate a random K_{t-1}
  - Compute $K_0, \ldots, K_{t-2}$ by unwinding
  - Distribute K_0 to clients

- Client joining at time i
  - Receive K_i from publisher
  - To read content encrypted with K_j for $j < i$, unwind K_i

Publisher:

$$K_0 \xleftarrow{U(K_1)} K_1 \xleftarrow{U(K_2)} \cdots \xleftarrow{U(K_{t-2})} K_{t-2} \xleftarrow{U(K_{t-1})} K_{t-1}$$

Client:

$$K_j \xleftarrow{U(K_{j+1})} \cdots \xleftarrow{U(K_{i-1})} K_{i-1} \xleftarrow{U(K_i)} K_i$$

Simplest way to use key regression

- Publisher initialization:
  - Generate a random K_{t-1}
  - Compute $K_0, \ldots, K_{t-2}$ by unwinding
  - Distribute K_0 to clients

- Client joining at time i
  - Receive K_i from publisher
  - To read content encrypted with K_j for $j < i$, unwind K_i
  - Decrypt content with K_j

Publisher:

$$K_0 \xleftarrow{U(K_1)} K_1 \xleftarrow{U(K_2)} \cdots \xleftarrow{U(K_{t-2})} K_{t-2} \xleftarrow{U(K_{t-1})} K_{t-1}$$

Client:

$$K_j \xleftarrow{U(K_{j+1})} \cdots \xleftarrow{U(K_{i-1})} K_{i-1} \xleftarrow{U(K_i)} K_i$$

Key regression produces a key sequence

$K_i = \mathsf{H}(K_{i+1})$

$$\underbrace{K_0 \xleftarrow{H(K_1)} K_1 \xleftarrow{H(K_2)} \cdots \xleftarrow{H(K_{t-1})} K_{t-1}}_{\text{group key sequence}}$$

where H could be

In practice: SHA-1 $(\cdot)$ hash function

In theory: PRF $F(\cdot)$ in random oracle model

[Lamport:81], [Anderson:97]

An extension to key regression

- Dynamically grow a key sequence
  - Sequence length not determined a priori
  - Use a trapdoor pseudorandom *permutation*

RSA-based key regression: mechanics

- Publisher winds keys forward to grow a sequence:

$K_{i+1} = K_i^d \bmod N$

$$K_0$$

- Client unwinds keys:

$K_{i-1} = K_i^e \bmod N$

RSA-based key regression: mechanics

- Publisher winds keys forward to grow a sequence:

$K_{i+1} = K_i^d \text{ mod } N$

$$K_0 \xrightarrow{K_0^d \text{ mod } N} K_1$$

- Client unwinds keys:

$K_{i-1} = K_i^e \text{ mod } N$

RSA-based key regression: mechanics

- Publisher winds keys forward to grow a sequence:

  $K_{i+1} = K_i^d \text{ mod } N$

$$K_0 \xrightarrow{K_0^d \text{ mod } N} K_1 \xrightarrow{K_1^d \text{ mod } N} \cdots$$

- Client unwinds keys:

  $K_{i-1} = K_i^e \text{ mod } N$

RSA-based key regression: mechanics

- Publisher winds keys forward to grow a sequence:

  $K_{i+1} = K_i^d \bmod N$

$$K_0 \xrightarrow{K_0^d \bmod N} K_1 \xrightarrow{K_1^d \bmod N} \cdots \xrightarrow{K_{t-2}^d \bmod N} K_{t-1}$$

- Client unwinds keys:

  $K_{i-1} = K_i^e \bmod N$

RSA-based key regression: mechanics

- Publisher winds keys forward to grow a sequence:

  $K_{i+1} = K_i^d \text{ mod } N$

$$K_0 \xrightarrow{K_0^d \text{ mod } N} K_1 \xrightarrow{K_1^d \text{ mod } N} \cdots \xrightarrow{K_{t-2}^d \text{ mod } N} K_{t-1}$$

- Client unwinds keys:

  $K_{i-1} = K_i^e \text{ mod } N$

$$K_{t-1}$$

RSA-based key regression: mechanics

- Publisher winds keys forward to grow a sequence:

  $K_{i+1} = K_i^d \bmod N$

$$K_0 \xrightarrow{K_0^d \bmod N} K_1 \xrightarrow{K_1^d \bmod N} \cdots \xrightarrow{K_{t-2}^d \bmod N} K_{t-1}$$

- Client unwinds keys:

  $K_{i-1} = K_i^e \bmod N$

$$\cdots \xleftarrow{K_{t-1}^e \bmod N} K_{t-1}$$

RSA-based key regression: mechanics

- Publisher winds keys forward to grow a sequence:

  $K_{i+1} = K_i^d \bmod N$

$$K_0 \xrightarrow{K_0^d \bmod N} K_1 \xrightarrow{K_1^d \bmod N} \cdots \xrightarrow{K_{t-2}^d \bmod N} K_{t-1}$$

- Client unwinds keys:

  $K_{i-1} = K_i^e \bmod N$

$$\xleftarrow{K_2^e \bmod N} \cdots \xleftarrow{K_{t-1}^e \bmod N} K_{t-1}$$

RSA-based key regression: mechanics

- Publisher winds keys forward to grow a sequence:

  $K_{i+1} = K_i^d \text{ mod } N$

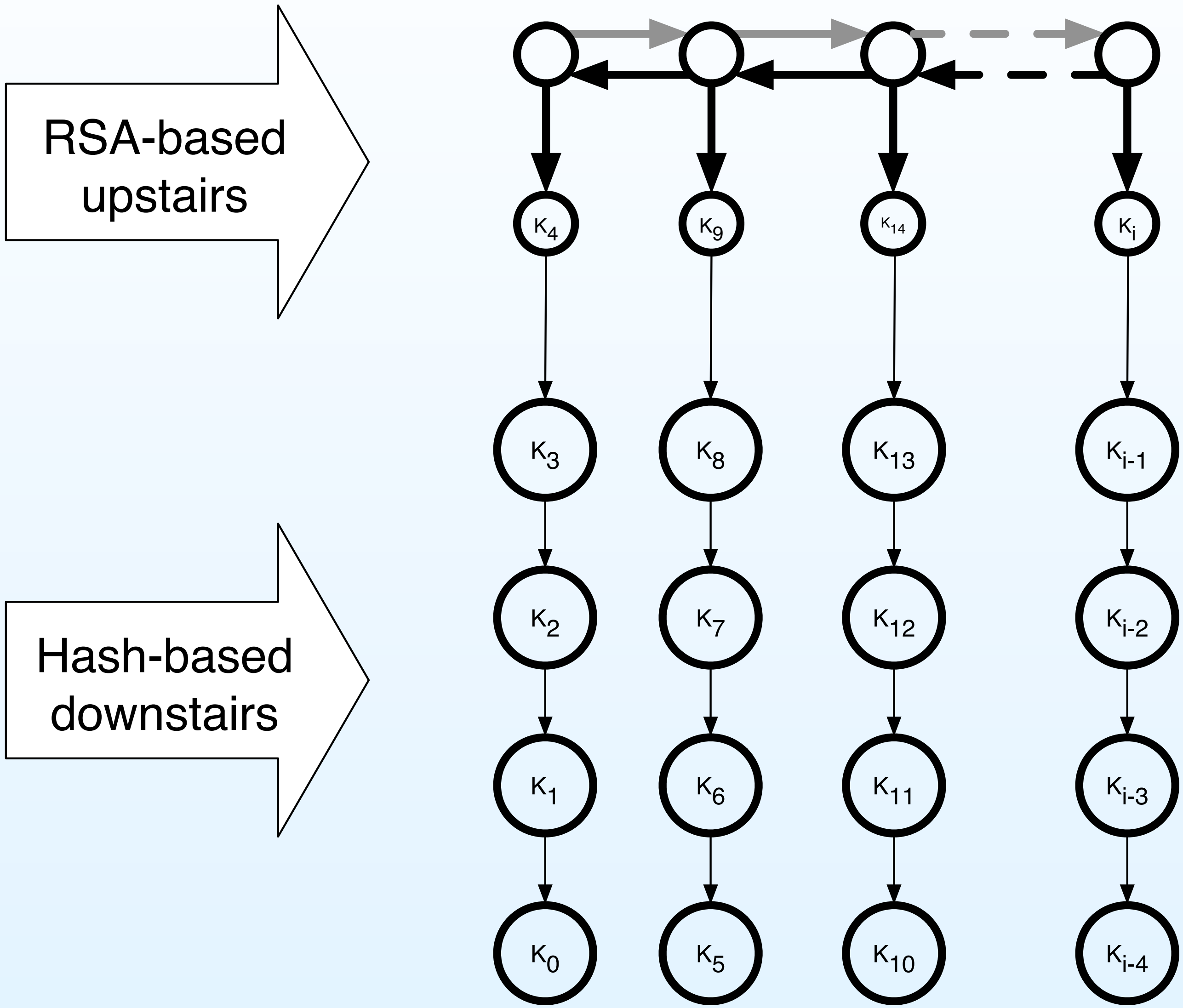
$$K_0 \xrightarrow{K_0^d \text{ mod } N} K_1 \xrightarrow{K_1^d \text{ mod } N} \cdots \xrightarrow{K_{t-2}^d \text{ mod } N} K_{t-1}$$

- Client unwinds keys:

  $K_{i-1} = K_i^e \text{ mod } N$

$$K_0 \xleftarrow{K_1^e \text{ mod } N} K_1 \xleftarrow{K_2^e \text{ mod } N} \cdots \xleftarrow{K_{t-1}^e \text{ mod } N} K_{t-1}$$

Dynamically growing + efficient



[Micali:87]
[Jakobsson:02]

Implementation of Chefs = SFSRO + access control

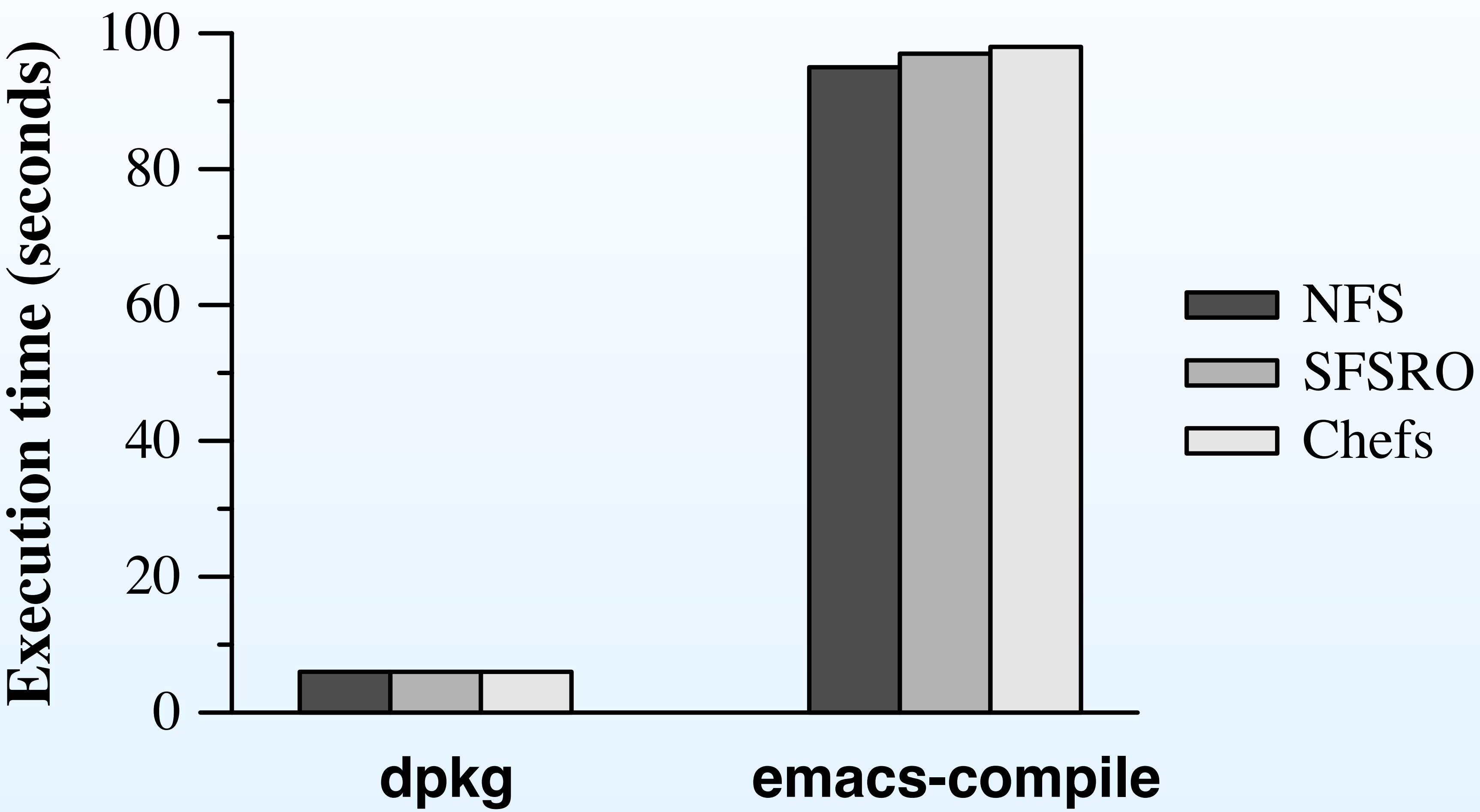
- Server remains unmodified
- More sophisticated algorithm for incremental updates
  - Lazy revocation
  - Database re-generation not idempotent
- Key regression based on SHA-1
  - Keys downloaded out-of-band
  - Must extract pseudorandom keys from unpredictable keys

Performance evaluation

Performance evaluation

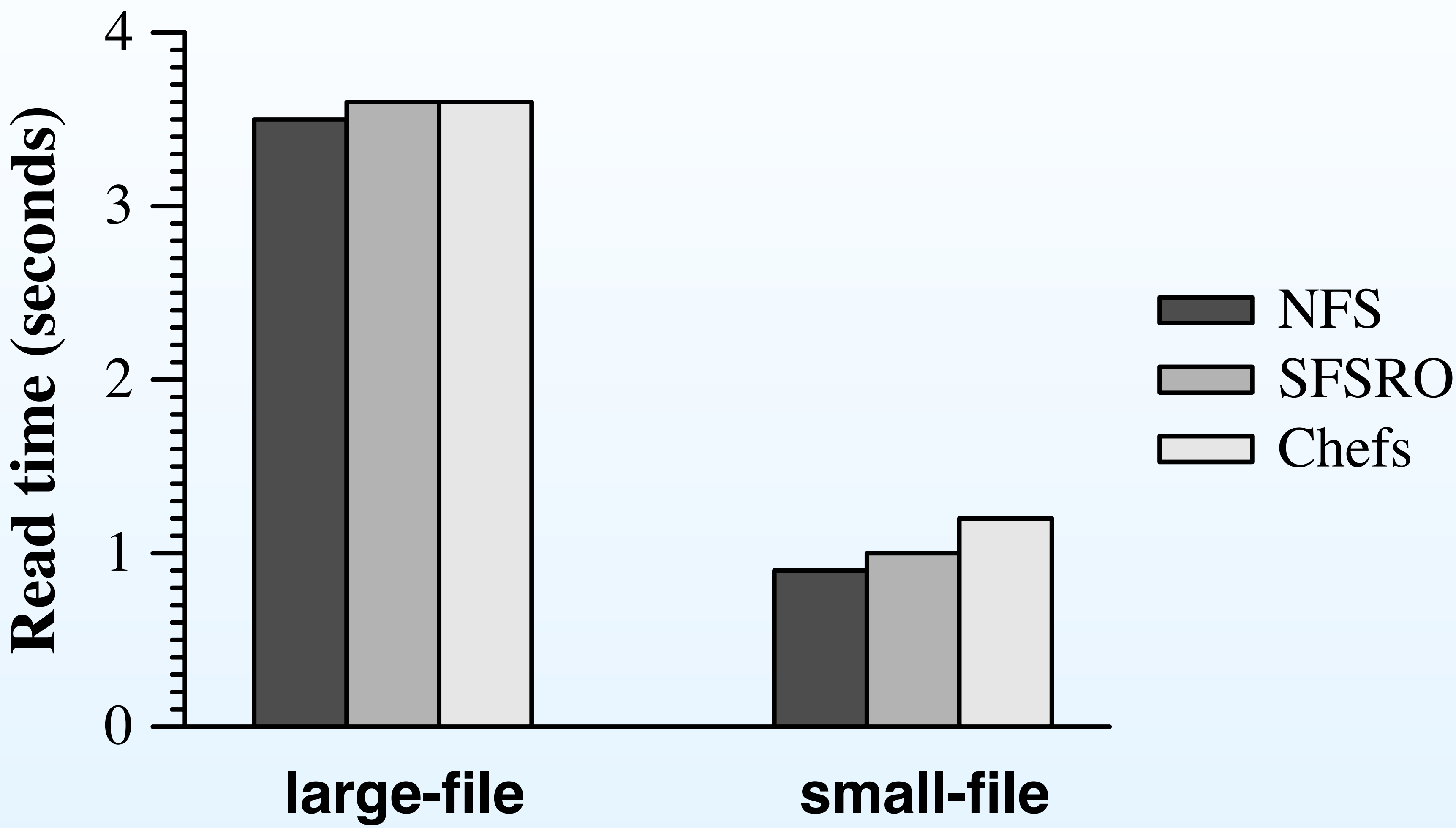
- Throughput independent of a publisher's local resources
- Individual servers support many simultaneous clients
- Acceptable latency for clients
- Chefs performs equally to SFSRO, except for downloading keys

SFSRO and Chefs are efficient despite cryptography



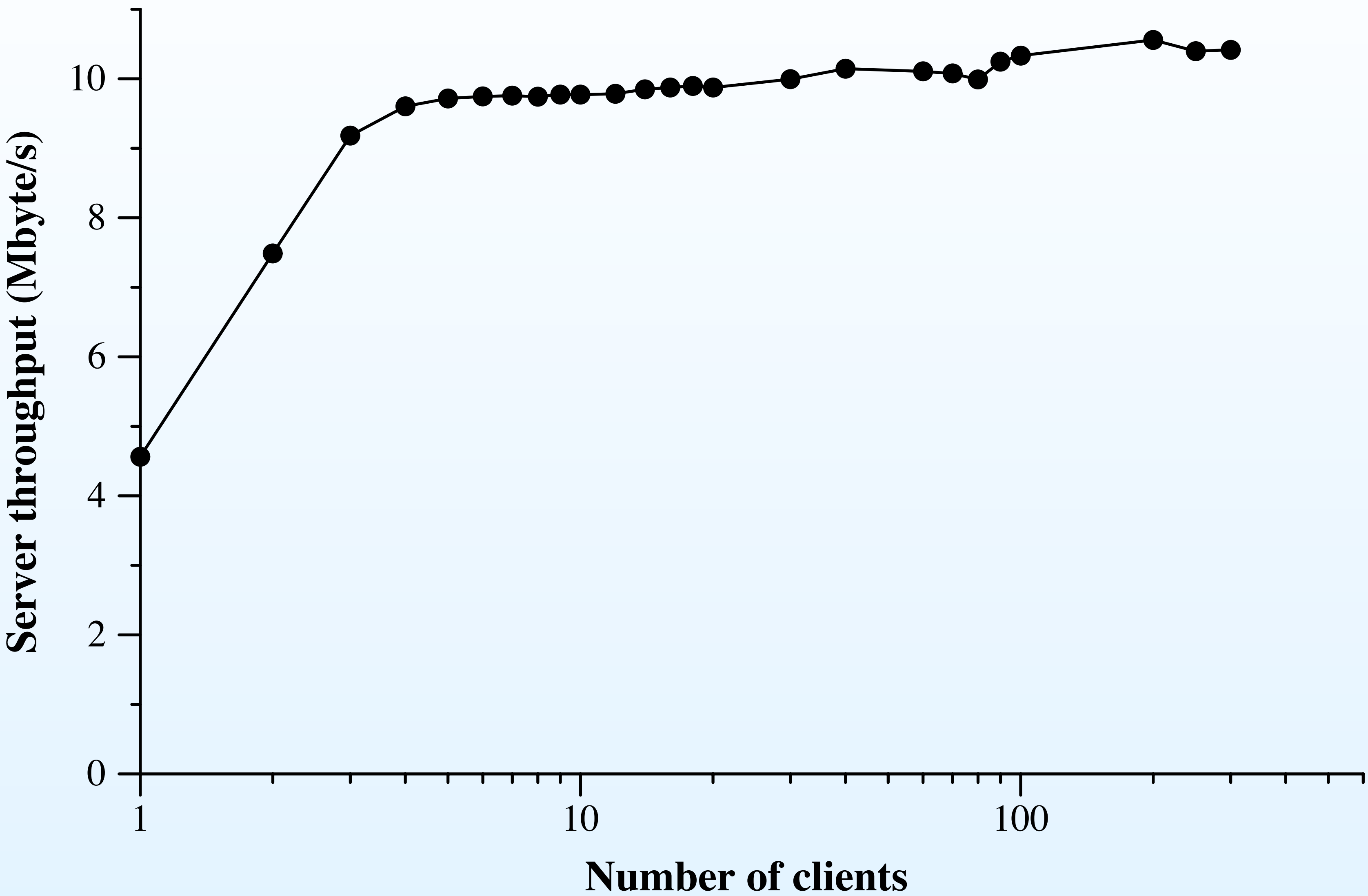
2.8GHz Pentium 4 machines, 100 Mbit network, 266 μsec round-trip

SFSRO and Chefs are efficient despite cryptography



2.8GHz Pentium 4 machines, 100 Mbit network, 266 μsec round-trip

Servers scale because no online crypto



550 MHz Pentium III machines, 100 Mbit (12.5 Mbyte/s) network

Wrap up

Related work

- Secure file systems:
  Swallow [Reed:81], Cryptographic FS [Blaze:93], Byzantine FS [Castro:99], OceanStore [Kubi:00], Farsite [Adya:02], Untrusted data repositories (SUNDR) [Mazières:02], Venti [Quinlan:02], Snapdragon [Aguilera:03]
- Content distribution networks:
  SHTTP [Rescorla:99], Consistent hashing [Karger:99], Publius [Waldman:2000], Cooperative FS [Dabek:01], Publish-Subscribe [Wang:02], Authentic data publication [Devanbu:02], BitTorrent [Cohen:03], CoDeeN [Pai:03], SSL splitting [Laas:03], XML access control [Miklau:03], Coral [Freedman:04]
- Cryptography:
  One-time signatures [Lamport:79], One-time passwords [Lamport:81], Merkle trees [Merkle:79], Timestamping [Haber:91], Key escrow [Micali:92], Forward-secure encryption [Anderson:97,Bellare:99], Fractal hash sequence traversal [Jakobsson:02], Self-healing keys [Staddon:02], Related-key attacks [Bellare:03], group key distribution

Future work

Past	Present	Future

Untrusted Storage and File Systems

Cepheus	SFSRO	Plutus	Key regression
[Masters'99]	[OSDI'00,TOCS'02]	[FAST'03]	[any day now]

Future work

Past	Present	Future

Untrusted Storage and File Systems

Email revocation [ACISP'97]	Cepheus [Masters'99]	SFSRO [OSDI'00,TOCS'02]	Plutus [FAST'03]	REX [USENIX'04]	Key regression [any day now]

Future work

Past	Present	Future

Untrusted Storage and File Systems

Email revocation	Cepheus	SFSRO	Plutus	REX	Key regression
[ACISP'97]	[Masters'99]	[OSDI'00,TOCS'02]	[FAST'03]	[USENIX'04]	[any day now]

Web authentication

[USENIX Security '01, CACM Sept '01]

Future work

Past	Present	Future

Untrusted Storage and File Systems

Email revocation	Cepheus	SFSRO	Plutus	REX	Key regression
[ACISP'97]	[Masters'99]	[OSDI'00,TOCS'02]	[FAST'03]	[USENIX'04]	[any day now]

Web authentication
[USENIX Security '01, CACM Sept '01]

Proxy Re-Encryption
[NDSS'05], [ePrint '05]

RFID Security
[Reading signals]

Summary

- Distributing public content
  - Authenticity, integrity, freshness
  - High throughput
- Access control of private content
  - Efficient eviction
  - Efficient key distribution
- Implementation and performance measurements

Summary

- Distributing public content ←SFSRO
  - Authenticity, integrity, freshness
  - High throughput
- Access control of private content ←Chefs
  - Efficient eviction ←Lazy revocation
  - Efficient key distribution ←Key regression
- Implementation and performance measurements ←Works in practice

Linux BSDs Mac OS X

Bon Appetit

Download SFSRO and Chefs.

`http://www.fs.net/`

Questions?

Break in case of emergency

Key regression security

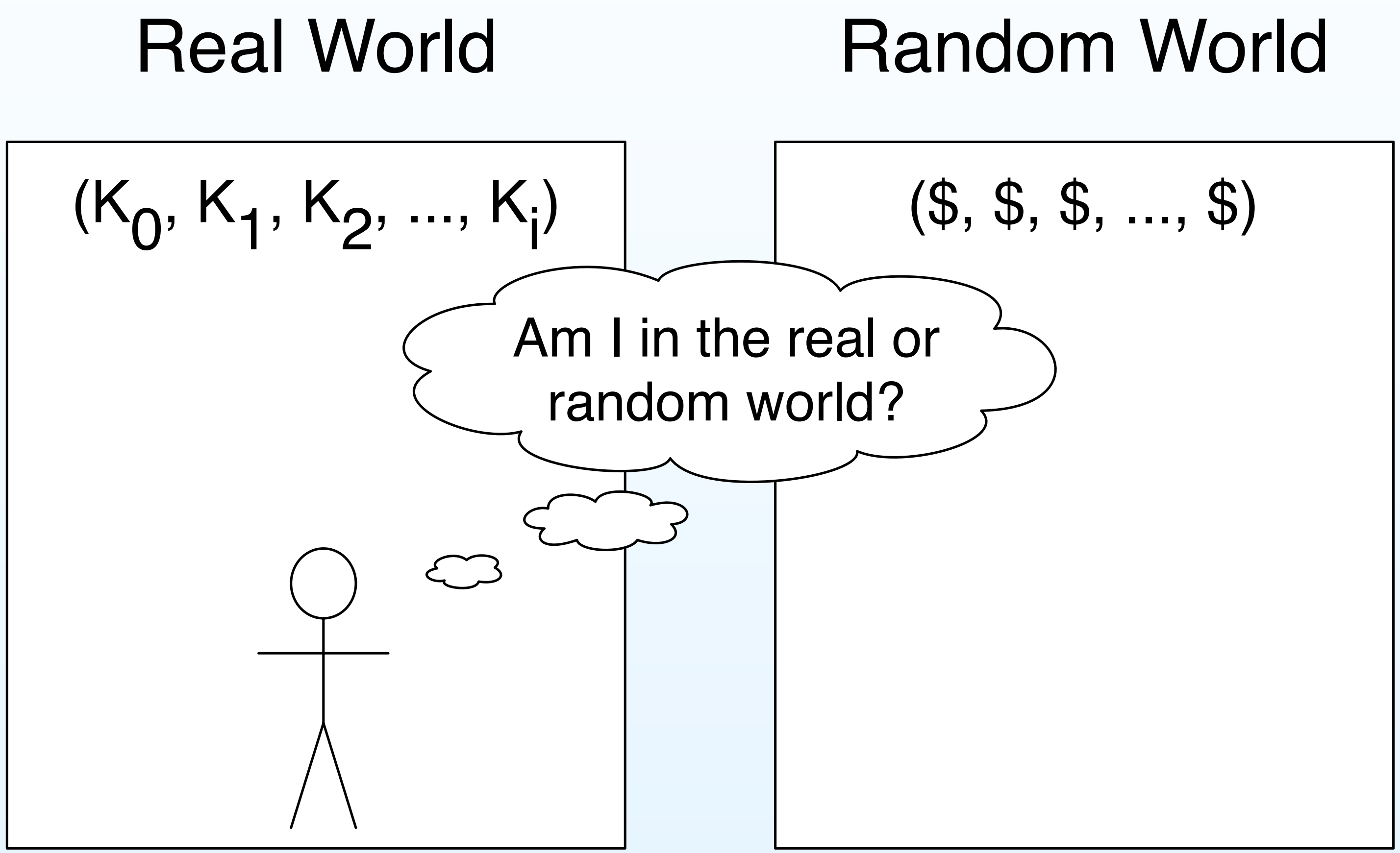
Real World

$(K_0, K_1, K_2, \ldots, K_i)$

Random World

(\$, \$, \$, ..., \$)

Key regression security



- Distinguish randomly generated sequence from key regression sequence?
- [Bellare:99, Bellare:03]

Represents a natural notion of security

- Why distinguishability instead of key recovery?
  - Captures notion of partial information
  - Only publisher can wind (unpredictable)
  - Only clients can unwind (pseudorandom)

- But are the hash-based and RSA-based schemes secure?

Keys must be unpredictable AND pseudorandom

- Hash-based scheme easily distinguishable
  - Given challenge, attempt to unwind
  - Check whether past keys match

Keys must be unpredictable AND pseudorandom

- Hash-based scheme easily distinguishable
  - Given challenge, attempt to unwind
  - Check whether past keys match
- RSA-based scheme easily distinguishable
  - What if $e = 3$
  - Guess N by looking at the size of keys
  - Check if unwinding works with $(e = 3, N)$

Solution: extract pseudorandomness

- Publisher winds *intermediate* keys:

  $\kappa_{i+1} = \kappa_i^d \bmod N$

$$\kappa_0 \xrightarrow{\kappa_0^d \bmod N} \kappa_1 \xrightarrow{\kappa_1^d \bmod N} \cdots \xrightarrow{\kappa_{t-2}^d \bmod N} \kappa_{t-1}$$

Solution: extract pseudorandomness

- Publisher winds *intermediate* keys:

$\kappa_{i+1} = \kappa_i^d \bmod N$

$$\kappa_0 \xrightarrow{\kappa_0^d \bmod N} \kappa_1 \xrightarrow{\kappa_1^d \bmod N} \cdots \xrightarrow{\kappa_{t-2}^d \bmod N} \kappa_{t-1}$$

- Client unwinds *intermediate* keys:

$\kappa_{i-1} = \kappa_i^e \bmod N$

$$\kappa_0 \xleftarrow{\kappa_1^e \bmod N} \kappa_1 \xleftarrow{\kappa_2^e \bmod N} \cdots \xleftarrow{\kappa_{t-1}^e \bmod N} \kappa_{t-1}$$

Solution: extract pseudorandomness

- Publisher winds *intermediate* keys:

$\kappa_{i+1} = \kappa_i^d \bmod N$

$$\kappa_0 \xrightarrow{\kappa_0^d \bmod N} \kappa_1 \xrightarrow{\kappa_1^d \bmod N} \cdots \xrightarrow{\kappa_{t-2}^d \bmod N} \kappa_{t-1}$$

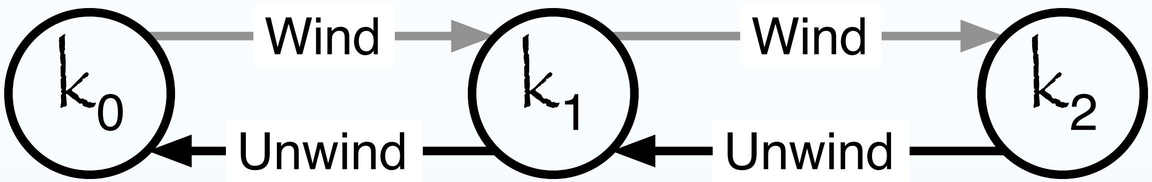
- Client unwinds *intermediate* keys:

$\kappa_{i-1} = \kappa_i^e \bmod N$

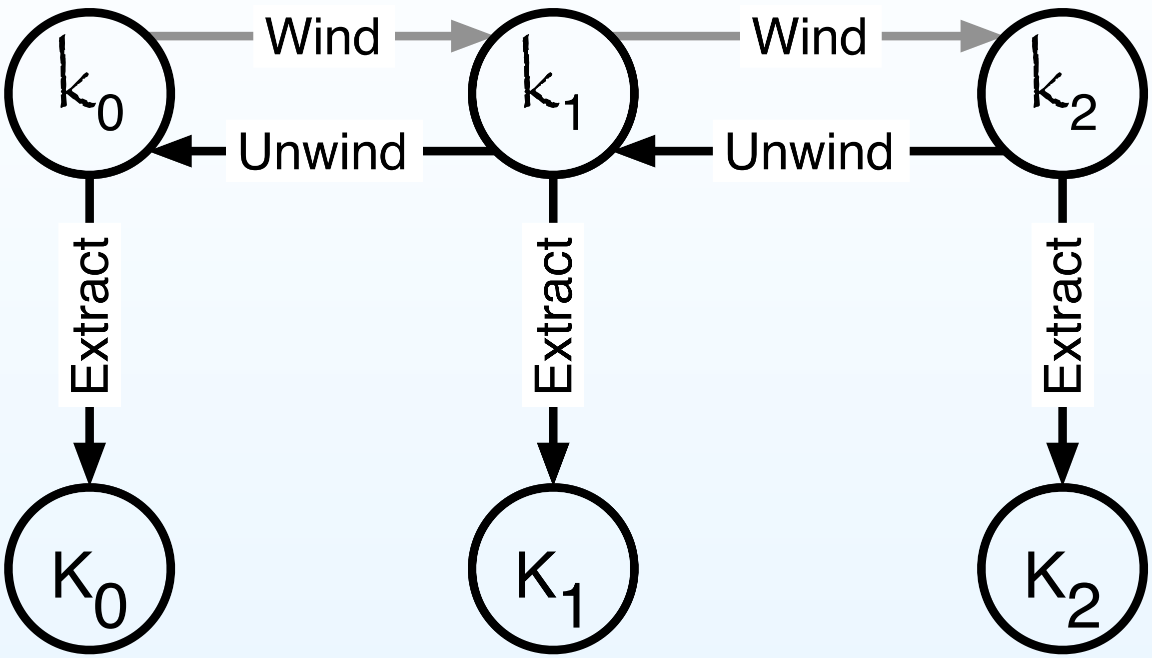
$$\kappa_0 \xleftarrow{\kappa_1^e \bmod N} \kappa_1 \xleftarrow{\kappa_2^e \bmod N} \cdots \xleftarrow{\kappa_{t-1}^e \bmod N} \kappa_{t-1}$$

- Extract pseudorandom K_i from unpredictable κ_i
  - Using a one-way function: $K_i = F(\kappa_i)$

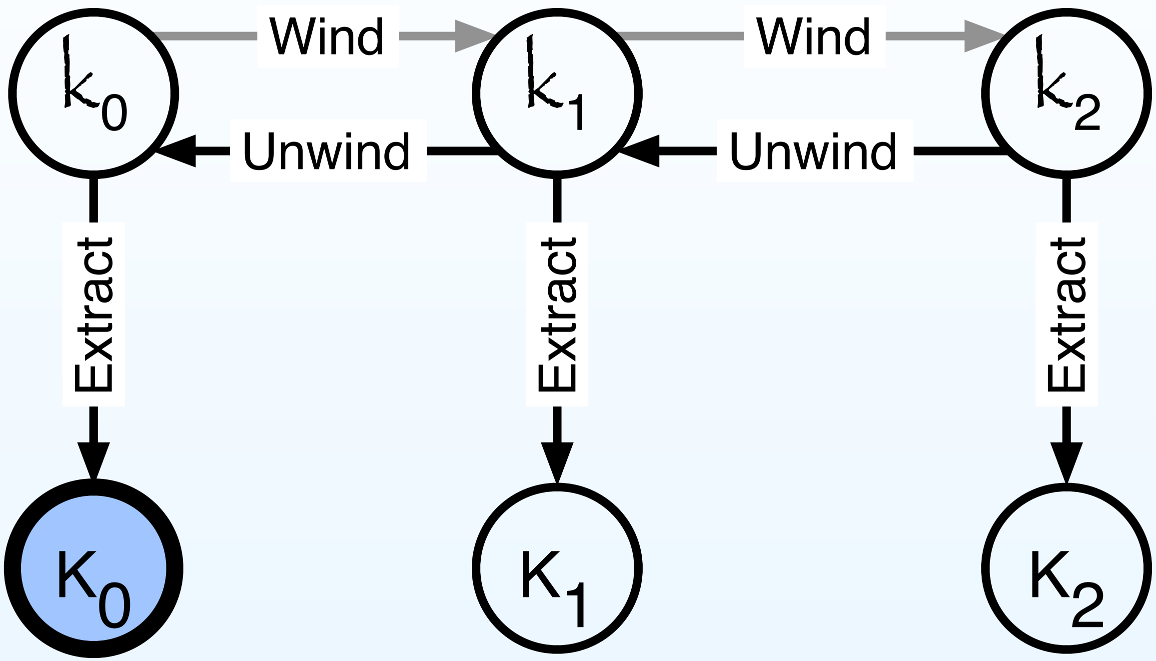
Security of key regression with extractor



Security of key regression with extractor

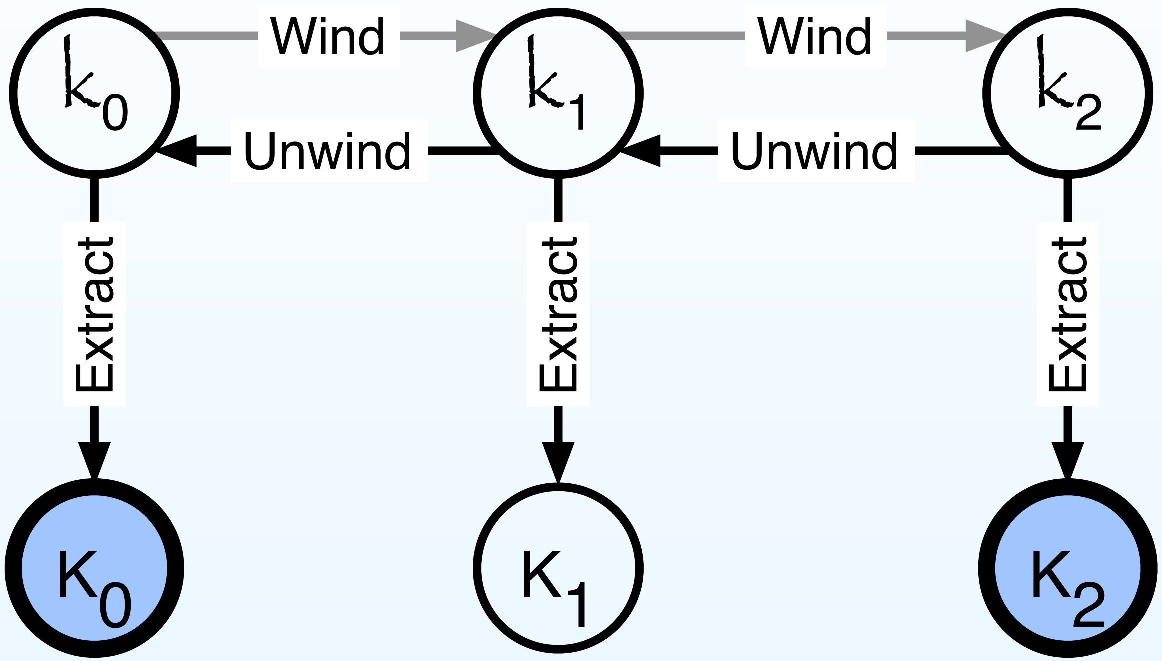


Security of key regression with extractor



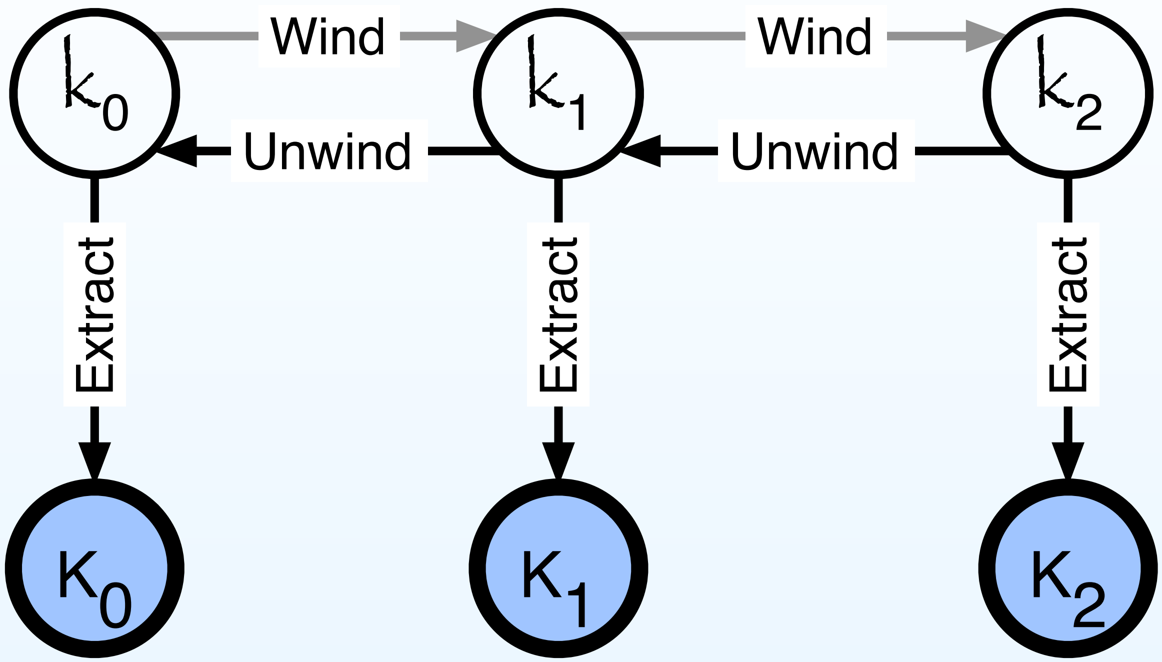
- Adversary queries oracle for keys

Security of key regression with extractor



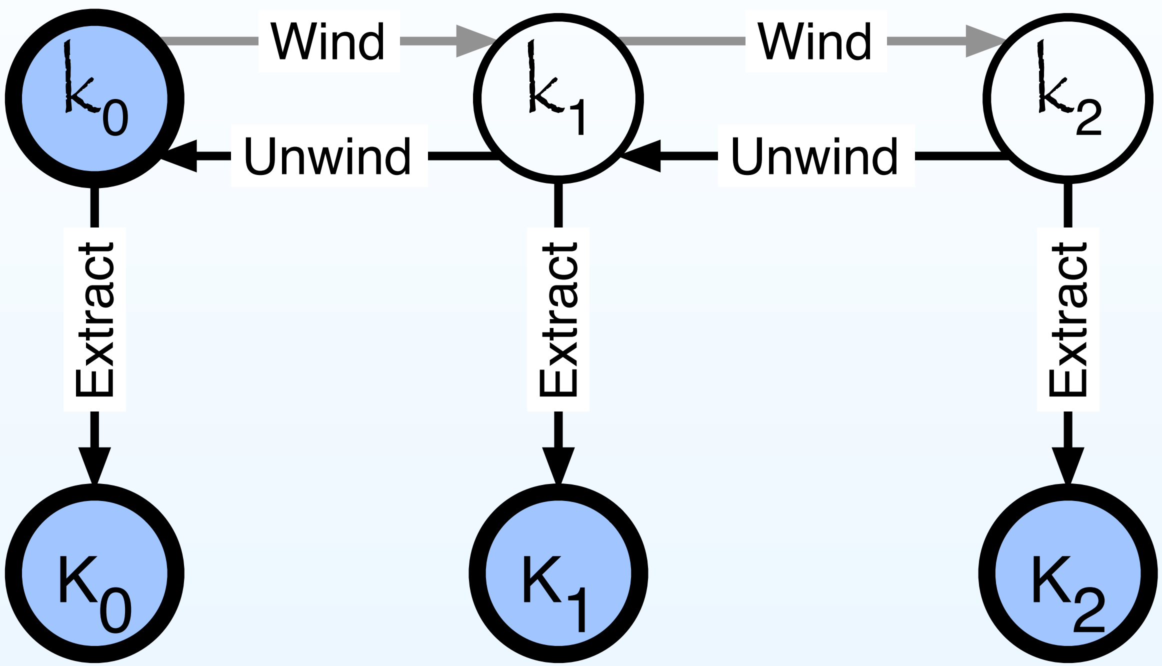
- Adversary queries oracle for keys

Security of key regression with extractor



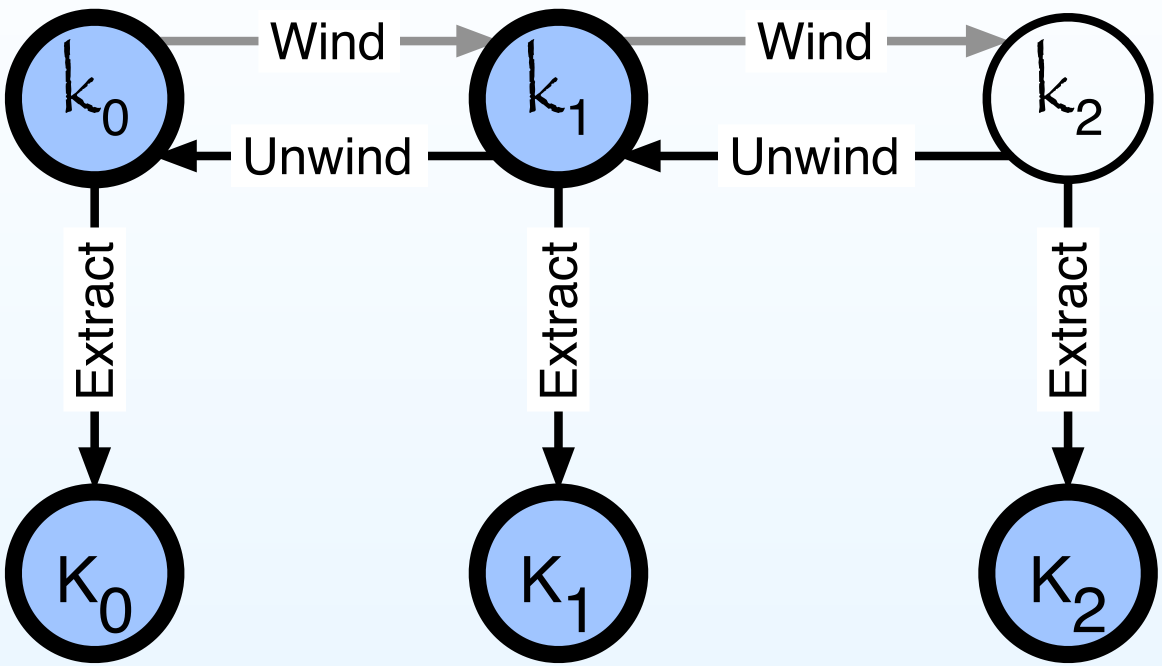
- Adversary queries oracle for keys

Security of key regression with extractor



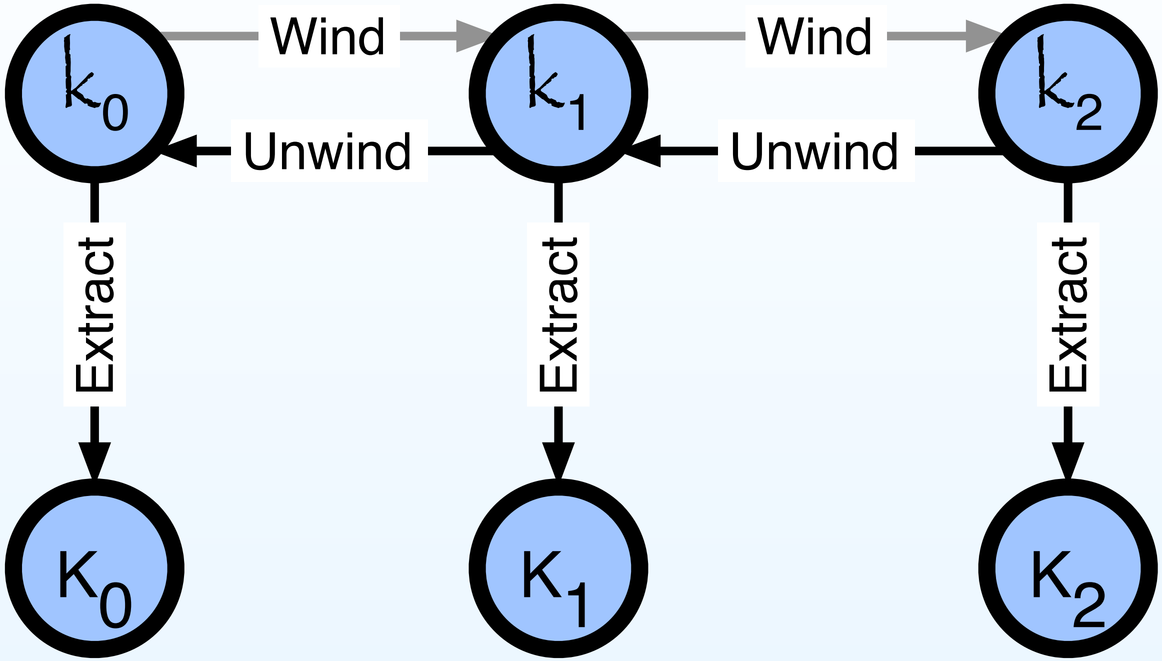
- Adversary queries oracle for keys
- Adversary queries oracle for intermediate keys

Security of key regression with extractor



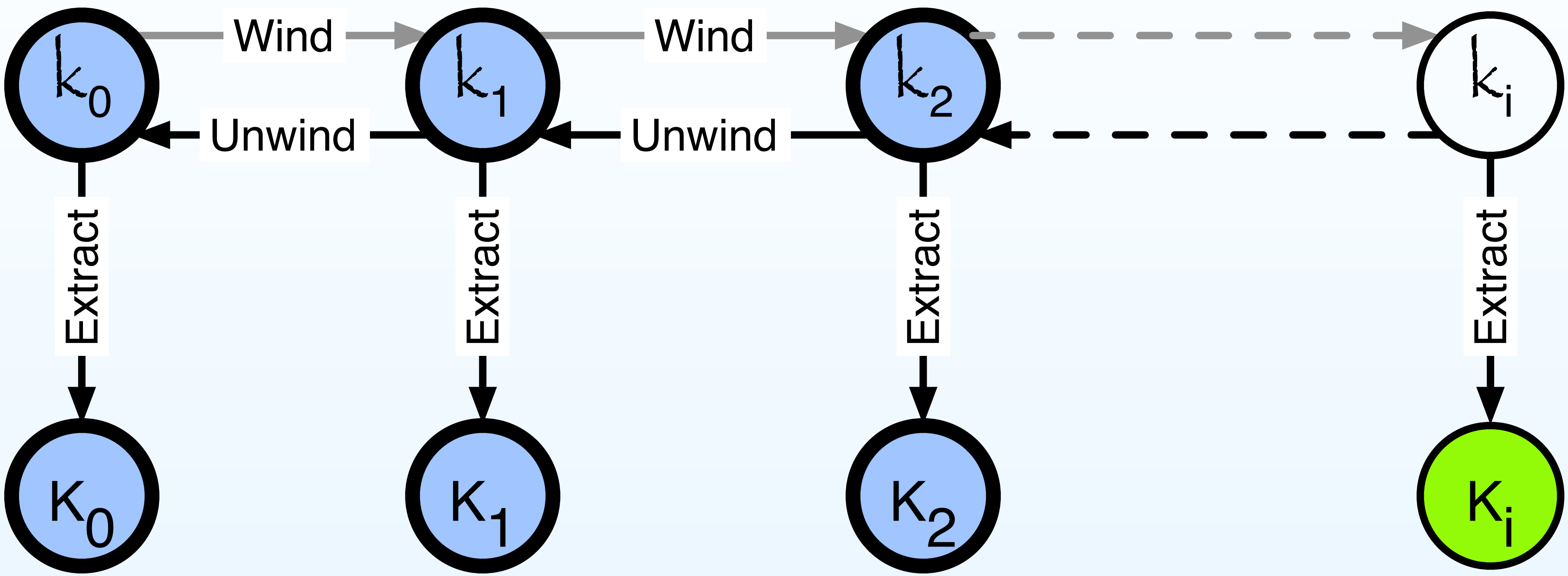
- Adversary queries oracle for keys
- Adversary queries oracle for intermediate keys

Security of key regression with extractor



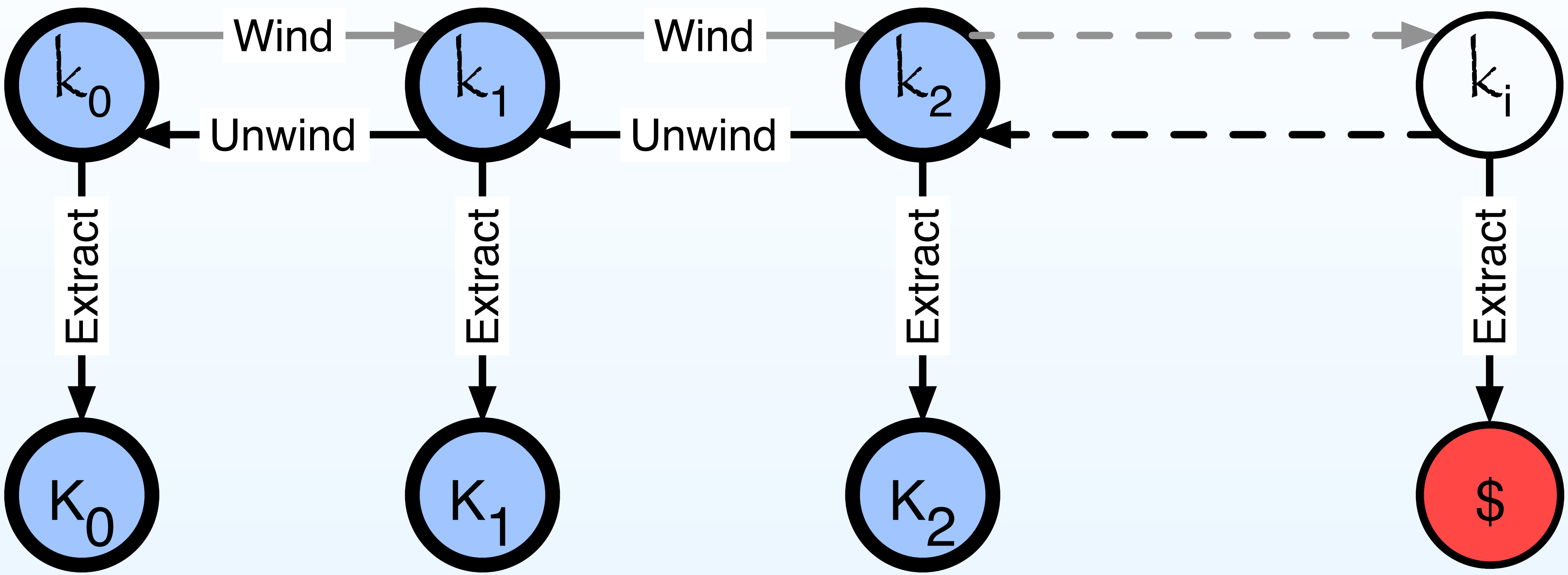
- Adversary queries oracle for keys
- Adversary queries oracle for intermediate keys

Security of key regression with extractor



- Adversary queries oracle for keys
- Adversary queries oracle for intermediate keys
- Adversary receives real or random challenge

Security of key regression with extractor



- Adversary queries oracle for keys
- Adversary queries oracle for intermediate keys
- Adversary receives real or random challenge
- Notion works for arbitrary constructions

Emergency Slide: Hash collisions

- Collision resistance (find any two inputs)
  - Brute force 2^{80}
  - Wang, Yin, Yu attack (2^{69})
  - 2^{69} bytes $\equiv$ 524,288 Pbytes
- 2nd pre-image resistance (find a second input)
  - Brute force 2^{159}
  - Kesley, Schneier 2^{106} for particular messages

Emergency Slide: Economics

- Incentives
- How to collect payments
- Fair sharing

Emergency Slide: Applications

- Public content
  - Certificate authorities
  - Software distribution
- Private content
  - Subscriptions
  - Time-delayed release

Emergency Slide: SFSRO protocol

- CONNECT () – Initiate SFSRO protocol
- GETFSINFO () – Get signed hash of root directory
- GETDATA (*hash*) – Get block with *hash* value
- All data interpreted entirely by client
  - Server need know nothing about file system structure
  - Makes server fast and simple (< 400 lines of code)

Emergency Slide: SHA-1 broken!

- Move cautiously to SHA-256 or others
- Rely on different type of collision resistance

Emergency Slide: Broadcast encryption

- Modified Naor-Pinkas non-interactive key distribution
- $K_i = g^{r_i P(0)}$ ←secret sharing in the exponent
- New this year: Boneh/Waters ePrint manuscript
- Communication vs. storage (lazy revocation)
- Broadcast imposes constraints on the key

Emergency Slide: Forward security

- Forward-secure encryption (signatures...)
- Key regression differences
  - Opposite of FSE + trapdoor
  - Adversary can ask oracle for future keys
  - Adversary can ask for intermediate keys
  - Secure enough for chosen-plaintext attack with XOR
  - Equivalency of key regression and FSE

Emergency Slide: Incremental replication

- Servers need transfer only modified data
  - Traverse file system w/ SFSRO protocol
  - Stores all hashes/values encountered in new database
  - Avoids re-transferring any hashes already in old database
  - Unchanged directories automatically pruned from transfer
- Makes short signature durations practical

Emergency Slide: Evicted clients?

- Easy to distinguish worlds
- Given a key sequence, run unwind
- If previous key matches, we are using real key regression

Emergency Slide: Limit unwinding

- Line segment rather than ray of keys
- Double hash chain method
- Join-leave-join