Efficient Tamper-Evident Data Structures for Untrusted Servers

Dan S. Wallach

Rice University

Joint work with Scott A. Crosby

This talk vs. Preneel’s talk

- Preneel: how hash functions work (or don’t work)
- This talk: interesting things you can build with hash functions (assumption: “ideal” hash functions)

Problem

- Lots of untrusted servers
  - Outsourced
    - Backup services
    - Publishing services
    - Outsourced databases
  - Insiders
    - Financial records
    - Forensic records
  - Hackers

Limitations and goals

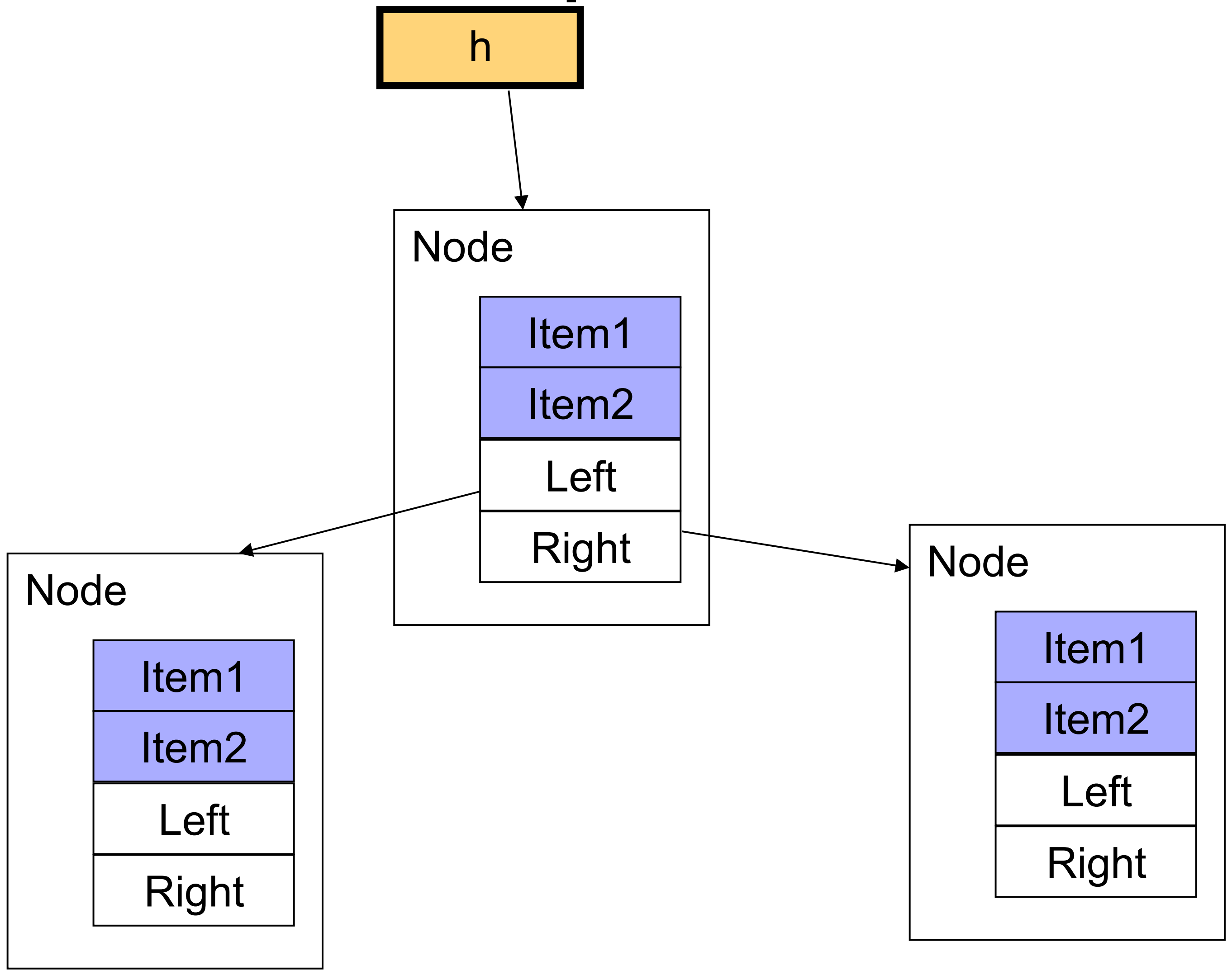
- Limitation
  - Untrusted server can do anything
- Best we can do
  - Tamper evidence
- Goal:
  - Tamper-evident primitives
    - Efficient
    - Secure

Tamper-evident primitives

- Classic
  - Merkle tree [Merkle 88]
  - Digital signatures

- More interesting ones
  - Tamper-evident logs [Kelsey and Schneier 99]
  - Authenticated dictionaries [Naor and Nissim 98]
  - Graph and geometric searching [Goodrich et al 03]
  - Searching XML documents [Devanbu et al 04]

Classic example: Merkle tree



Example: Tamper-evident logging

- Security model
  - Mostly untrusted clients
  - Untrusted log server
  - Trusted auditors
    - Detect tampering
- Useful for
  - Election results
  - Financial transactions

Example: Authenticated dictionary

- Security model
  - Data produced by trusted authors
  - Stored on untrusted servers
  - Fetched by clients
- Key-value data store
- Useful for
  - Price lists
  - Voting
  - Publishing

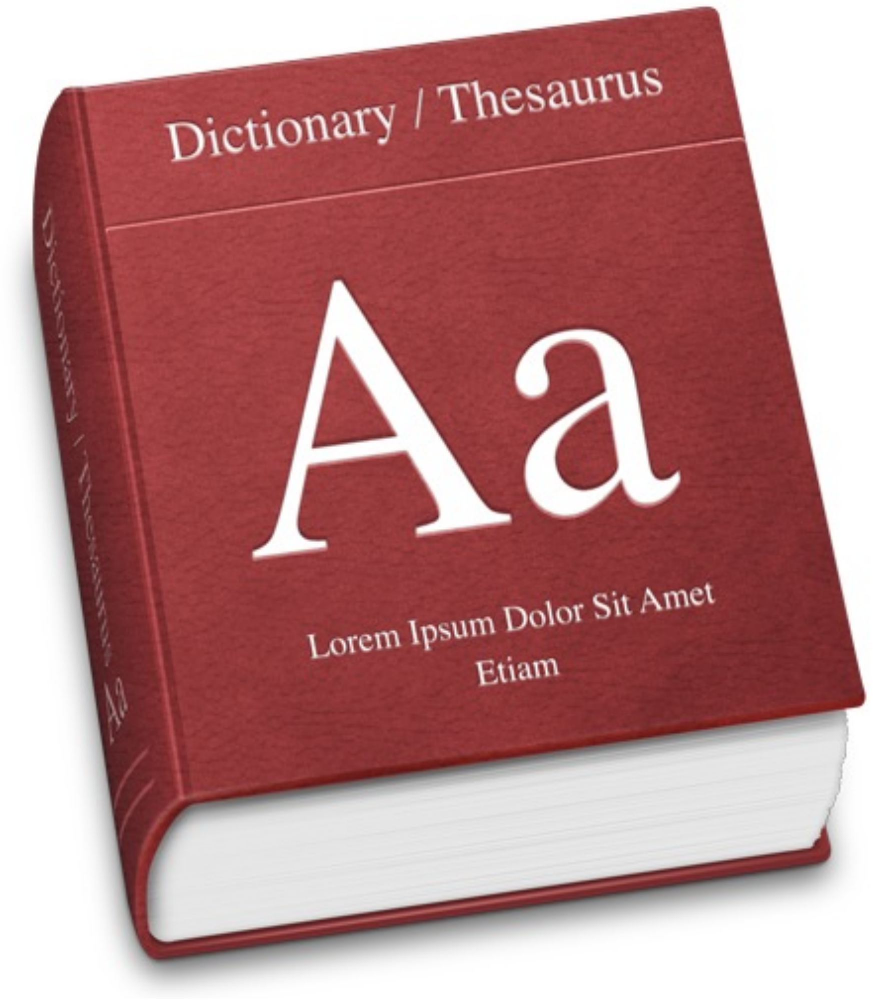
Our research

- Investigate two data structure problems
  - Persistent authenticated dictionary (PAD)
    - Efficiency improves from O(log n) to O(1)
  - Comprehensive PAD benchmarks
  - Tamper-evident log
    - Efficiency improves from O(n) to O(log n)
    - Newer work on fast digital signatures
- Code and papers online

  http://tamperevident.cs.rice.edu

Persistent authenticated dictionaries (PADs)

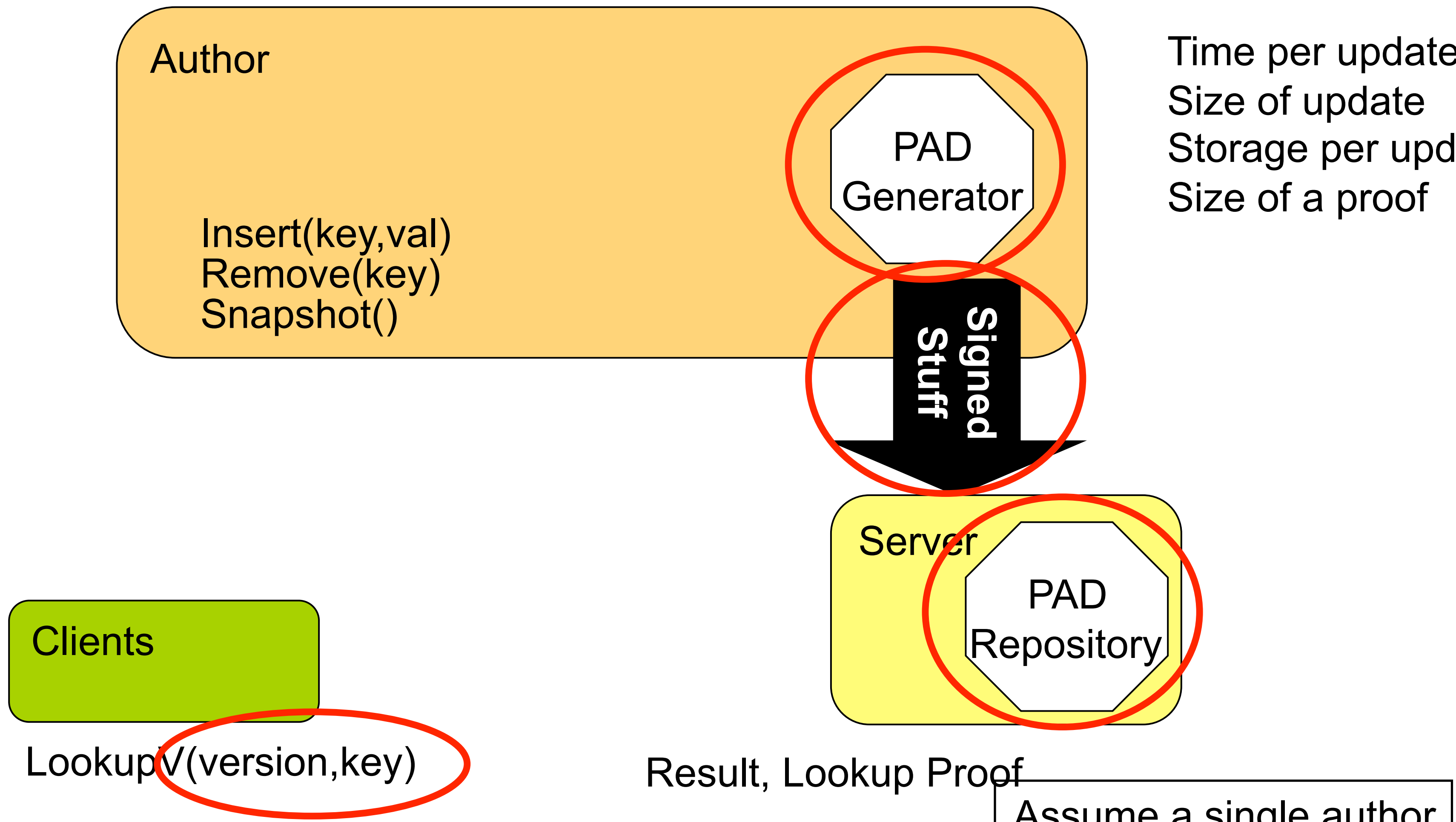
What is a PAD?


Dictionary / Thesaurus
Aa
Lorem Ipsum Dolor Sit Amet
Etiam
Dictionary / Thesaurus
Aa

What is a PAD?

- What is an authenticated dictionary?
  - Tamper-evident key/value data store
  - Invented for storing CRLs [Naor and Nissim 98]
- Security model
  - Created by trusted author
  - Stored on untrusted server
  - Accessed by clients
    - Responses authenticated by author's signature
- **PAD adds the ability to access old versions**
  - [Anagnostopoulos et al 01]

PAD design



Time per update
Size of update
Storage per update
Size of a proof

Assume a single author

Assume snapshot after every update

Applications of PADs

- Outsource storage and publishing
  - CRL
  - Cloud computing
  - Remote backups
  - Subversion repository
  - Stock ticker
  - Software updates
  - Smart cards
- Want to look up historical data

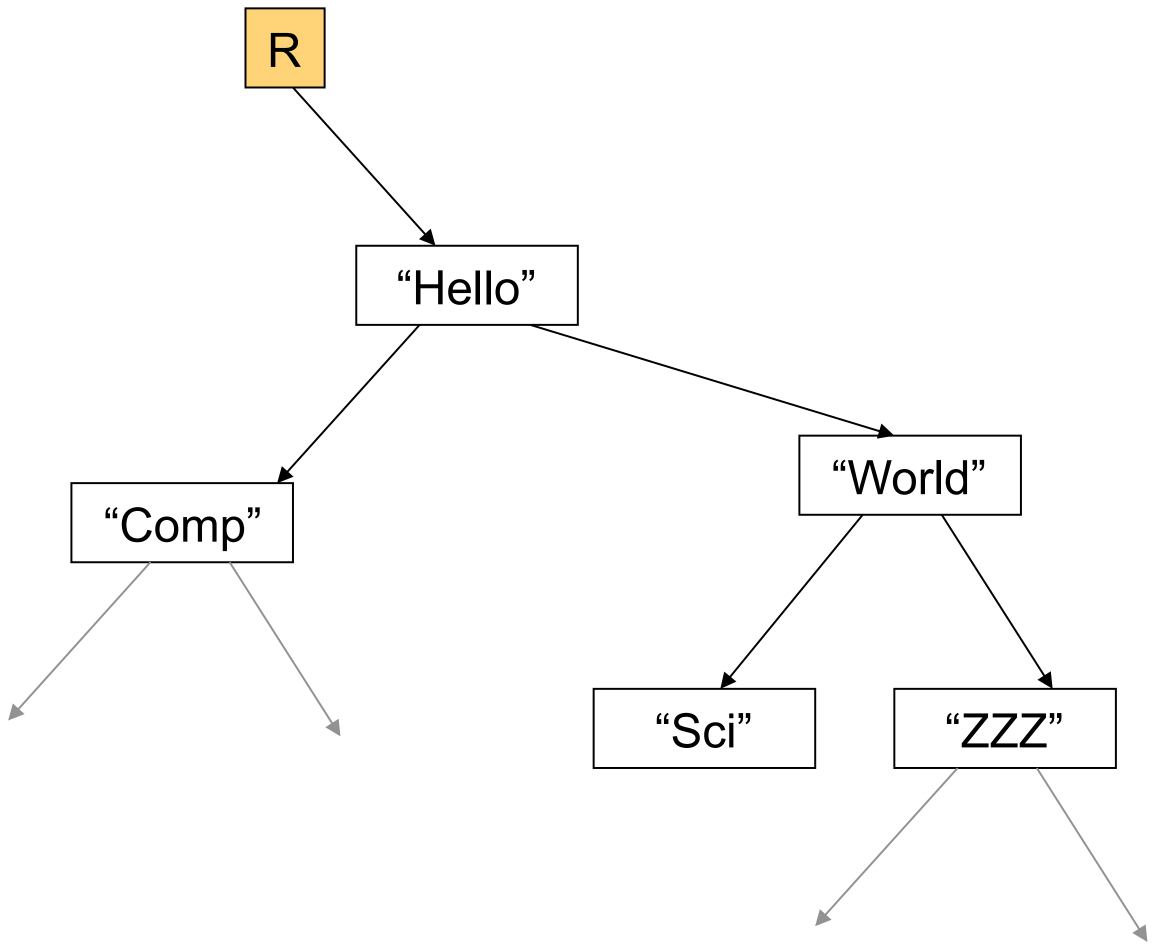
PAD Designs

- Tree-based PADs [Anagnostopoulos et al., Crosby and Wallach]
  - O(log *n*) storage per update
  - O(log *n*) lookup proof size
- Tuple PADS [Crosby and Wallach]
  - O(1) storage per update
  - O(1) proof size

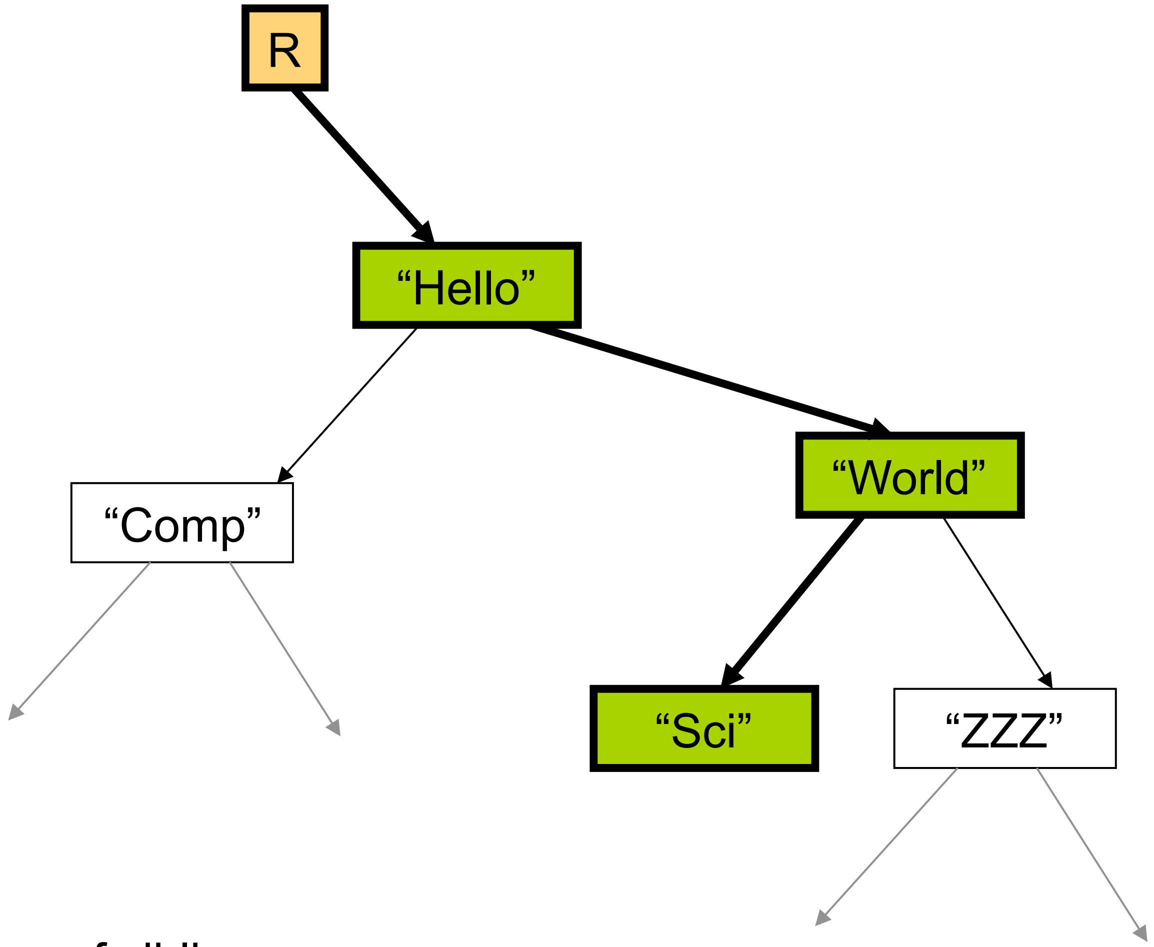
Other related work

- Authenticated dictionaries
  - [Kocher 1998,Naor and Nissim 1998]
- Merkle trees [Merkle 1988]

Tree-based authenticated dictionary

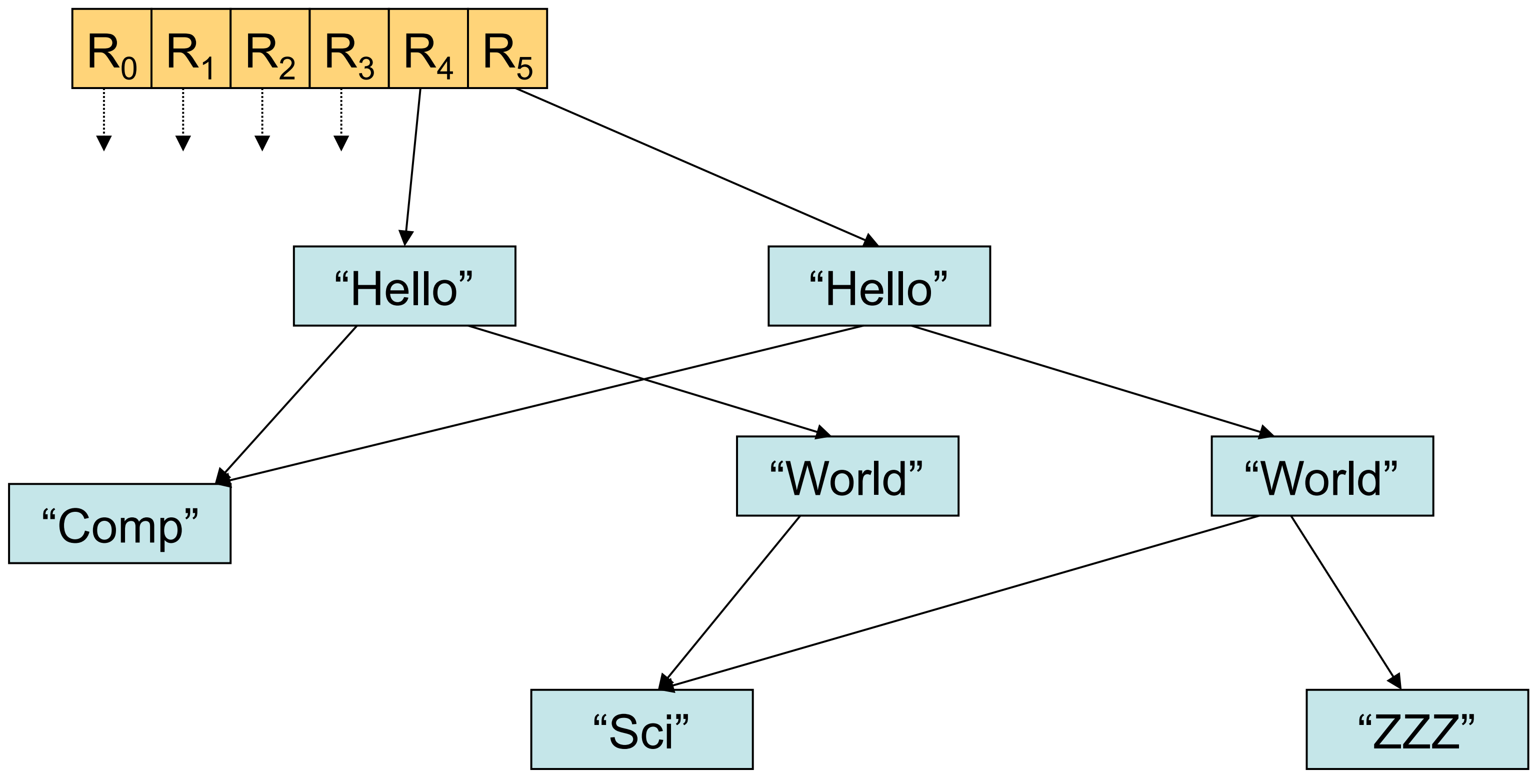


Proofs in a tree-based authenticated dictionary



Proof: Hashes of sibling nodes on path to lookup key

Path copying

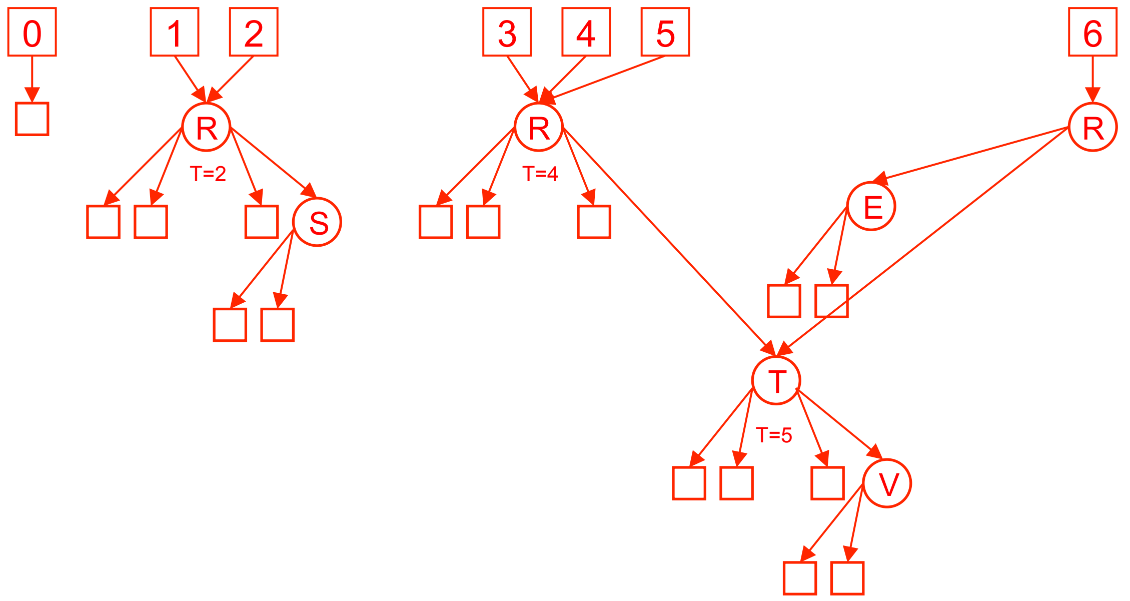


Storage: O(log n) per update

Building a PAD

- Other ways to make trees persistent
  - Versioned nodes [Sarnak and Tarjan 86]
    - O(1) amortized storage per update.
  - Our contribution:
    - Combining versioned nodes with authenticated dictionaries
    - Reduce memory consumption on the server

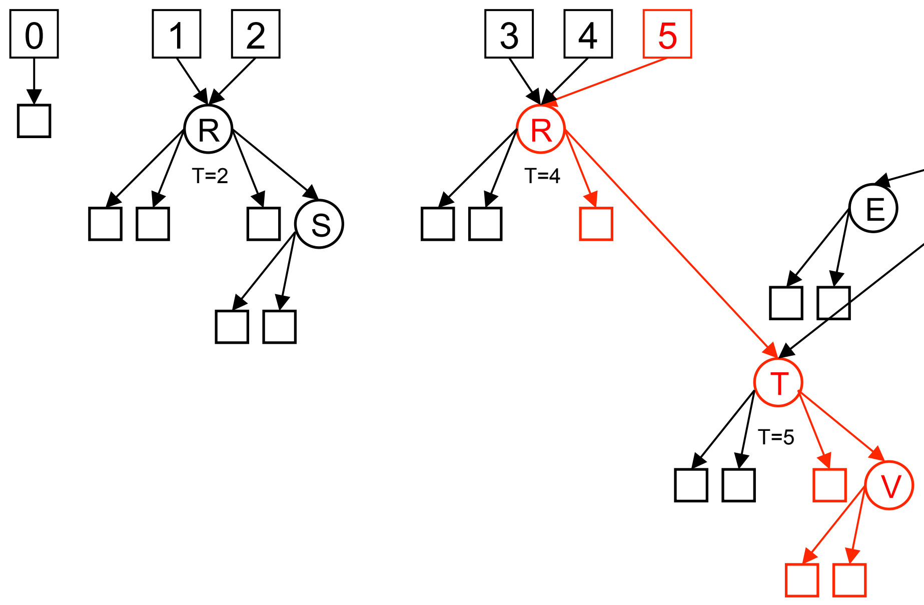
Sarnak-Tarjan tree



Add R
Add S
Del S
Add T
Add V
Add E

Note: 7 snapshots represented with 7 nodes.

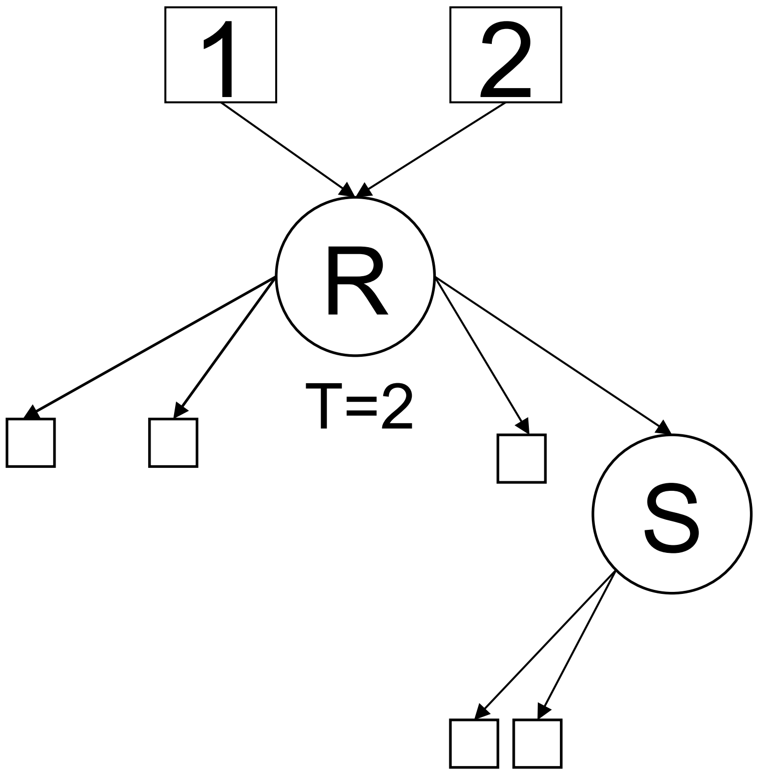
Accessing snapshot 5



Add R
Add S
Del S
Add T
Add V
Add E

Sarnak-Tarjan node

- Each node has two sets of children pointers
- Hash is not constant
- Not needed
  - Can be recomputed from tree
- Only a cache
  - Affect performance



Comparing caching strategies

	Storage	Lookup Proof Generation
	(Server)	(Server)
Cache nowhere	O(1)	O(n)
Cache everywhere	O(log n)	O((log n) *(log v))
Cache median layer	O(2)	O(√n * (log v))

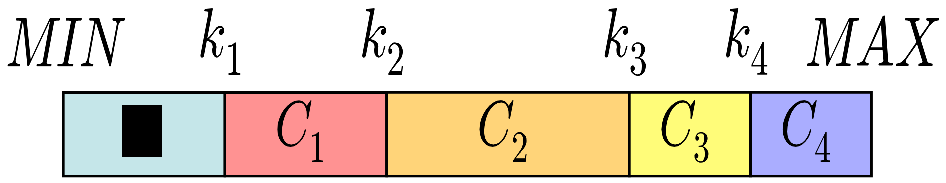
- Logarithmic
  - Update time
  - Lookup size
  - Verification time
- Constant
  - Update size

Tuple PADs

- Our new PAD design
  - **Constant lookup proof size**
  - **Constant storage per update**

Tuple PADs

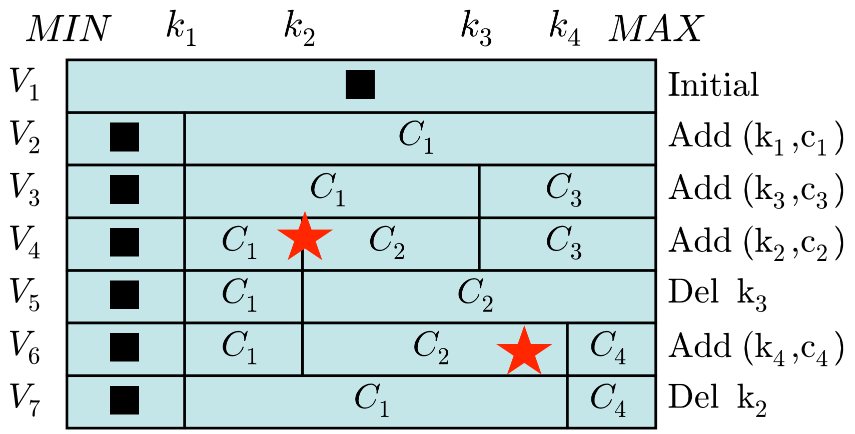
- Dictionary contents:
  - { $k_1 = c_1$, $k_2 = c_2$, $k_3 = c_3$, $k_4 = c_4$ }
- Divide key-space into intervals
- Tuples:
  - ([MIN,k_1),■)
  - ([k_1,k_2),c_1)
  - ([k_2,k_3),c_2)
  - ([k_3,k_4),c_3)
  - ([k_4,MAX),c_4)



"Key k_1 has value c_1, and there is no key in the dictionary between k_1 and k_2"

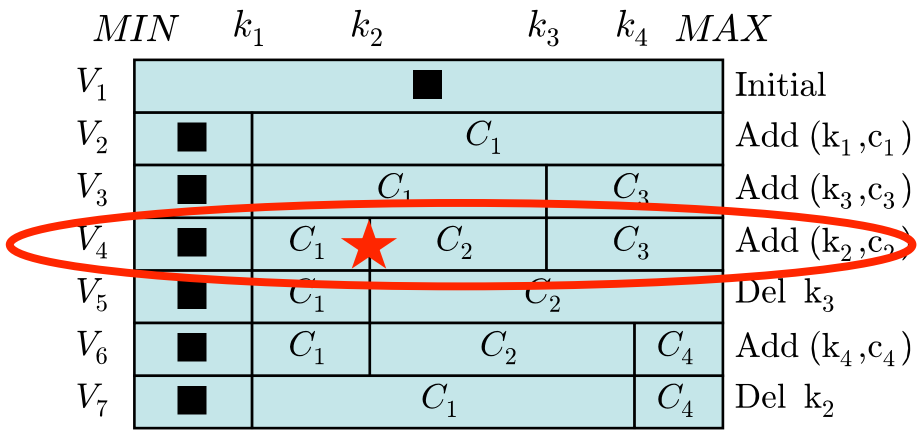
Making it persistent

- $(v_1,[k_1,k_2),c_1)$
  - "In snapshot v_1, key k_1 has value c_1, and there is no key in the dictionary between k_1 and k_2"



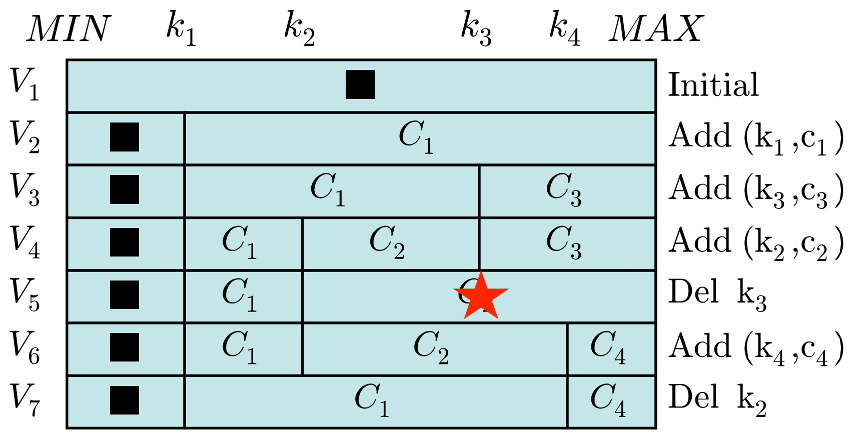
Lookups

- Proof that k_2 is in snapshot v_4
  - (v_4, [k_2,k_3), c_2), signed by author



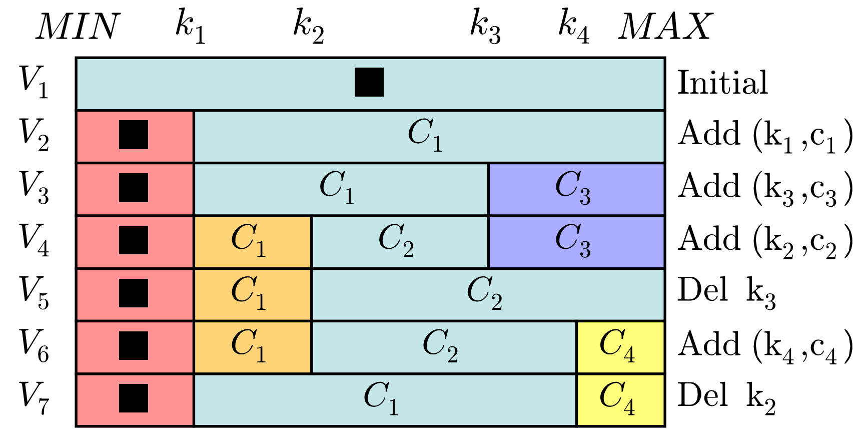
Lookups

- Proof that k_3 not in snapshot v_5
  - – (v_5, [k_2,k_4), c_2), signed by author



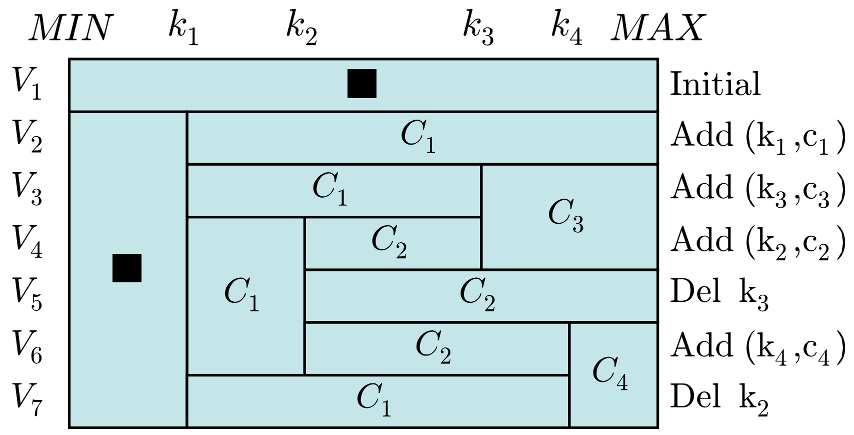
Observation

- Most tuples stay same between snapshots
- Every update
  - Creates ≤ 2 tuples not in prior snapshot

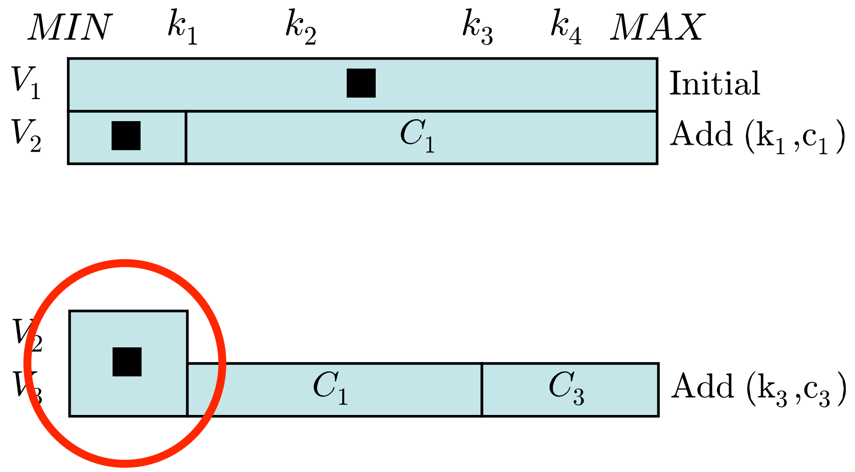


Tuple superseding

- Indicate a version range in each tuple
  - ([v_1,v_2+1], [k_1,k_2),c_1)
    - Which replaces ([v_1,v_2], [k_1,k_2),c_1)
    - At most 2 new tuples. Rest are replaced
  - Constant
    - Storage on server
  - Still have the same
    - Update time
    - Update size

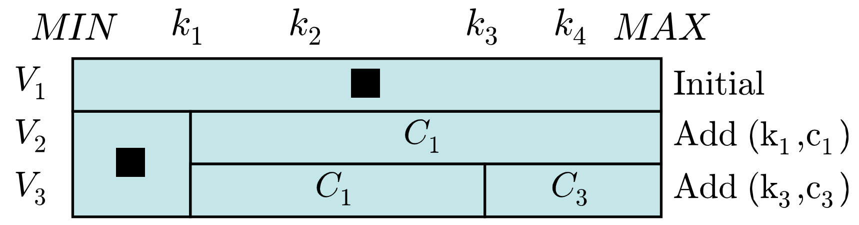


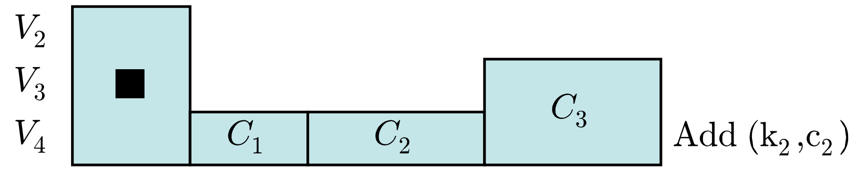
Tuple superseding



- ($[v_1,v_2]$, $[k_1,k_2)$, c_1)
  - “In snapshots v_1 through v_2 key k_1 has value c_1, and there is no key in the dictionary between k_1 and k_2”

Tuple superseding

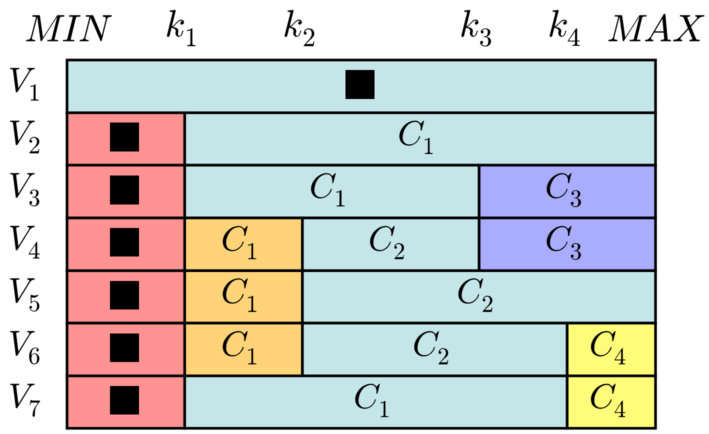




Lightweight signatures [Micali 1996]

- Most tuples are refreshed
- Can use lightweight signatures
  - Based on hashes
- Tuple includes iterated hash over random nonce
  - $A = \mathrm{H}^{k}(R)$
  - Author releases successive pre-images

Insight: Speculation

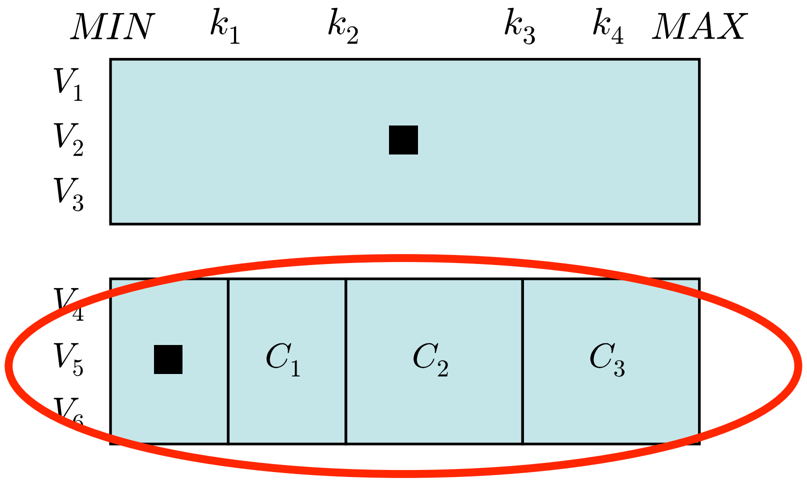


- Split PAD
  - Speculative tuples
    - Older generation
    - Signed in every epoch
  - Young generation
    - Correct mis-speculations
    - Signed every snapshot
    - Kept small, migrate keys into older generation
- O(G $n^{1/G}$) signatures per update
  - Combines with lightweight signatures

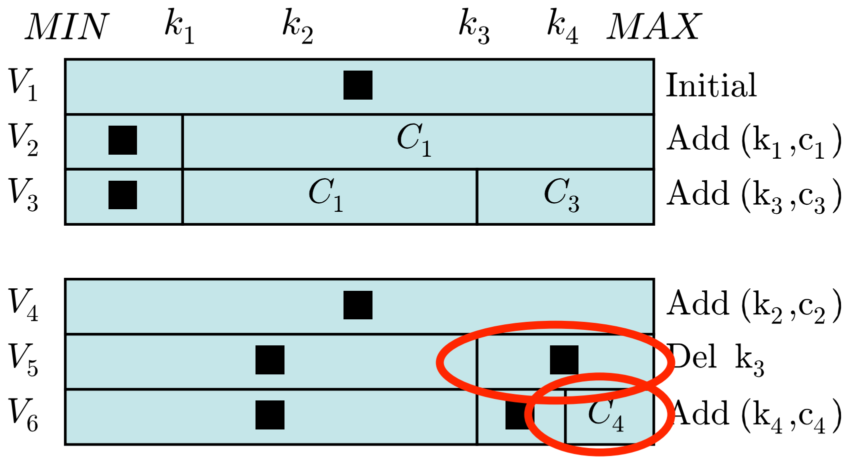
Speculation: Updating the PAD

- (g_0,[v_1,v_2],[k_1,k_2),c_1)
  - "In generation g_0 and snapshots v_1 through v_2 key k_1 has value c_1, and there is no key in the dictionary between k_1 and k_2"

Old generation g_1



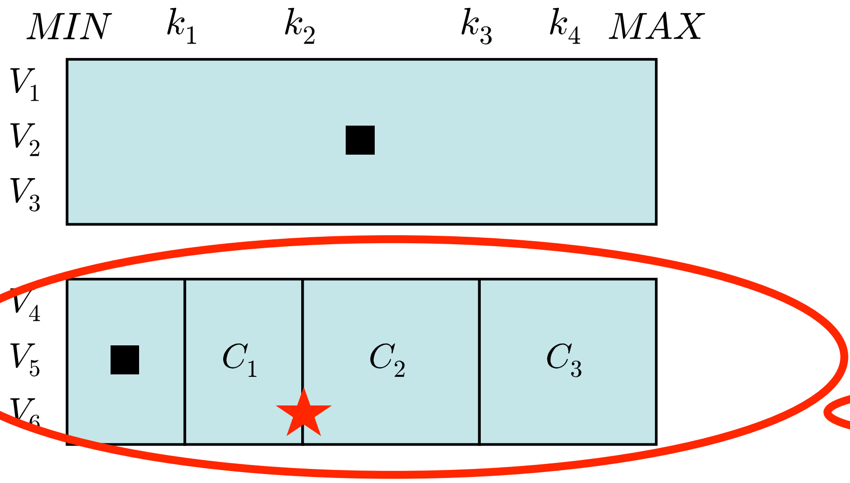
Young generation g_0



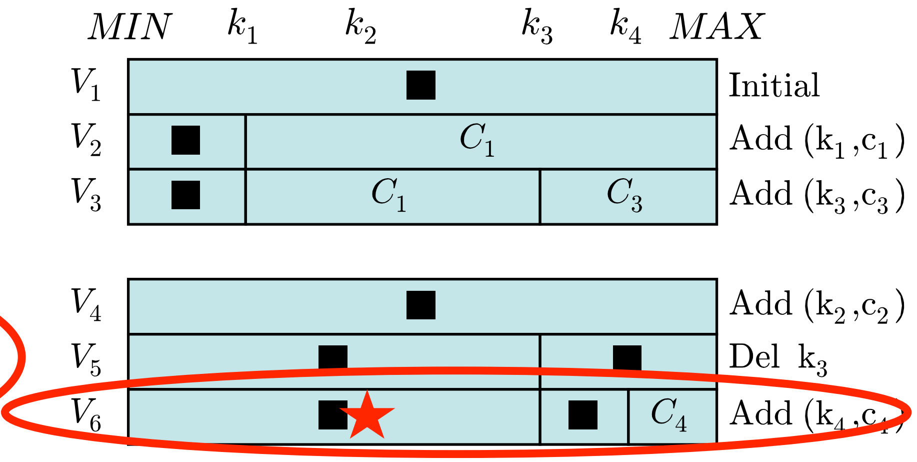
Speculation: Generating Proofs

- Proof that k_2 is in v_6
  - $(g_1,[v_4,v_6],[k_2,k_3),c_2)$ $(g_0,v_6,[\text{MIN},k_3),■)$

Old generation g_1



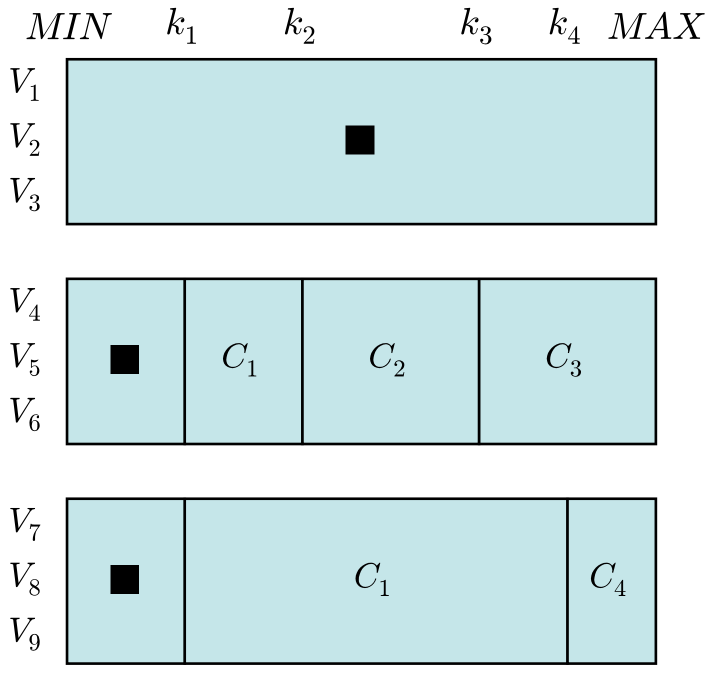
Young generation g_0



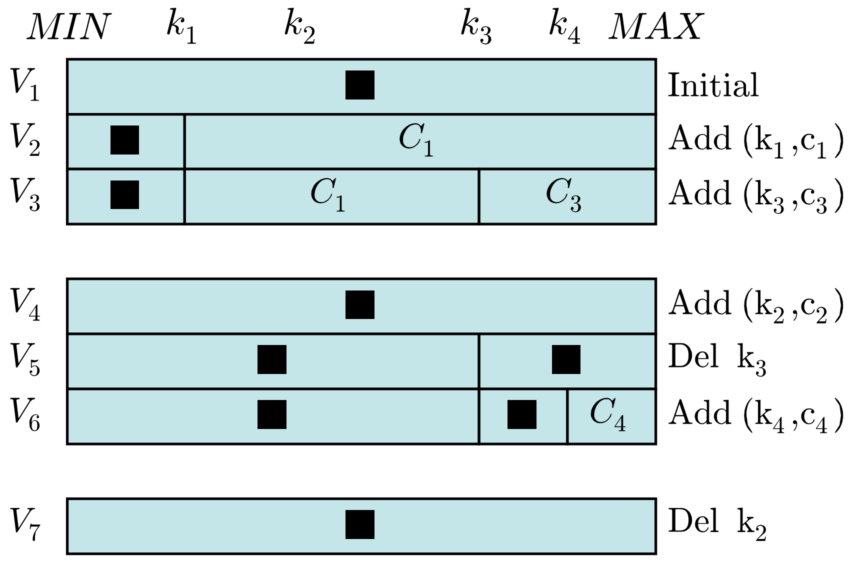
Speculation: Updating the PAD

- (g_0,[v_1,v_2],[k_1,k_2),c_1)
  - "In generation g_0 and snapshots v_1 through v_2 key k_1 has value c_1, and there is no key in the dictionary between k_1 and k_2"

Old generation g_1

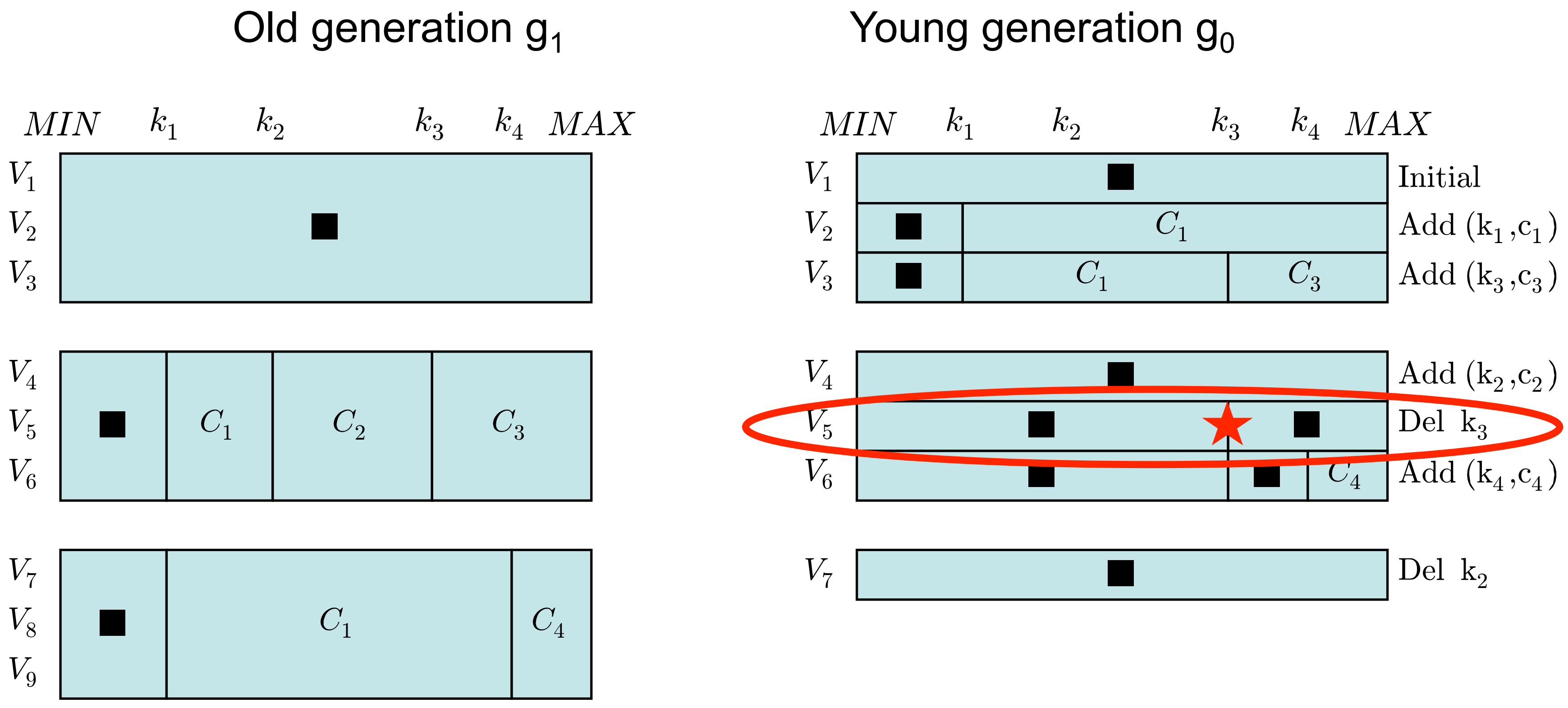


Young generation g_0



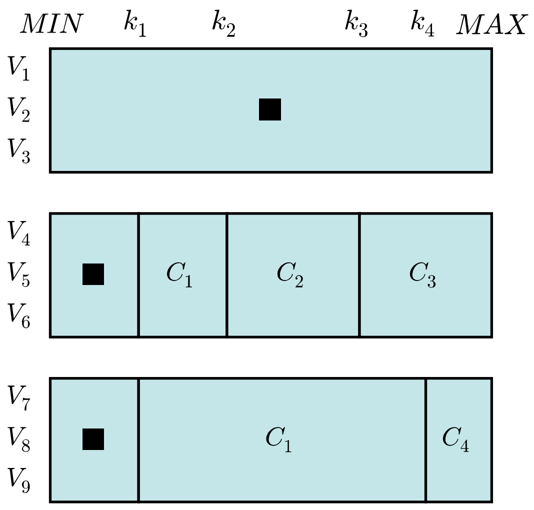
Speculation: Generating Proofs

- Proof that k_3 is not in v_5
  - $(g_0, v_5, [k_3, \mathrm{MAX}), ■)$

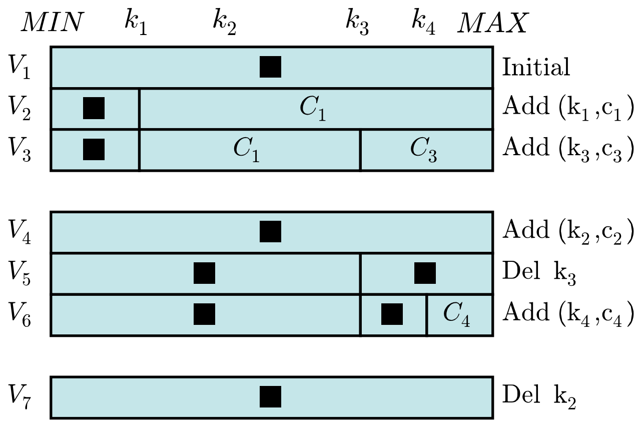


Costs of speculation

Old generation g_1



Young generation g_0

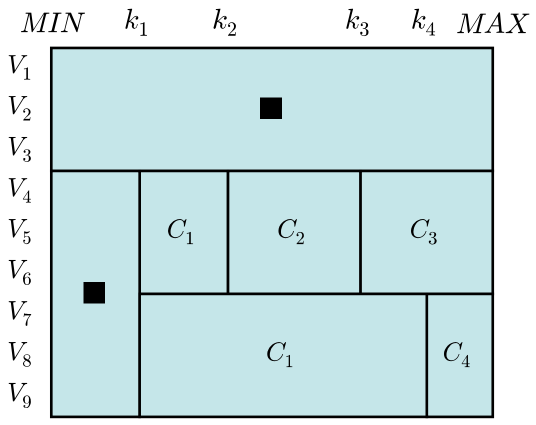


- Every E snapshots
  - O(*n*) signatures
- Each snapshot:
  - O(E) signatures

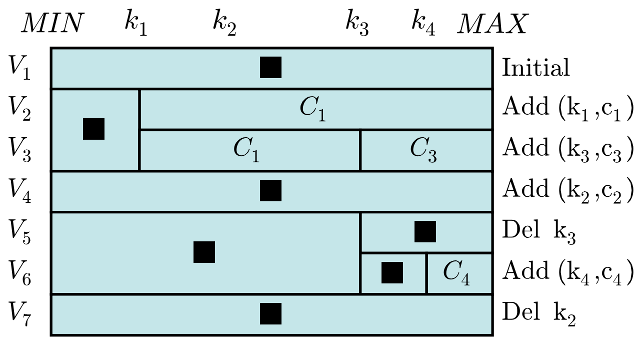
Overall: O(*n*/E + E) signatures per update. Minimum of O(2√*n*) when E=√*n*

Speculation and Superseding

Old generation g_1



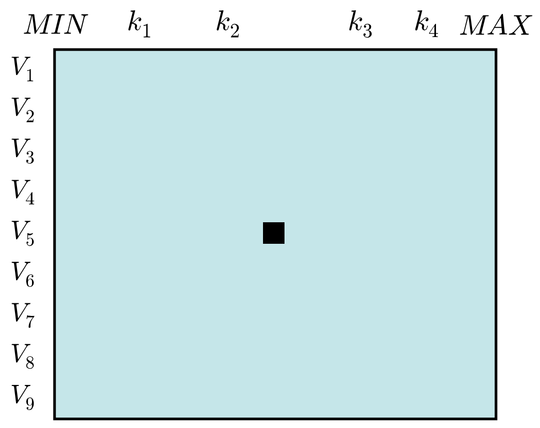
Young generation g_0



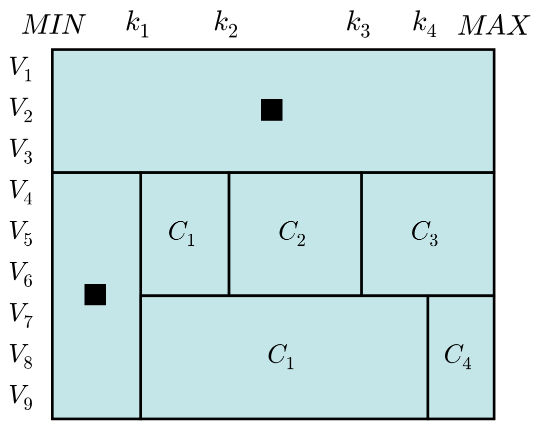
- O(2) storage per update
- O(2√*n*) signatures per update
- O(2) proof size

Multiple generations

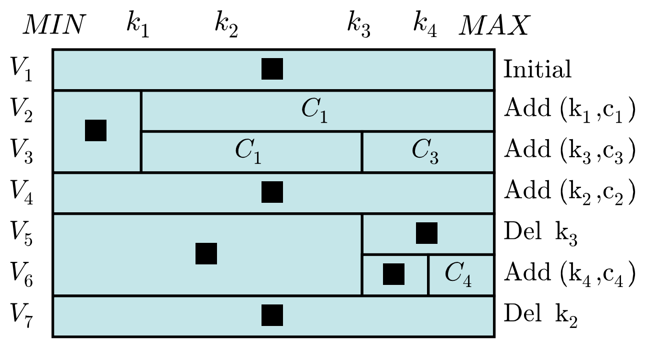
Oldest generation g_2



Middle generation g_1



Young generation g_0



- O(G) storage per update
- O(G $n^{1/G}$) signatures per update
- O(G) proof size

Reducing update costs

- Currently $O(G\ n^{1/G})$ update size
  - Requiring $O(G\ n^{1/G})$ work

- RSA accumulators [Benaloh and de Mare 93]
  - O(1)
    - Work on author
    - Update size
    - Lookup proof size
  - $O((G+1)\ n^{1/G}\ (\log n))$
    - Computation on server
    - Large constant factors

RSA accumulators [Benaloh, de Mare]

- Prove set membership
  - Constant size
  - $A = g^{a\ b\ c\ d\ e\ f} \pmod{n}$
    - A is signed by author
  - Prove membership:
    - (c, w_c) + signature on A
    - $w_c = g^{a\ b\ d\ e\ f} \pmod{n}$
  - Verify:
    - $A == (w_c)^c$?
- Computing witnesses
  - Need one for each tuple
  - O(n log n) exponentiations
- Combine
  - Tuple PAD
    - Speculation
    - Superseding
  - Accumulator

Comparing techniques

		Tree-based			Tuple-based		
		Path Copying	Cache Everywhere	Cache Median	Speculating+ Superseding	Superseding	Accumulators + Speculating
Updates	Time (Author)						O(1)
	Time (Server)	O(log n)			O(G * $n^{1/G}$)	O(n)	O(G * log(n) * $n^{1/G}$)
	Size						O(1)
Storage	(per update)	O(log n)		O(1)	O(G)	O(1)	
Lookup	Time (Server)	O(log n)	O(log n * log v)	O(√n)	O(G * log n)	O(log n)	
	Size	O(log n)			O(G)	O(1)	

What about the real world?

		Tree-based	Tuple-based		
				uperseding	Accumulators + Speculating
Updates	Time (Author)			O(n)	O(1)
	Time (Server)				O(G * log(n) * $n^{1/G}$)
	Size				O(1)
Storage	(per update)			O(1)	
Lookup	Time (Server)			O(log n)	
	Size	O(log n)	O(G)	O(1)	

Benchmarking PADs

Comprehensive implementation

- 21 algorithms
- Including all earlier designs
  - Path copy skiplists and path copy red-black trees [Anagnostopoulos et al.]

- Analysis also applies to non-persistent authenticated dictionaries

Algorithms

- Tree PADs – 12 designs
  - (4) Path copying, 3 caching strategies
  - (3) Red-black, Treap, and Skiplist
- Tuple PADs – 6 algorithms
  - (2) With and without speculation
  - (3) No-superseding, superseding, lightweight signatures
- Accumulator PADs – 3 algorithms

Implementation

- Hybrid of Python and C++
  - GMP for bignum arithmatic
  - OpenSSL for signatures
- Core 2 Duo CPU at 2.4 GHz
  - 4GB of RAM
  - 64-bit mode

Benchmark

- ‘Growing benchmark’
  - Insert 10,000 keys with a snapshot after every insert
- Play a trace of price changes of luxury goods
  - 27 snapshots
  - 14000 keys
  - 39000 updates

Tree PADs

- Comparing algorithms
  - Red-black
    - Smallest proofs, least RAM, highest performance
  - Skiplists do the worst
- Comparing repositories
  - Path copying
  - Sarnak-Tarjan nodes cache everywhere
    - Same performance
    - 40% of the RAM

Cache median vs Cache everywhere

- 100,000 keys

	Update Size	Update Rate	Lookup Size	Lookup Rate	Memory usage
Cache median	.15kb	730/sec	1.5kb	196/sec	205MB
Cache everywhere	.15kb	730/sec	1.5kb	7423/sec	358MB

The costs of an algorithm

- Care about the monetary costs
- Use prices from cloud computing providers
  - Currently, 200kb is worth 1sec of CPU time
    - Worth about $ .000030 = 3000µ¢

Monetary analysis

- Evaluate
  - Absolute costs per operation
    - CPU time and bandwidth
  - Relative contribution of
    - CPU
    - Bandwidth

Tree PAD caching strategies

- 37x slower, but only costs 2x as much
  - Sending a lookup reply
    - 1.5kb, costing **18µ¢**
  - Generating a lookup reply
    - Cache median: 5ms, costing **16µ¢**
    - Cache everywhere .13ms : **.4µ¢**

	Lookup size	Lookup rate	Cost per lookup	Memory usage
Cache median	1.5kb	196/sec	34 µ¢	205MB
Cache everywhere	1.5kb	7423/sec	18 µ¢	358MB

Other insights

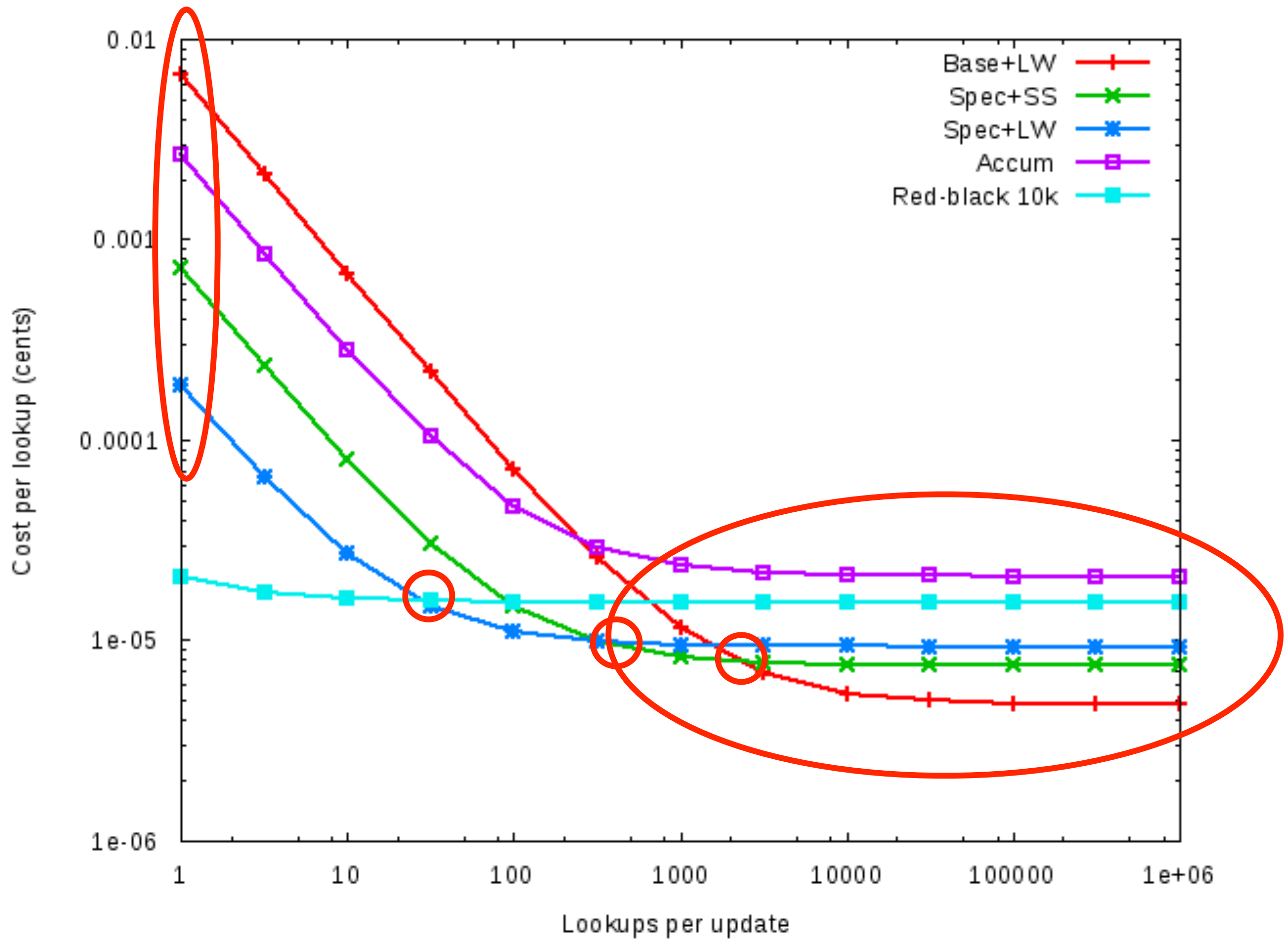
- Tuple PAD algorithms
  - Implemented in python
  - Slow
    - I estimate C++ would be 10x-30x faster
  - For lookups replies
    - 50%-70% monetary cost is in the message

Evaluating the monetary costs of updates and lookups

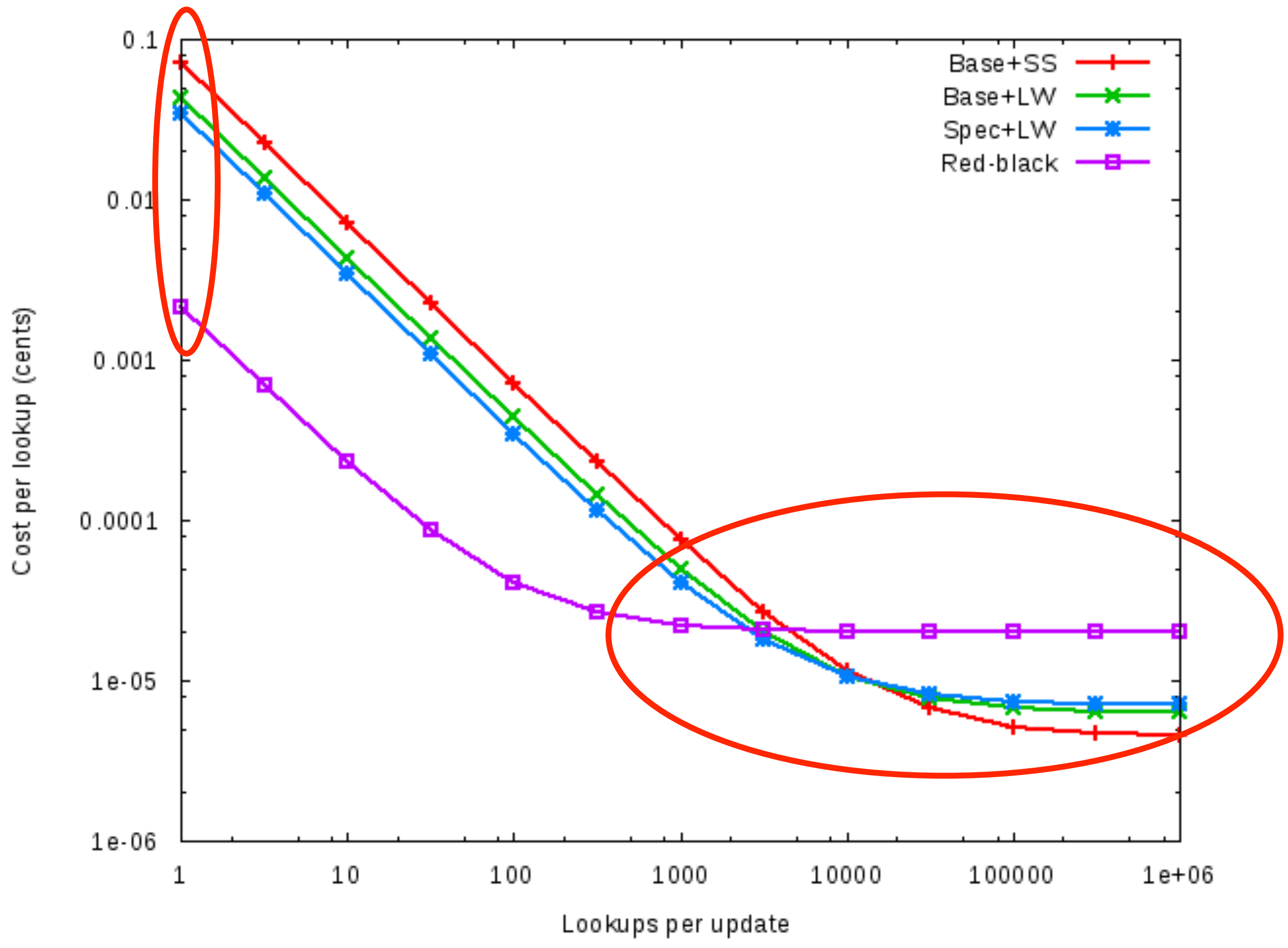
- Tuple PADs
  - Extremely cheap lookups
  - Expensive updates
- Tree PADs
  - Cheap lookups
  - Cheap updates

“What is the cost per lookup if there are *k* lookups for each update for different values of *k*.”

Costs per lookup on growing benchmark



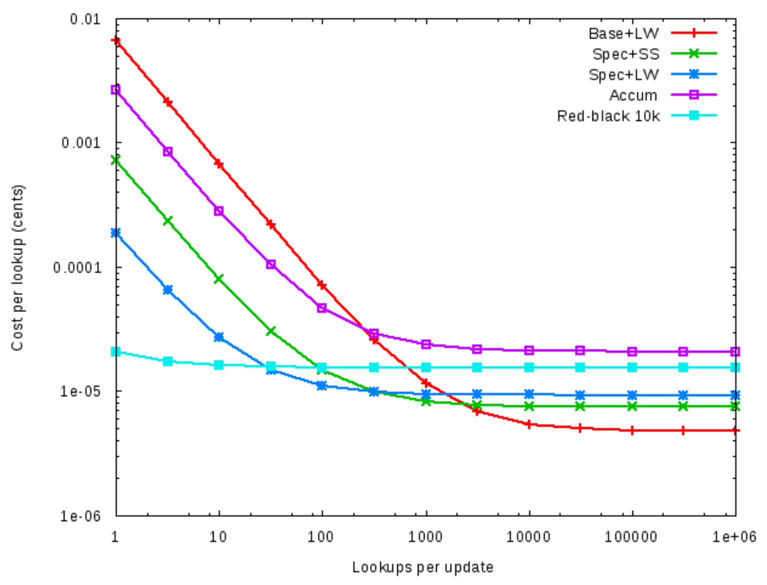
Costs per lookup on price dataset



These results

- Could not be presented without looking at costs of bandwidth and CPU time
- Constant factors matter

- Accumulators
  - Lookup proof >1kb
    - Just as big as red-black
  - Expensive updates

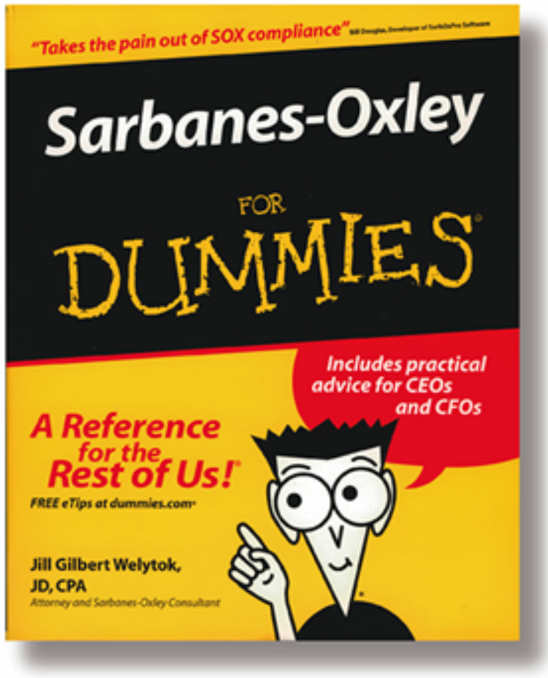


PAD designs

- Presented
  - New PAD designs
    - Improved tree PAD designs
    - New tuple PAD designs
      - Constant storage and constant sized lookup proofs
  - Comprehensive evaluation of PAD designs
    - Monetary analysis
- Focused on efficiency and the real-world

Tamper Evident Logging

Everyone has logs



"Takes the pain out of SOX compliance"

Sarbanes-Oxley FOR DUMMIES

Includes practical advice for CEOs and CFOs

A Reference for the Rest of Us!

FREE eTips at dummies.com

Jill Gilbert Welytok, JD, CPA

Attorney and Sarbanes-Oxley Consultant

The Standard of Good Practice for Information Security

ISF

2007

PCI Security Standards Council

HEALTH INSURANCE PORTABILITY and ACCOUNTABILITY ACT

HIPAA

ADMINISTRATIVE SIMPLIFICATION: PRIVACY, SECURITY, TRANSACTIONS

Current solutions

- ‘Write only’ hardware appliances
- Security depends on correct operation

- Would like cryptographic techniques
  - Logger **proves** correct behavior
  - Existing approaches too slow

Our solution

- History tree
  - Logarithmic for all operations
  - Benchmarks at >1,750 events/sec
  - Benchmarks at >8,000 audits/sec
- In addition
  - Propose new threat model
  - Demonstrate the importance of auditing

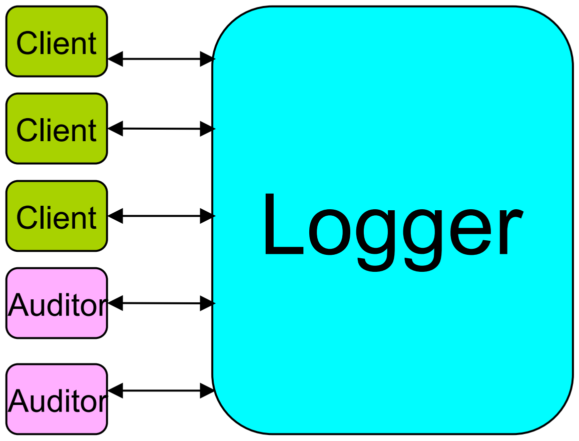
Threat model

- Strong insider attacks
  - Malicious administrator
    - Evil logger
  - Users collude with administrator

- Prior threat model
  - Forward intregity [Bellare et al 99]
  - Log tamper evident up to (unknown point), and untrusted thereafter

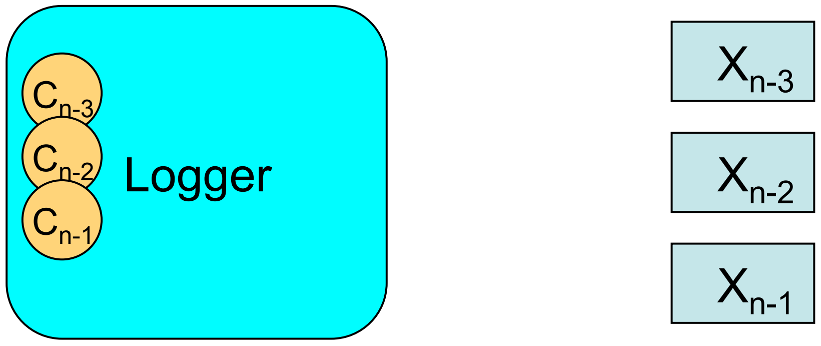
System design

- Logger
  - Stores events
  - Never trusted
- Clients
  - Little storage
  - Create events to be logged
  - Trusted only at time of event creation
  - Sends commitments to auditors
- Auditors
  - Verify correct operation
  - Little storage
  - Trusted, at least one is honest



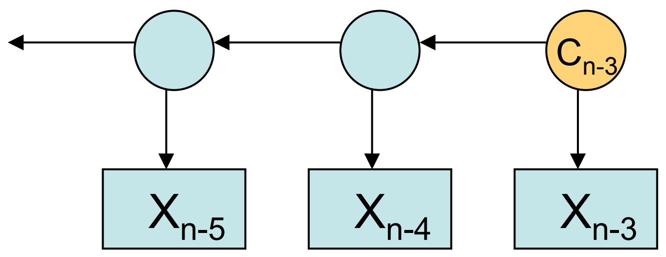
Tamper evident logging

- Events come in
  - Partially trusted clients
- Commitments go out
  - Each commits to the entire past



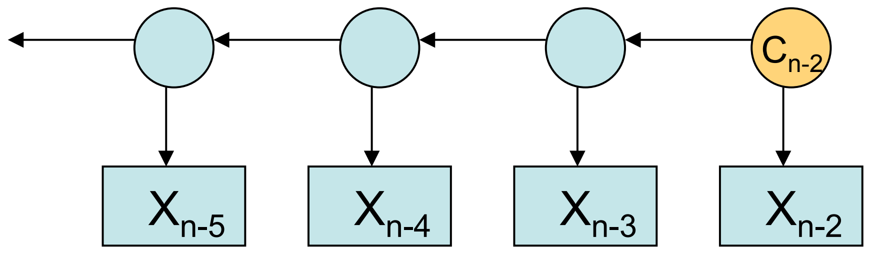
Hash chain log

- Existing approach [Kelsey and Schneier 98]
  - $C_n = H(C_{n-1} \| X_n)$
  - Logger signs C_n



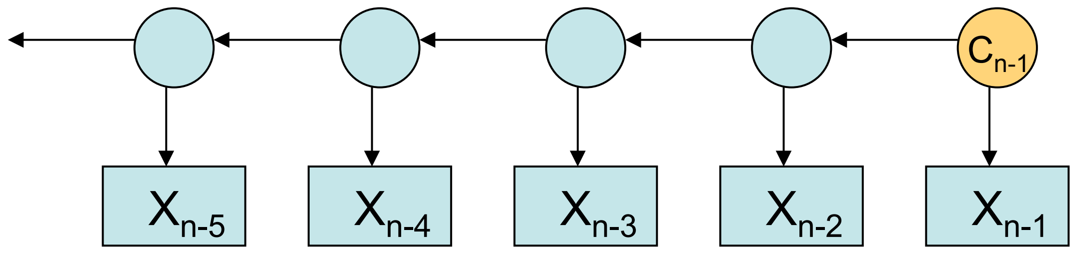
Hash chain log

- Existing approach [Kelsey,Schneier]
  - $C_n = H(C_{n-1} \| X_n)$
  - Logger signs C_n



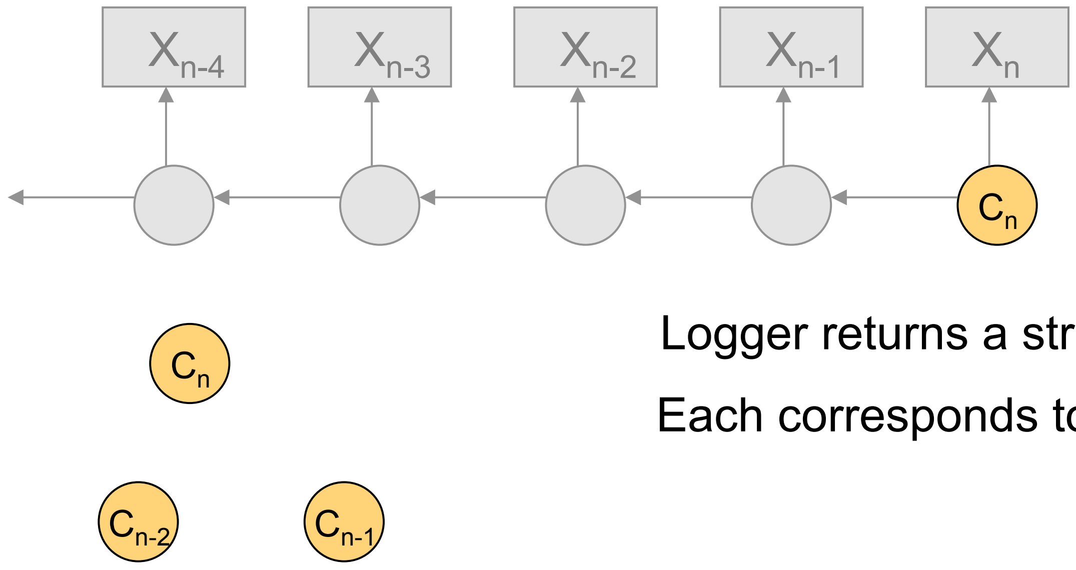
Hash chain log

- Existing approach [Kelsey,Schneier]
  - $C_n = H(C_{n-1} \| X_n)$
  - Logger signs C_n



Problem

- We don't trust the logger!

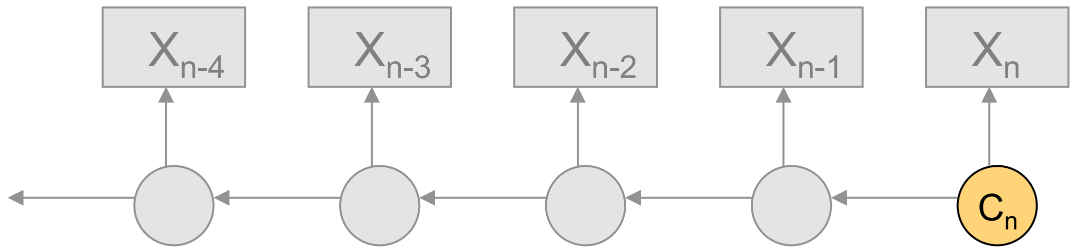


Logger returns a stream of commitments

Each corresponds to a log

Problem

- We don't trust the logger!



Does C_n really contain the just inserted $\boxed{X_n}$?

Do C_{n-2} and C_{n-1} really commit the same historical events?

Is the event at index *i* in log C_n really $\boxed{X_i}$?

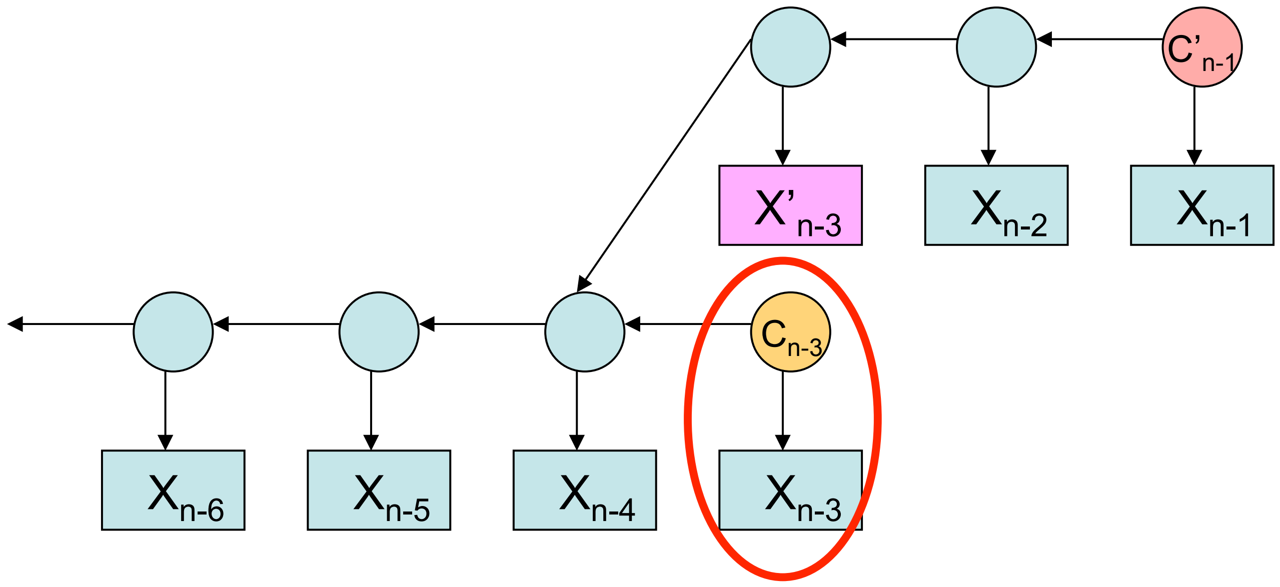
Solution

- Auditors check the returned commitments
  - For consistency $C_{n-2} \equiv C_{n-1}$
  - For correct event lookup $\boxed{X_{n-3}} \in C_{n-3}$

- Previously
  - Auditing = looking historical events
    - Assumed to infrequent
    - Performance was ignored

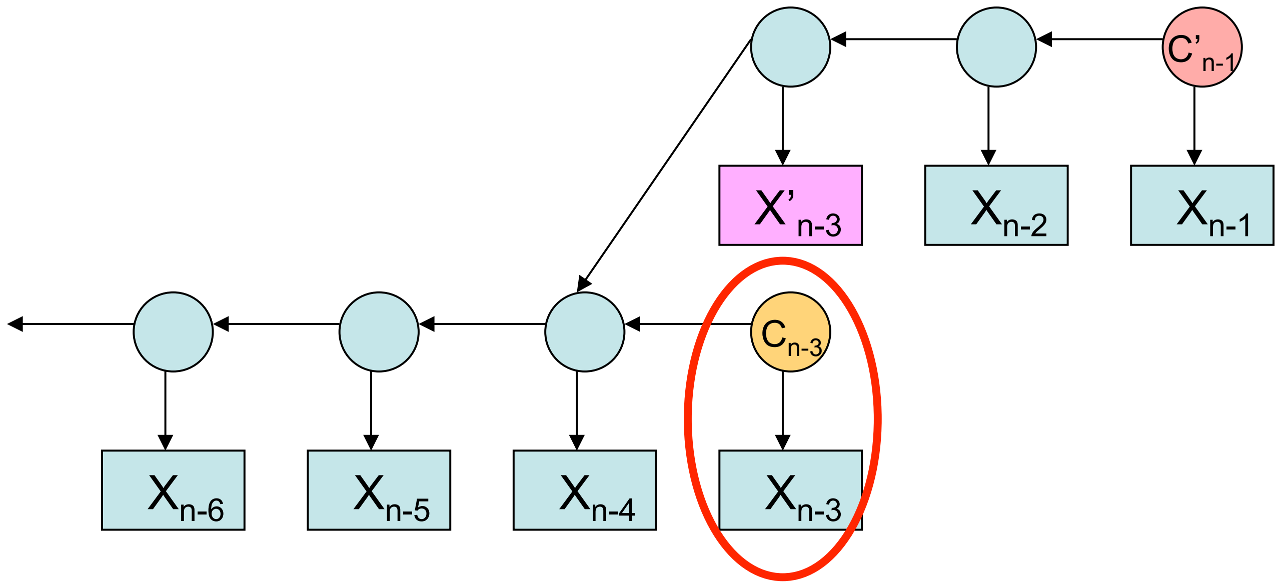
Auditing is a frequent operation

- If the logger knows this commitment will not be audited for consistency with a later commitment.



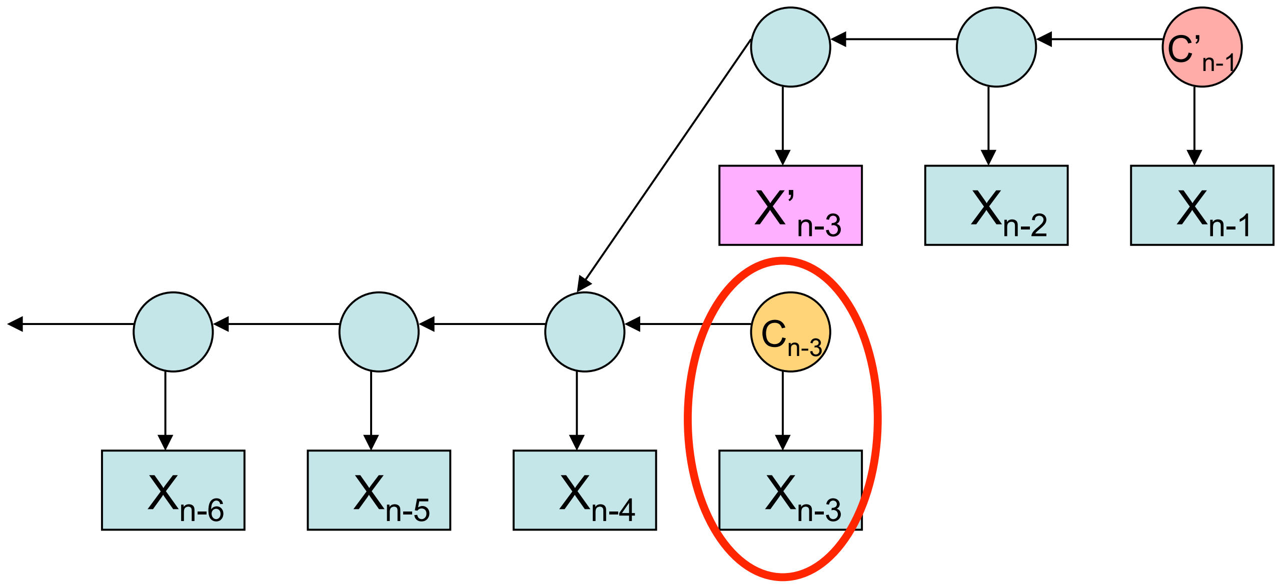
Auditing is a frequent operation

- Successfully tampered with a ‘tamper evident’ log



Auditing is a frequent operation

- Every commitment must have a non-zero chance of being audited



New paradigm

- Auditing cannot be avoided
- Audits should occur
  - On every event insertion
  - Between commitments returned by logger
- How to make inserts *and audits* cheap
  - CPU
  - Communications complexity
  - Storage

Two kinds of audits

- Membership auditing $\boxed{X_i} \in C_n$
  - Verify proper insertion
  - Lookup historical events

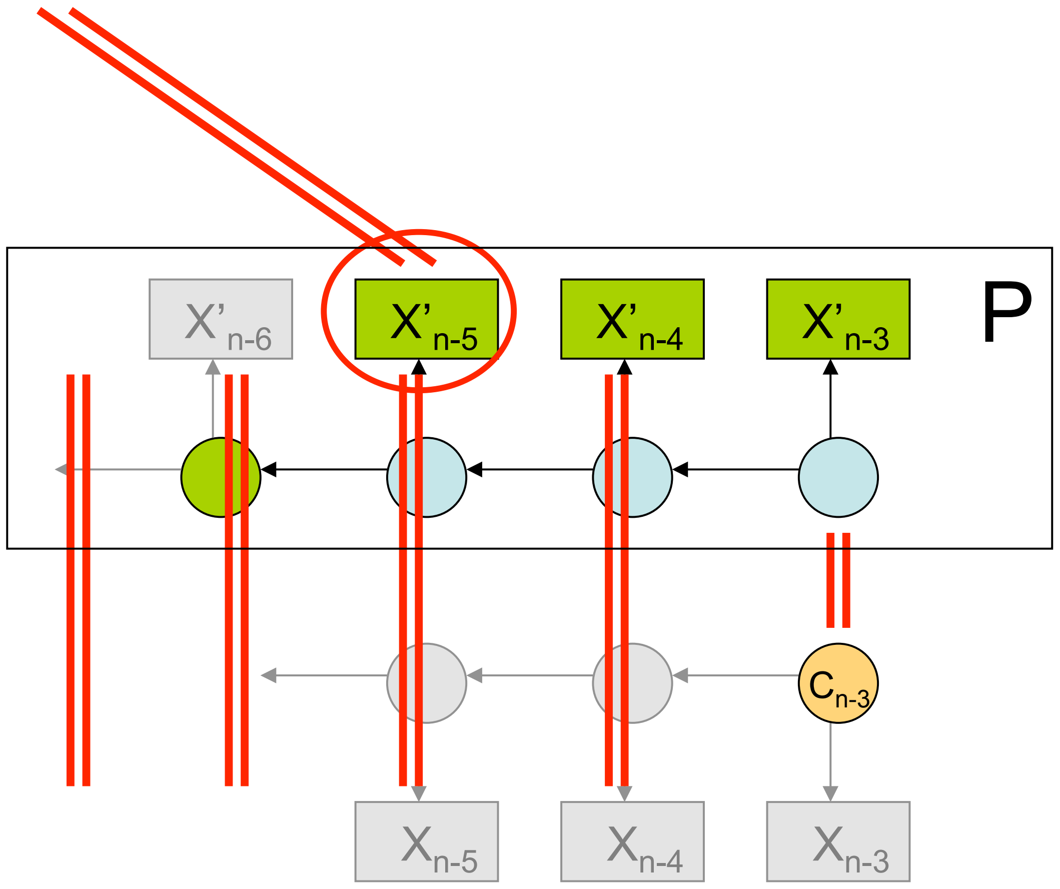
- Incremental auditing $C_i \equiv C_n$
  - Prove consistency between two commitments

Membership auditing a hash chain

- Is $\boxed{X_{n-5}} \in c_{n-3}$?

Membership auditing a hash chain

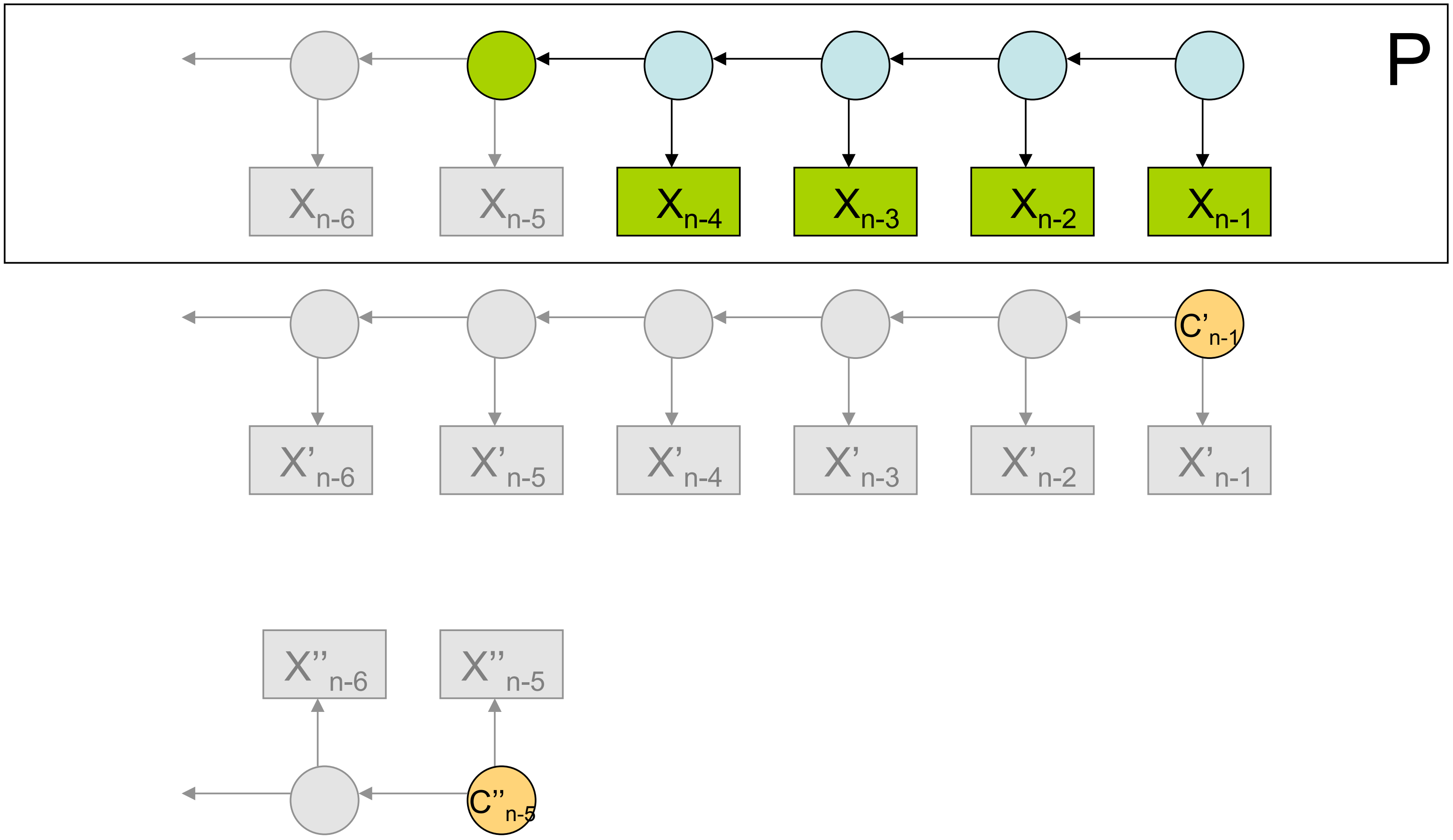
- Is $\boxed{X_{n-5}} \in C_{n-3}$?



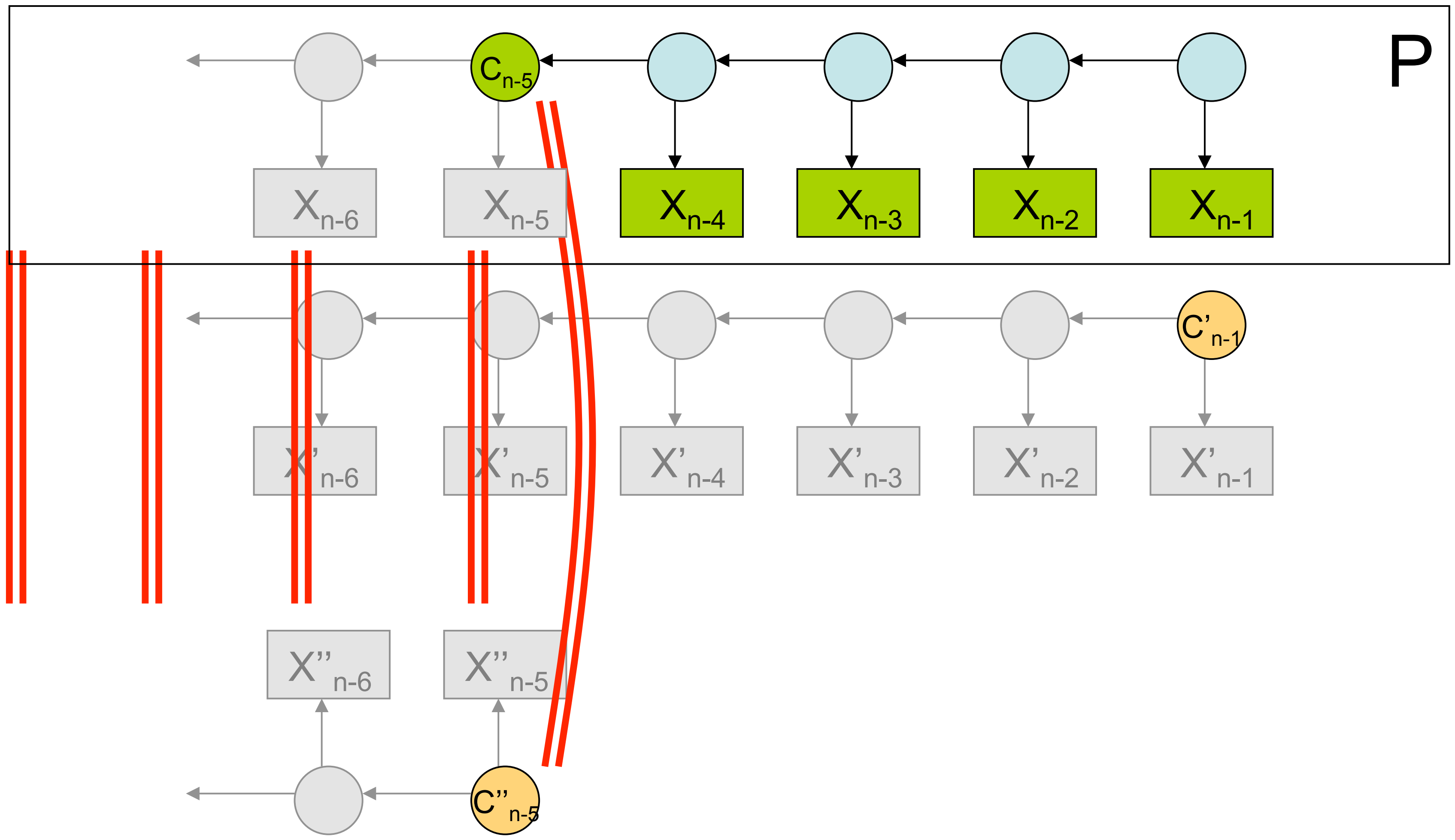
Incremental auditing a hash chain

- Are $C''_{n-5} \equiv C'_{n-1}$?

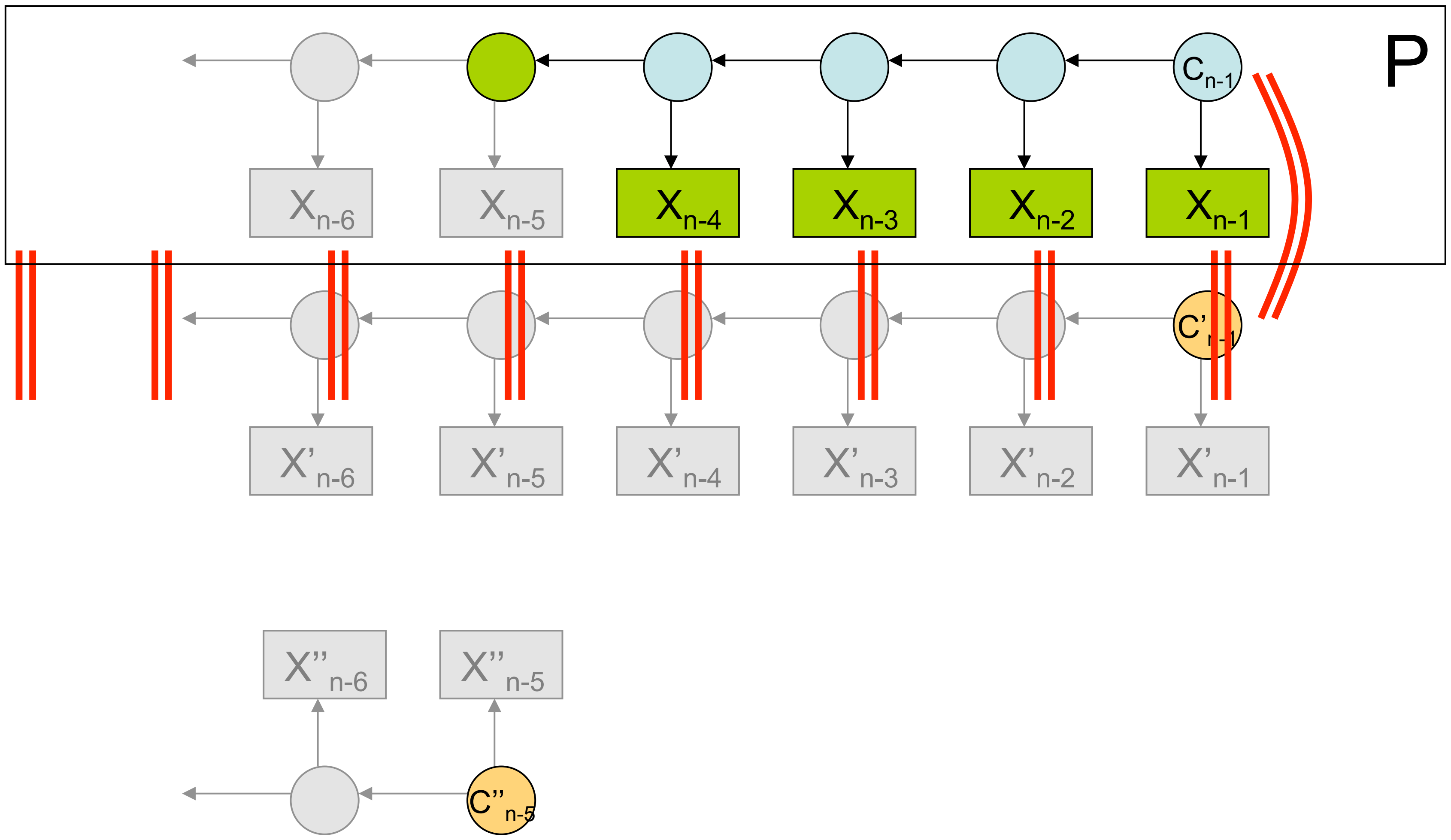
Incremental auditing a hash chain



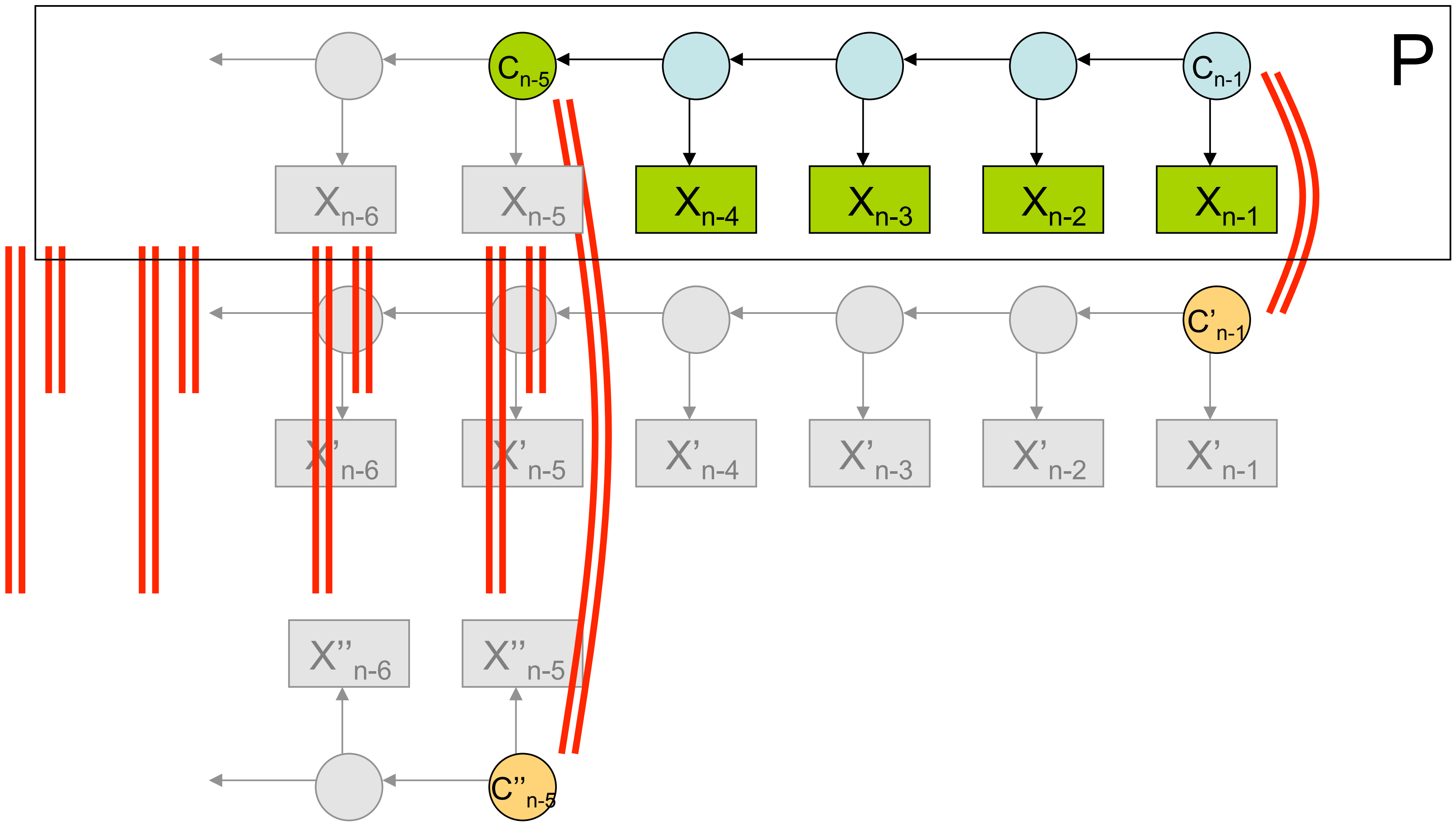
Incremental auditing a hash chain



Incremental auditing a hash chain



Incremental auditing a hash chain



Existing tamper evident log designs

- Hash chain [Kelsey and Schneier 98]
  - Auditing is linear time
  - Historical lookups
    - Very inefficient

- Skiplist history [Maniatis and Baker 02]
  - Auditing is still linear time
  - O(log n) historical lookups

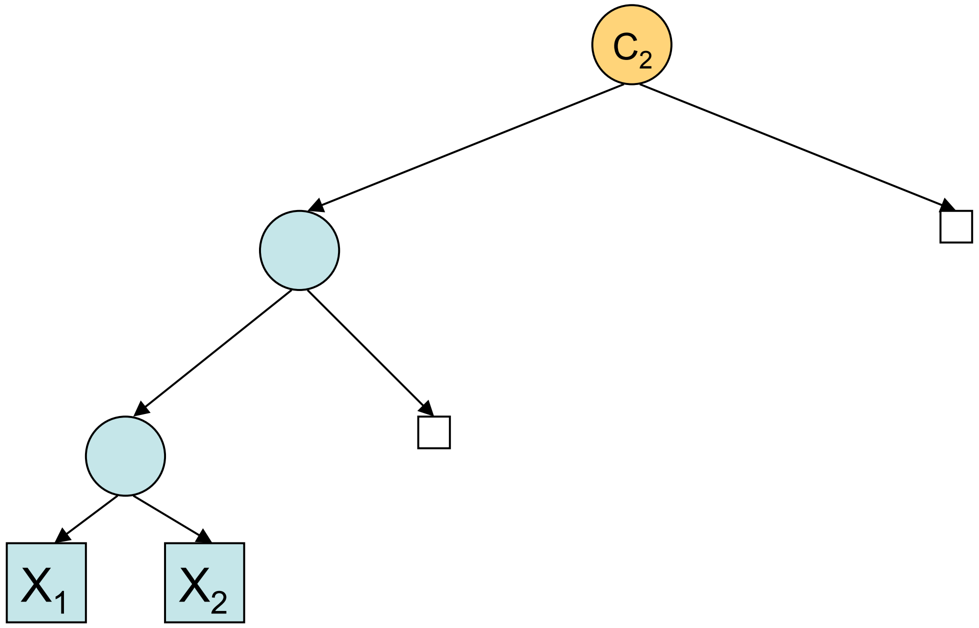
Our solution

- History tree
  - O(log n) instead of O(n) for all operations
  - Variety of useful features
    - Write-once append-only storage format
    - Predicate queries + safe deletion
    - May probabilistically detect tampering
      - Auditing random subset of events
      - Not beneficial for skip-lists or hash chains

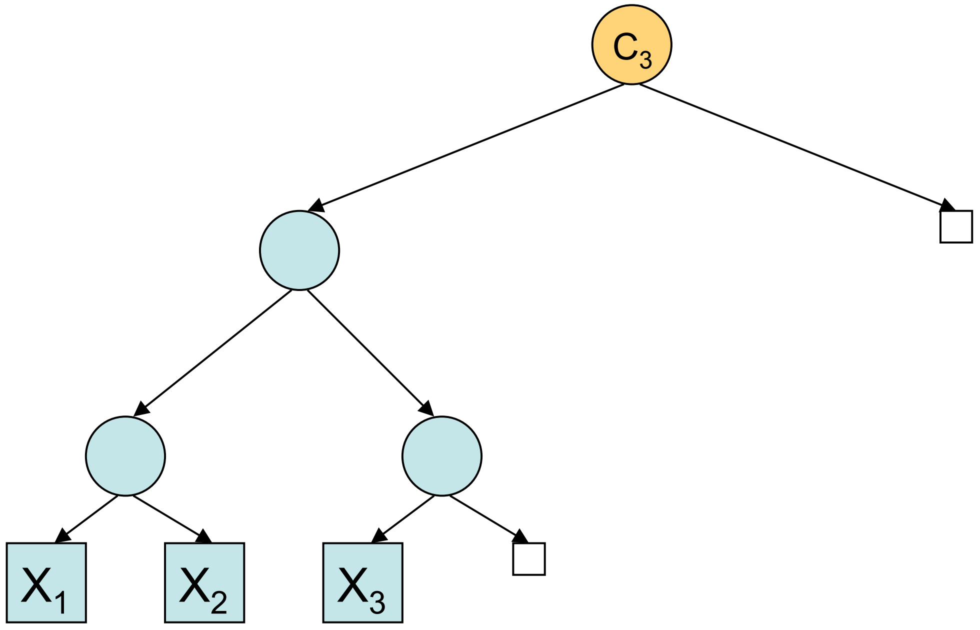
History tree

- Merkle binary tree
  - Events stored on leaves
  - Logarithmic path length
    - Random access
  - Permits reconstruction of past version and past commitments

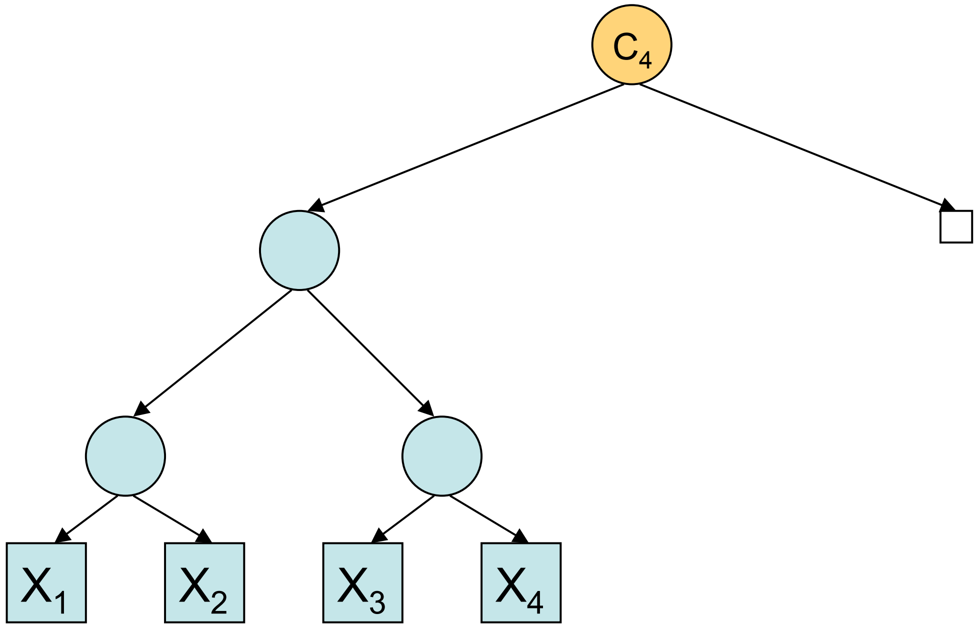
History tree



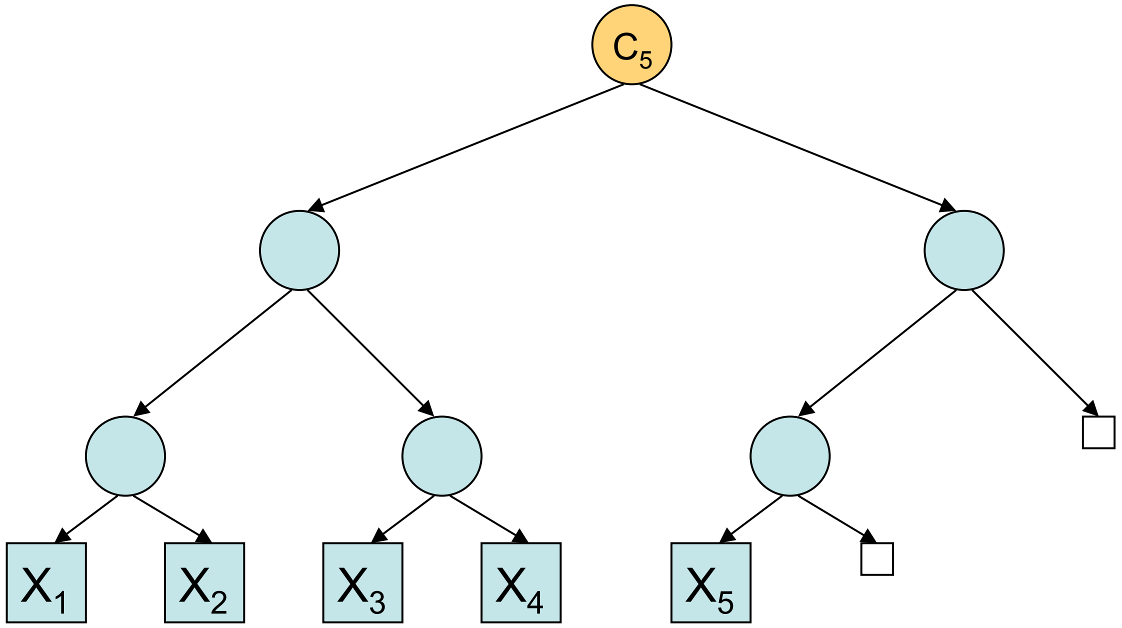
History tree



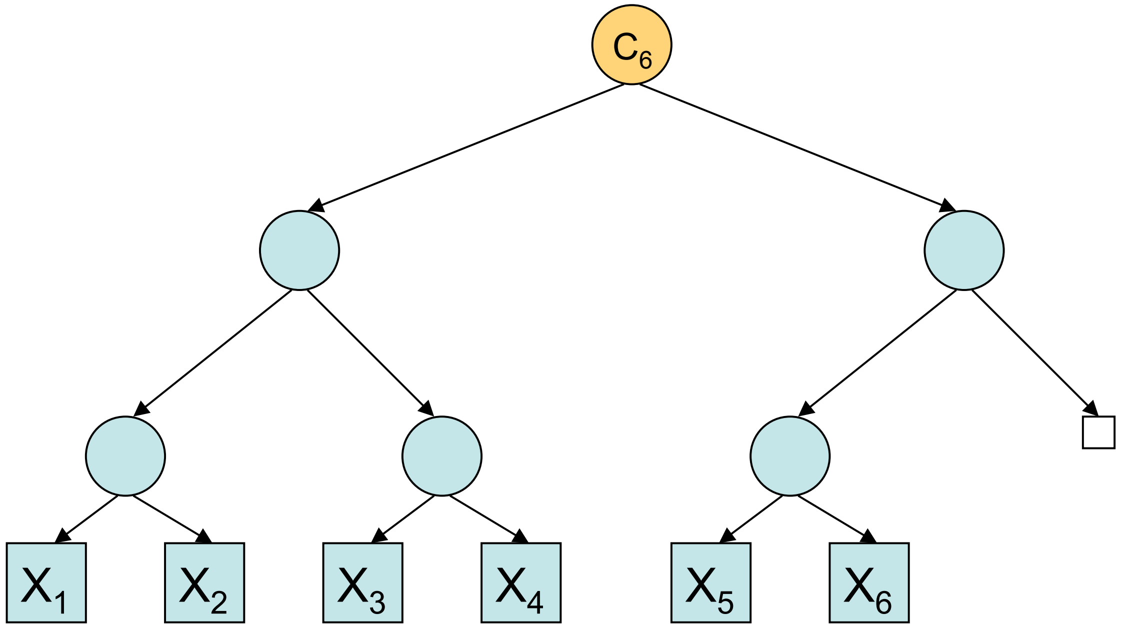
History tree



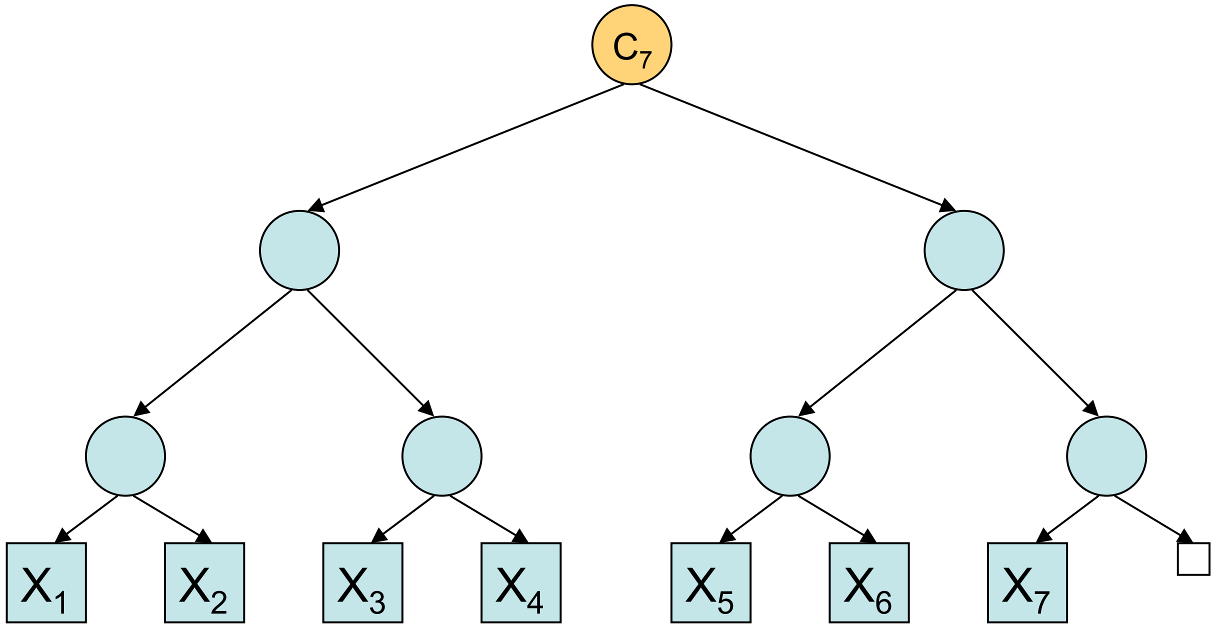
History tree



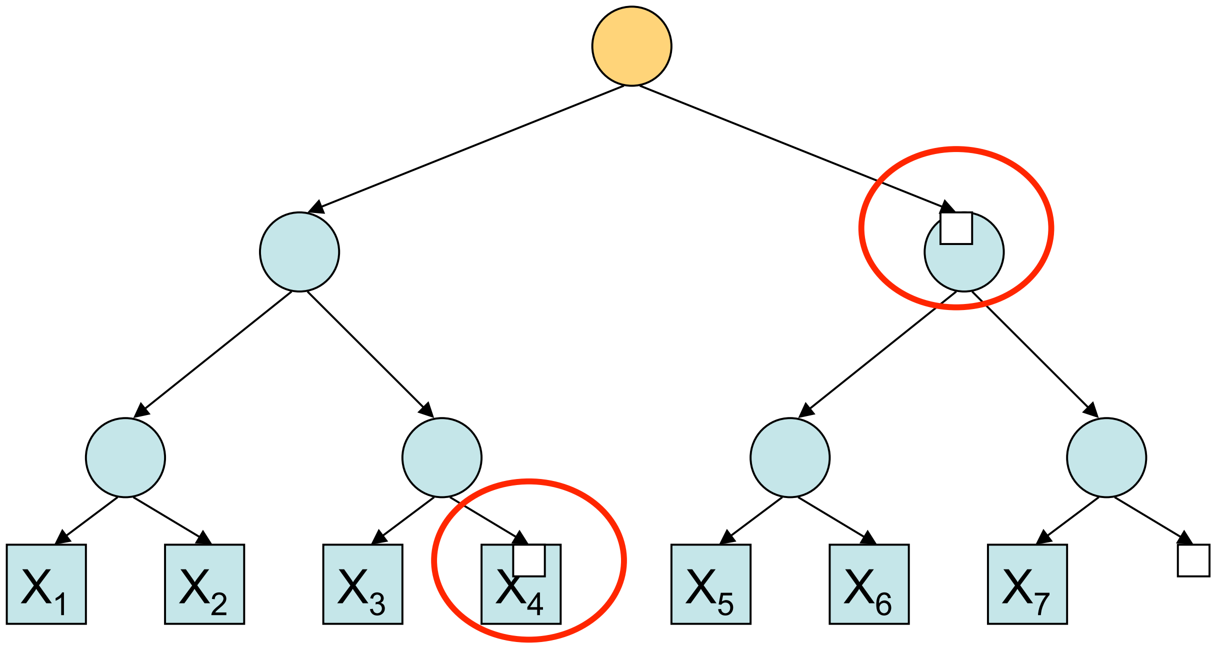
History tree



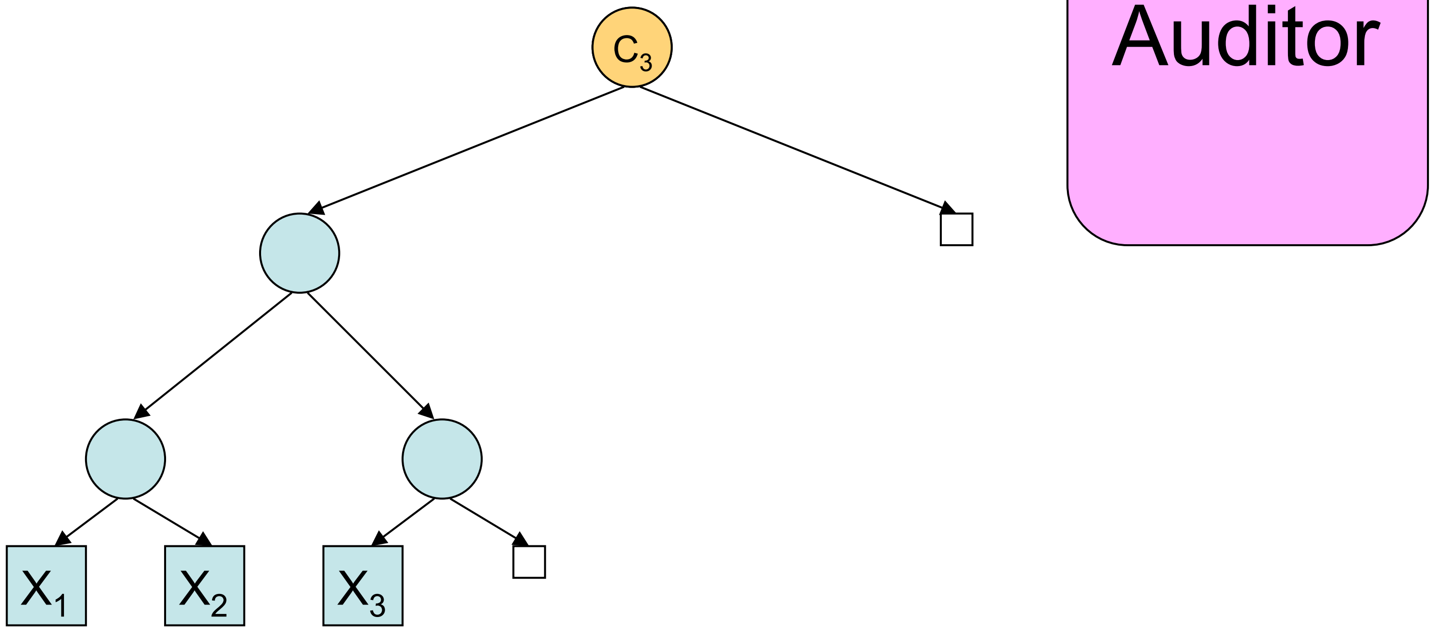
History tree

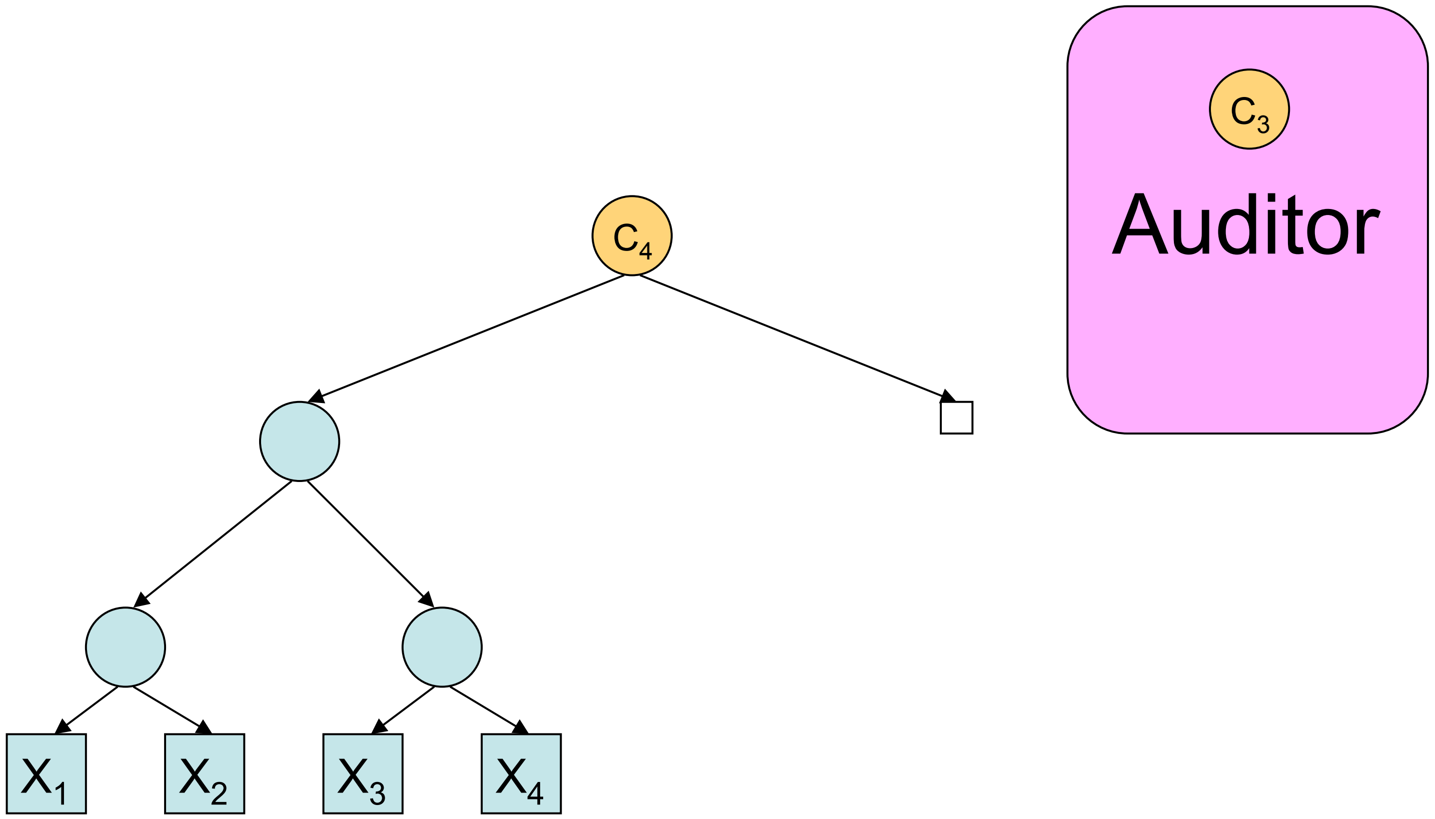


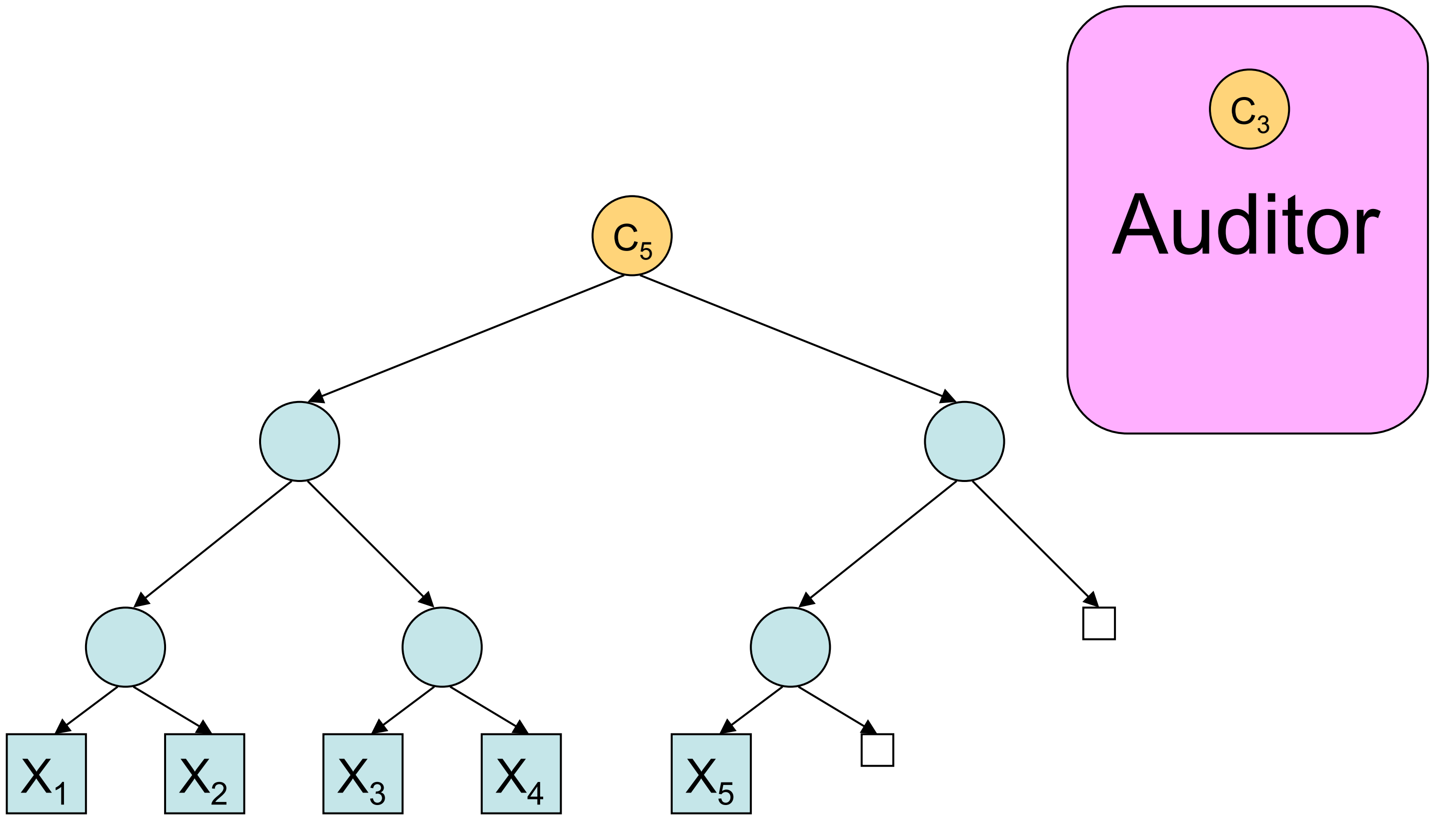
History tree

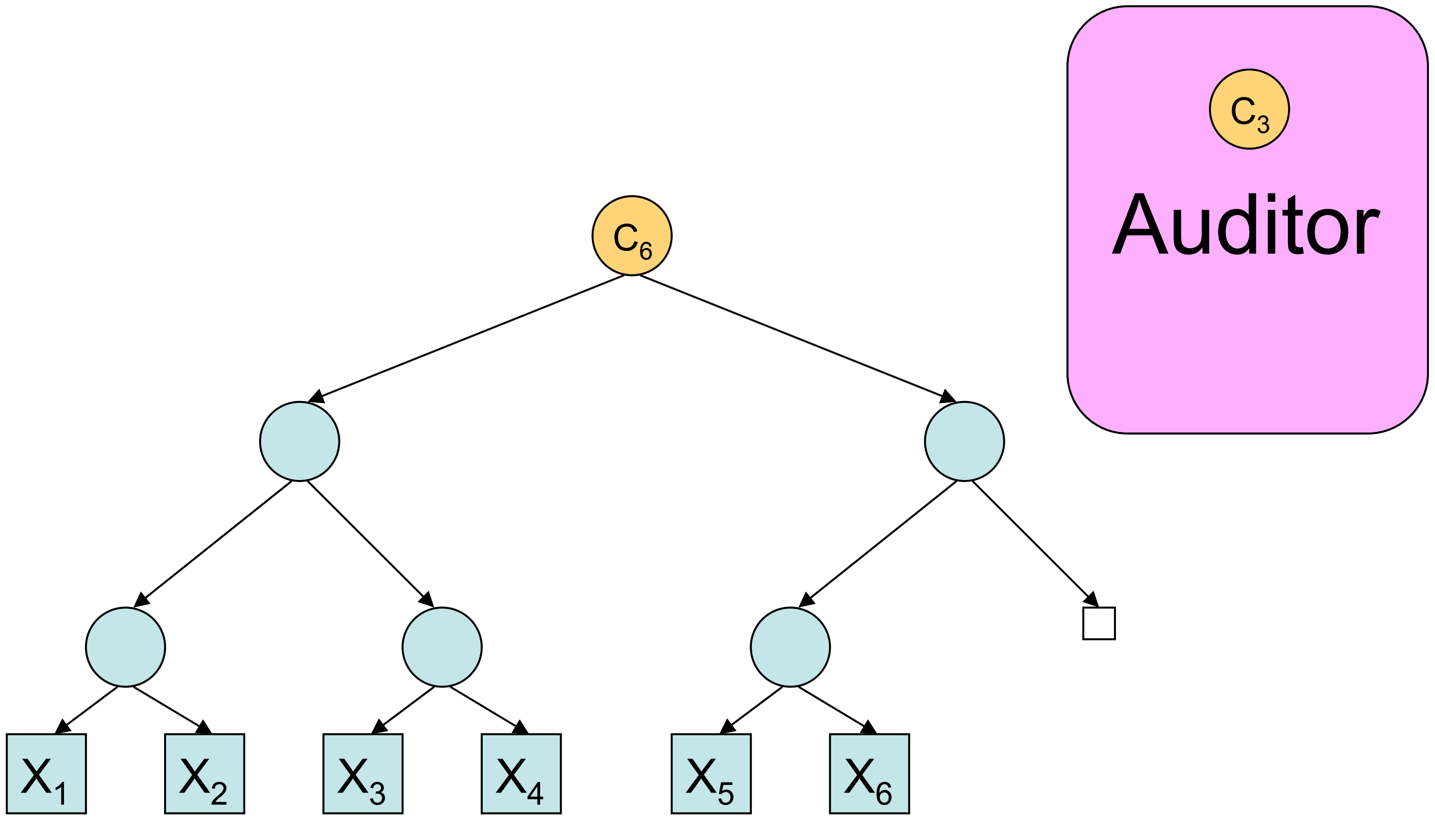


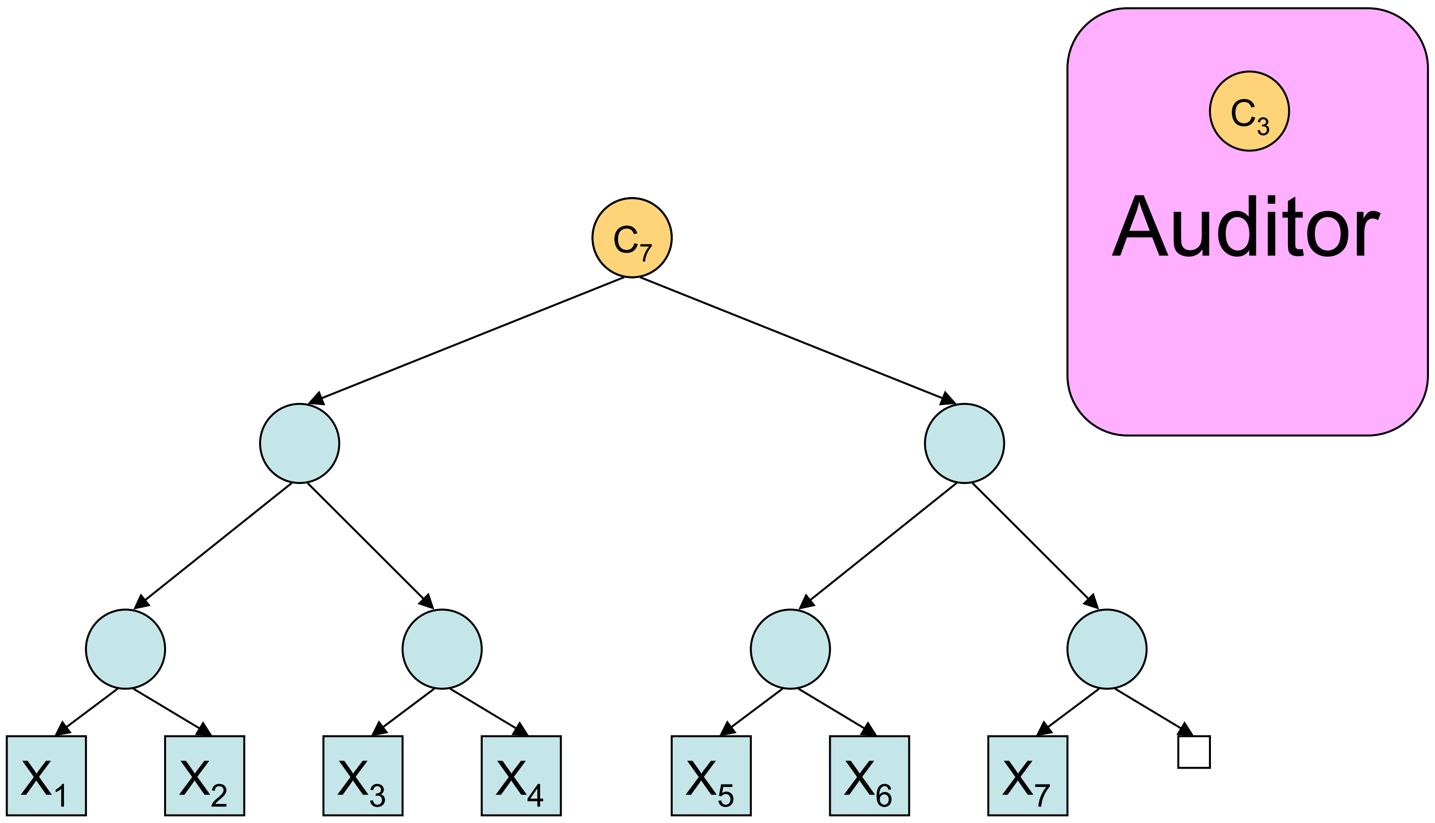
Incremental auditing


C_3
X_1
X_2
X_3
Auditor


C4
X1
X2
X3
X4
C3
Auditor

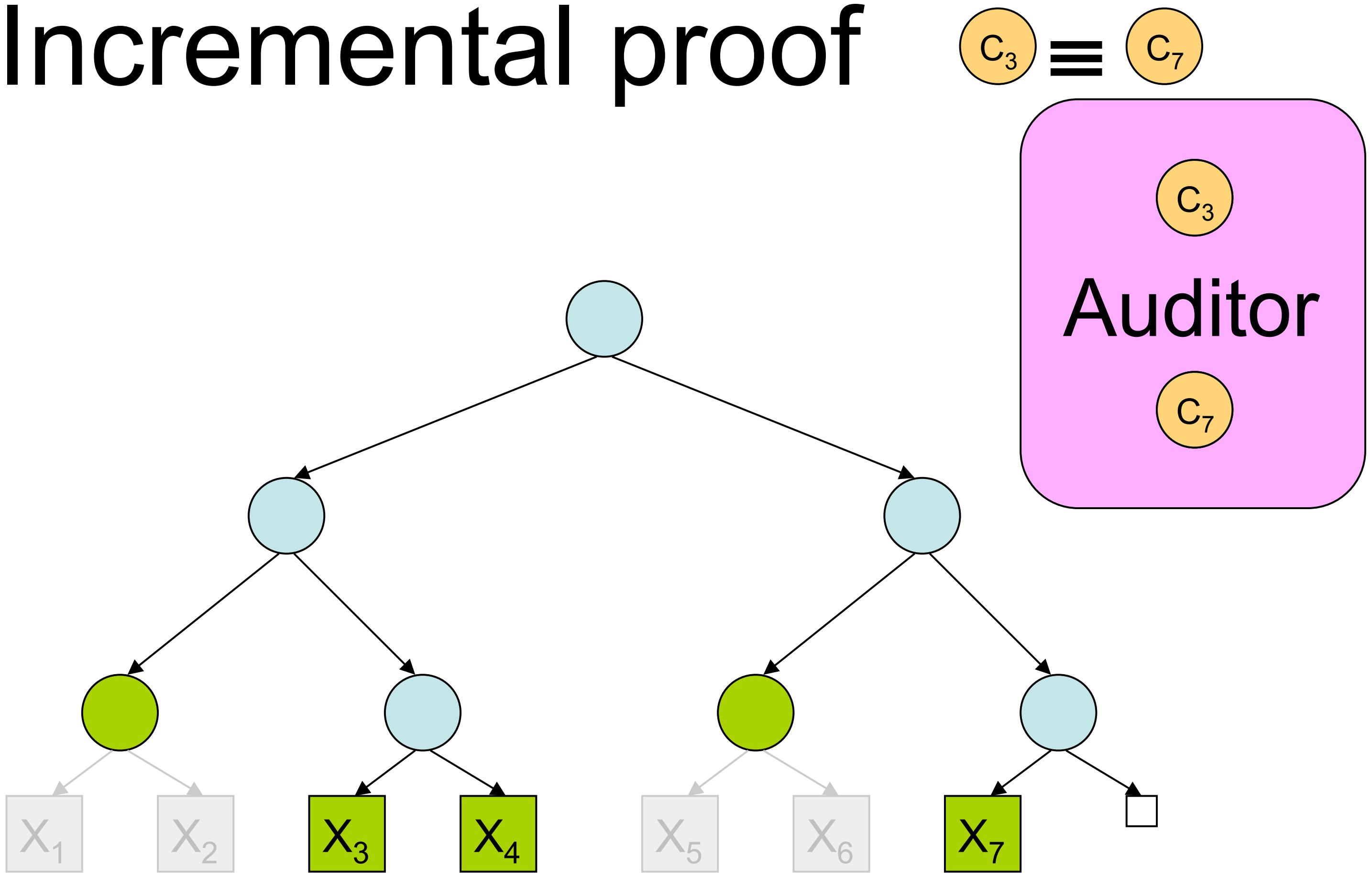

C_5
X_1
X_2
X_3
X_4
X_5
C_3
Auditor


C$_6$
X$_1$
X$_2$
X$_3$
X$_4$
X$_5$
X$_6$
C$_3$
Auditor

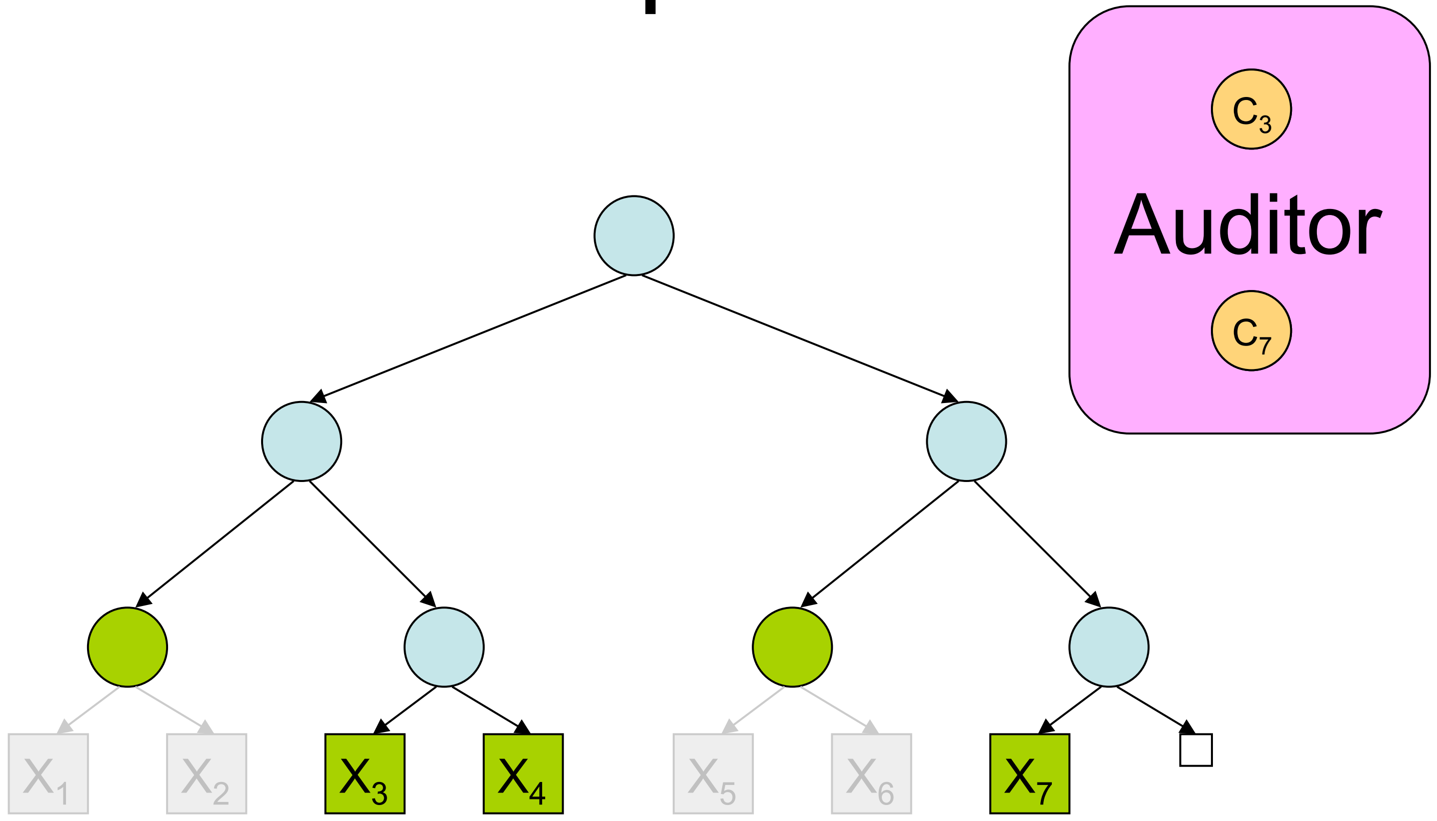


c_7
c_3
Auditor
X_1
X_2
X_3
X_4
X_5
X_6
X_7

Incremental proof

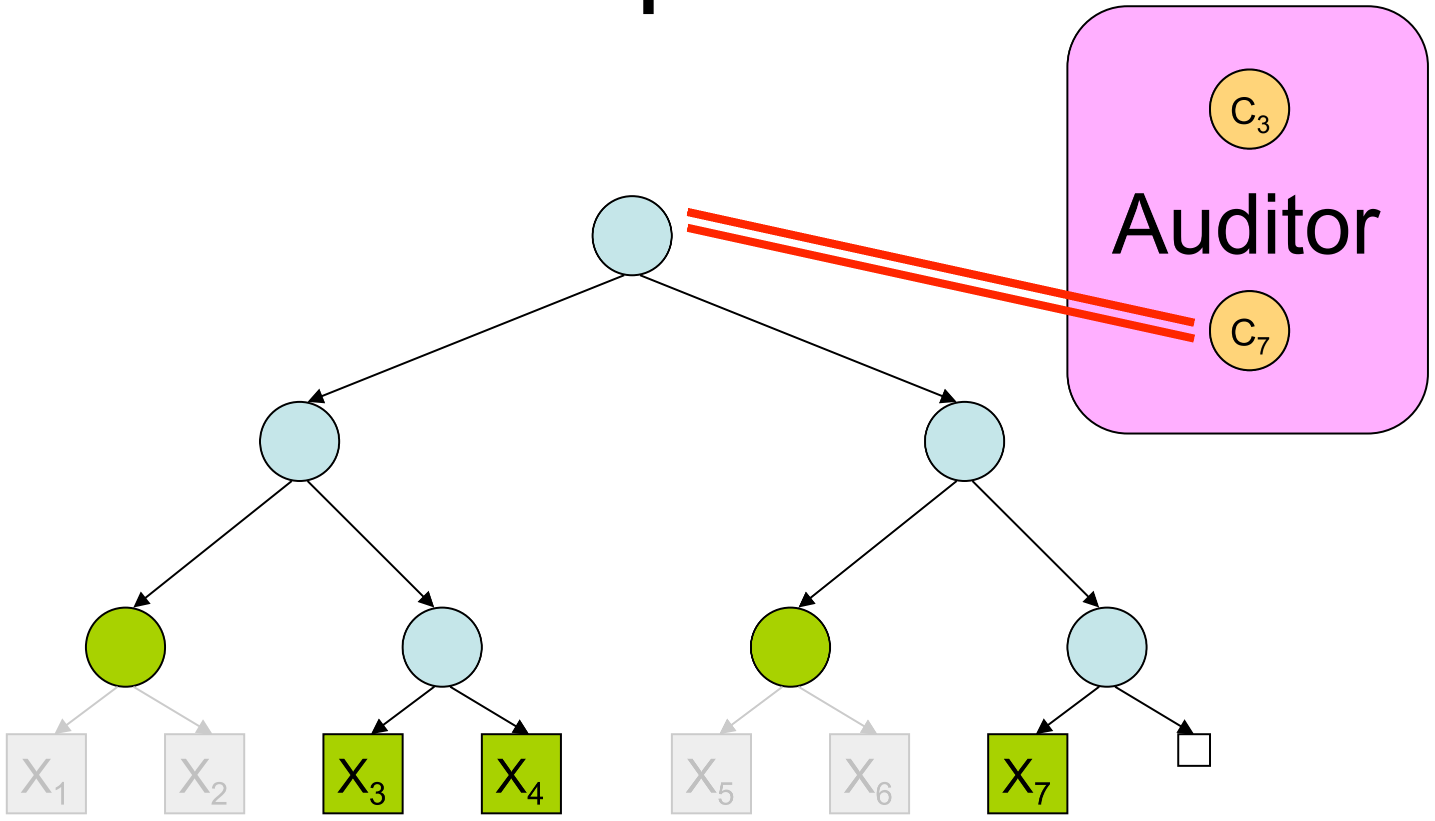


Incremental proof $C_3 \equiv C_7$



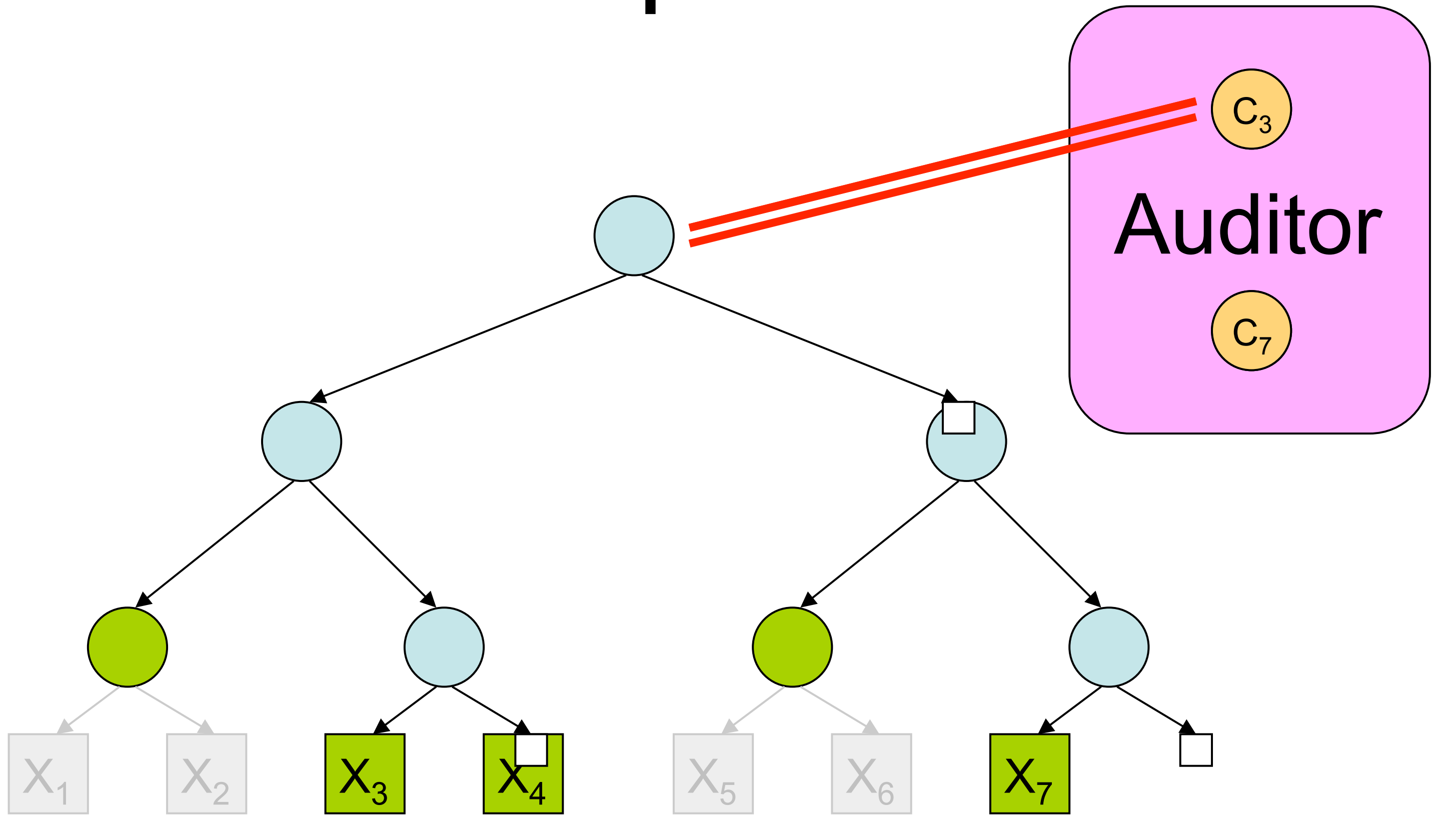
- P is consistent with C_7
- P is consistent with C_3
- Therefore C_7 and C_3 are consistent.

Incremental proof $C_3 \equiv C_7$



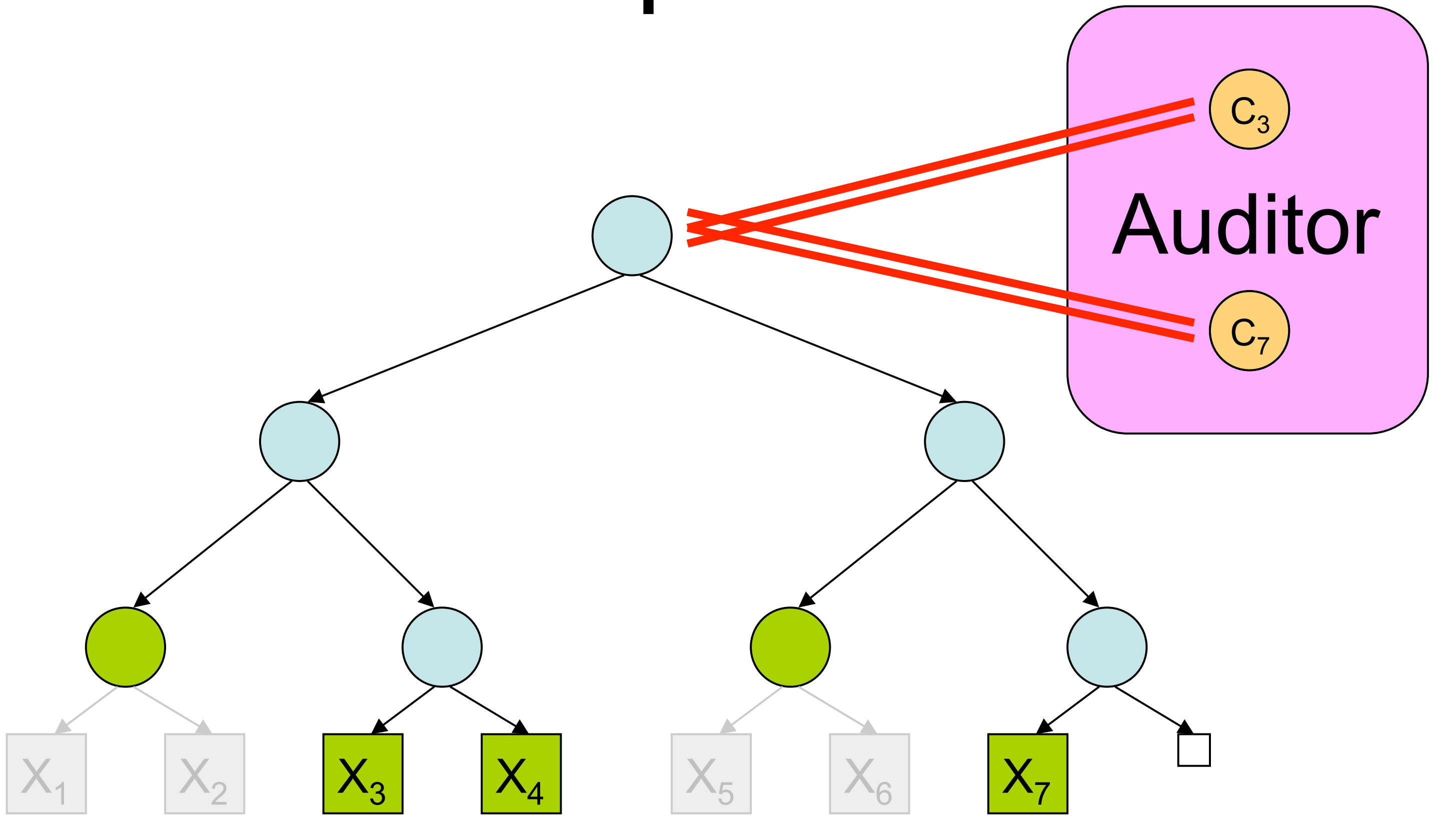
- P is consistent with C_7
- P is consistent with C_3
- Therefore C_7 and C_3 are consistent.

Incremental proof $C_3 \equiv C_7$



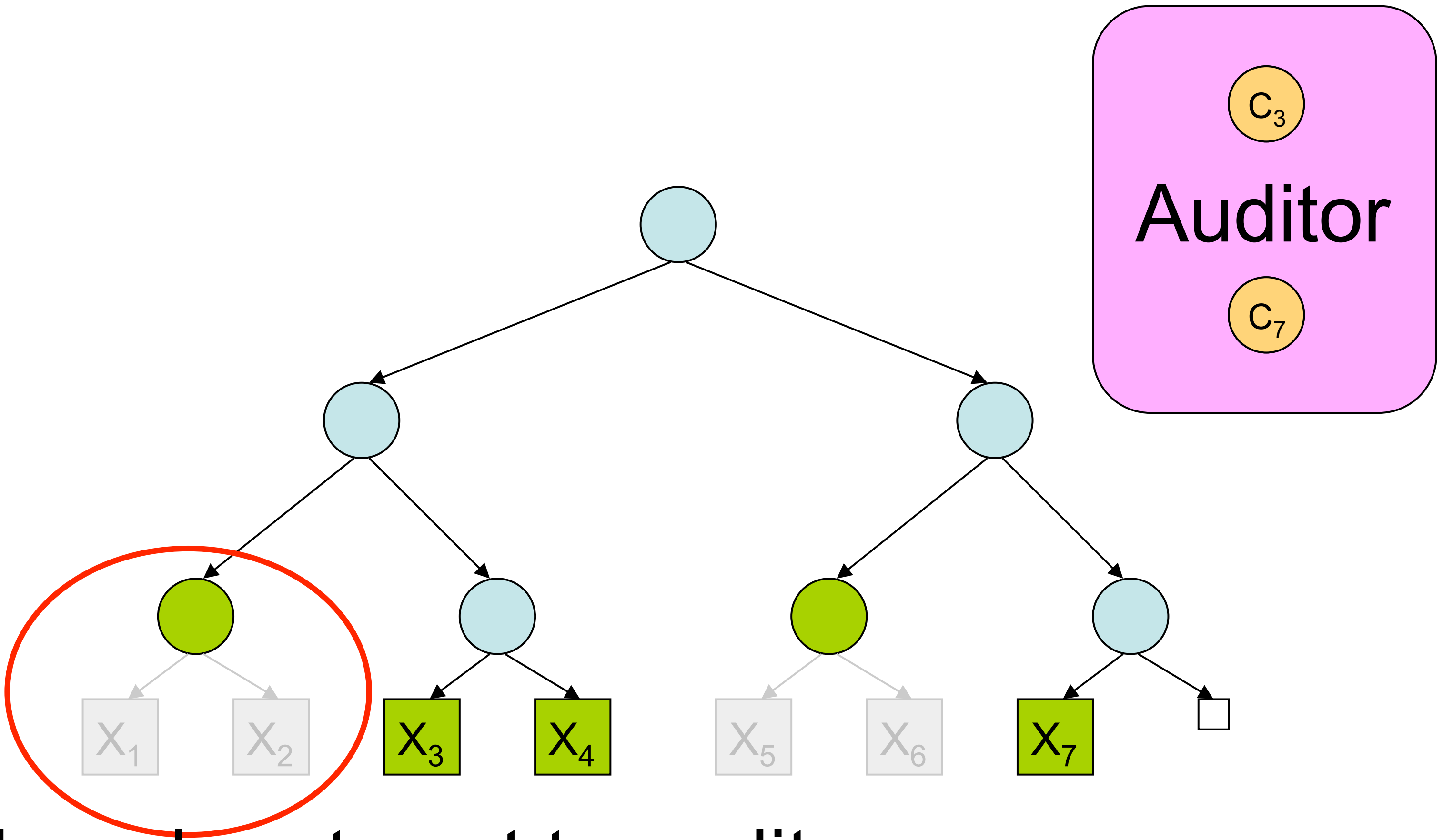
- P is consistent with C_7
- P is consistent with C_3
- Therefore C_7 and C_3 are consistent.

Incremental proof $C_3 \equiv C_7$



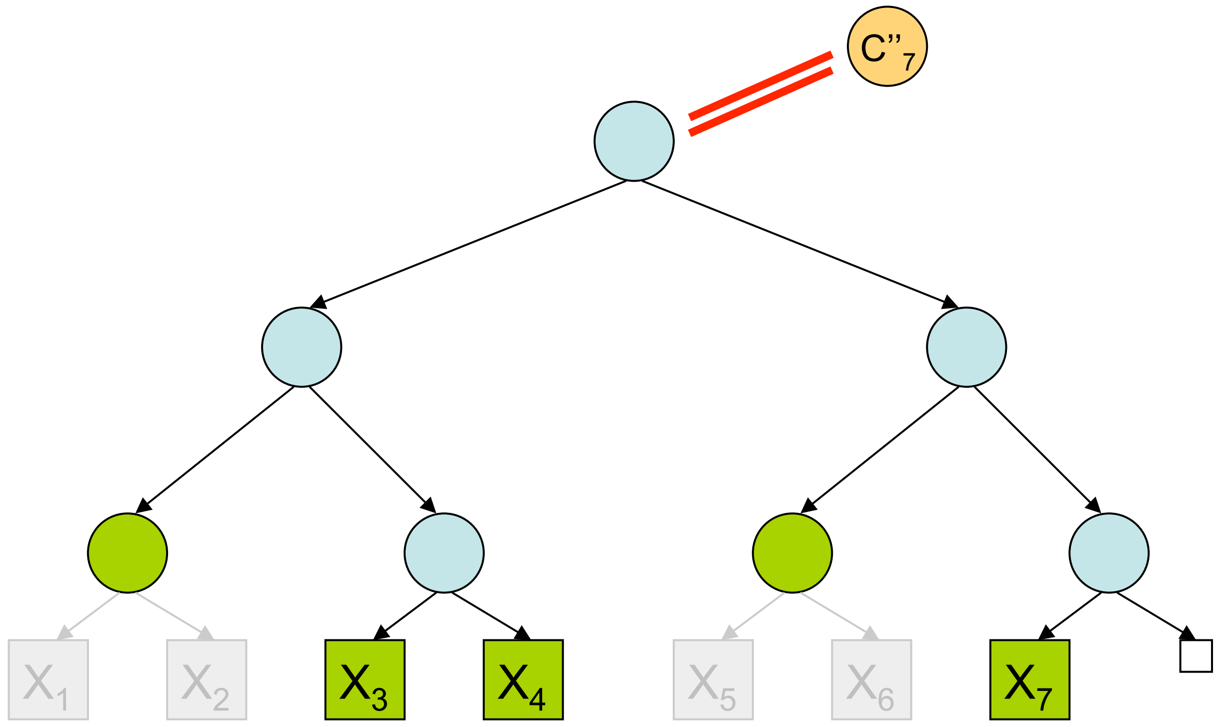
- P is consistent with C_7
- P is consistent with C_3
- Therefore C_7 and C_3 are consistent.

Pruned subtrees



- Although not sent to auditor
  - Fixed by hashes above them
  - c_3 , c_7 fix the same (unknown) events

Membership proof that $X_3 \in C''_7$

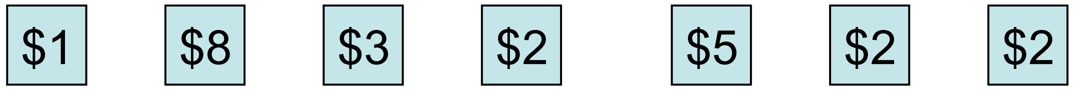


- Verify that C''_7 has the same contents as P
- Read out event X_3

Merkle aggregation

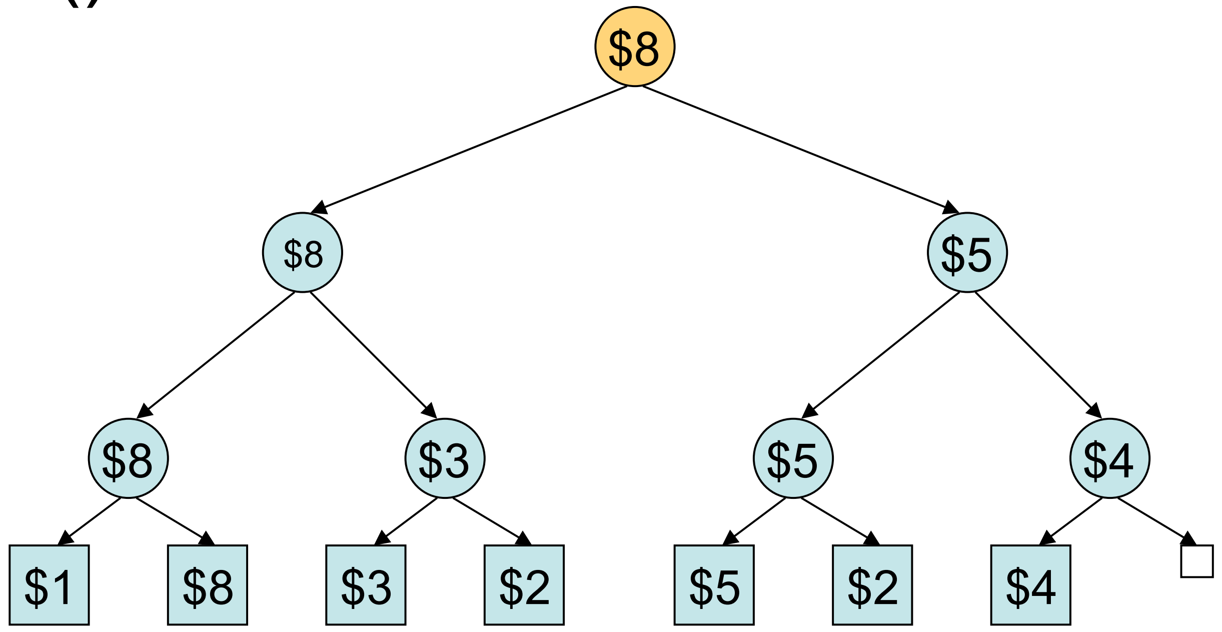
Merkle aggregation

- Annotate events with attributes



Aggregate them up the tree

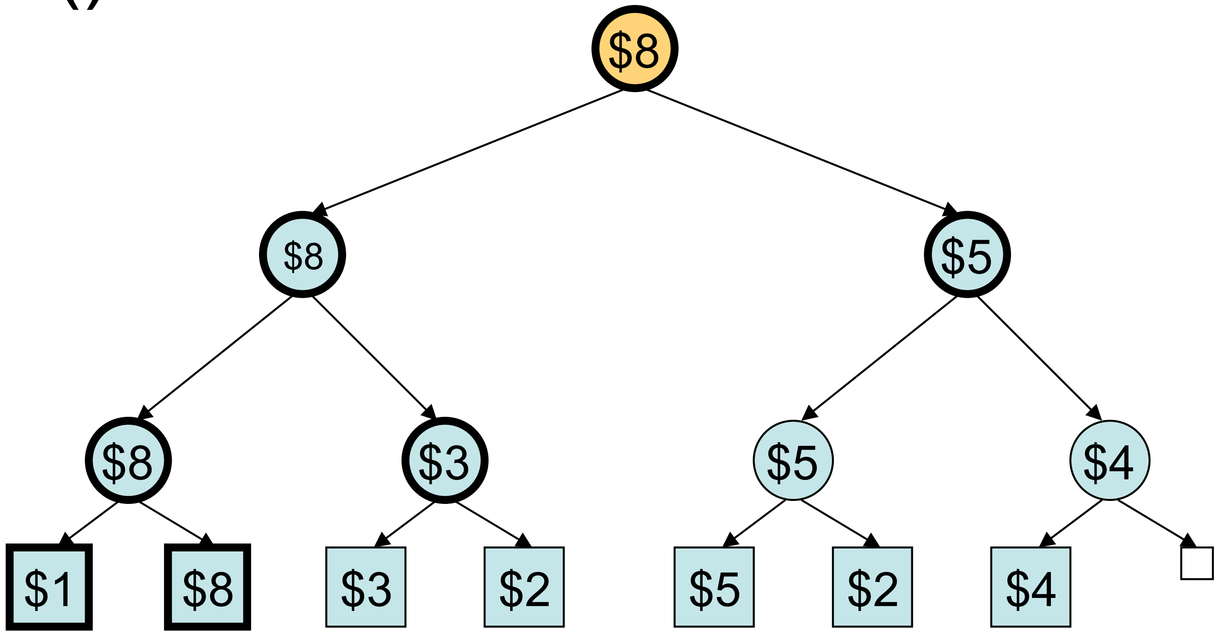
- Max()



Included in hashes and checked during audits

Querying the tree

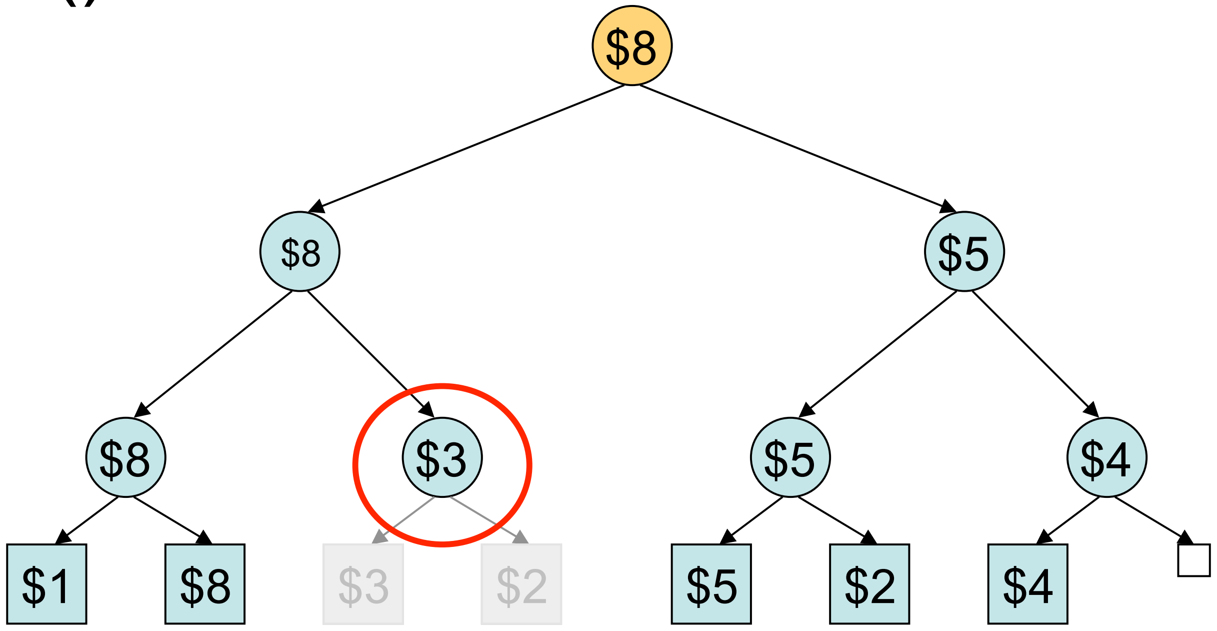
- Max()



Find all transactions over $6

Safe deletion

- Max()



Authorized to delete all transactions under $4

Merkle aggregation is flexible

- Many ways to map events to attributes
  - Arbitrary computable function
- Many attributes
  - Timestamps, dollar values, flags, tags
- Many aggregation strategies

  +, *, min(), max(), ranges, and/or, Bloom filters

Generic aggregation

- ([illegible],[illegible],[illegible])
  - [illegible] : Type of attributes on each node in history
  - [illegible] : Aggregation function
  - [illegible] : Maps an event to its attributes
- For any predicate P, as long as:
  - P(x) OR P(y) IMPLIES P(x[illegible]y)
  - Then:
    - Can query for events matching P
    - Can safe-delete events not matching P

Evaluating the history tree

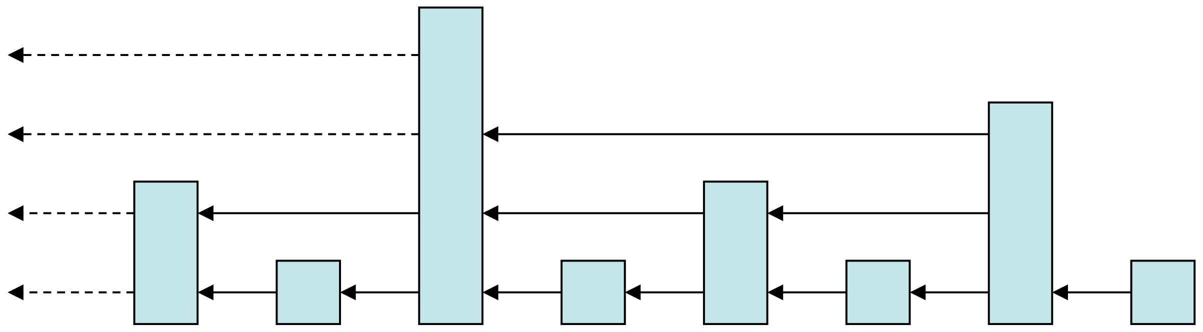
- Big-O performance
- Syslog implementation

Big-O performance

	$c_j \equiv c_i$	$X_i \in c_j$	Insert
History tree	O(log *n*)	O(log *n*)	O(log *n*)
Hash chain	O(*j*-*i*)	O(*j*-*i*)	O(1)
Skip-list history [Maniatis and Baker]	O(*j*-*i*) or O(*n*)	O(log *n*) or O(*n*)	O(1)

Skiplist history [Maniatis and Baker]

- Hash chain with extra links
  - Extra links cannot be trusted without auditing
    - Checking them
      - Best case: only events since last audit
      - Worst case: examining the whole history
  - If extra links are valid
    - Using them for historical lookups
      - O(log n) time and space



Syslog implementation

- We ran 80-bit security level
  - 1024 bit DSA signatures
  - 160 bit SHA-1 Hash
- We recommend 112-bit security level
  - 224 bit ECDSA signatures
    - 66% faster
  - SHA-224 (Truncated SHA-256)
    - 33% slower

- [NIST SP800-57 Part 1, Recommendations for Key Magament – Part 1: General (Revised 2007)]

Syslog implementation

- Syslog
  - Trace from Rice CS departmental servers
  - 4M events, 11 hosts over 4 days, 5 attributes per event
    - Repeated 20 times to create 80M event trace

Syslog implementation

- Implementation
  - Hybrid C++ and Python
  - Single threaded
  - MMAP-based append-only write-once storage for log
  - 1024-bit DSA signatures and 160-bit SHA-1 hashes
- Test platform
  - 2.4 GHz Core 2 Duo (circa 2007) desktop machine
  - 4GB RAM

Performance

- Insert performance: 1,750 events/sec
  - 83.3% : Sign commitment
- Auditing performance
  - With locality (last 5M events)
    - 10,000-18,000 incremental proofs/sec
    - 8,600 membership proofs/sec
  - Without locality
    - 30 membership proofs/sec
  - < 4,000 byte self-contained proof size

Improving performance

- Increasing audit throughput above
  - 8,000 audits/sec
- Increasing insert throughput above
  - 1,750 inserts/sec

Increasing audit throughput

- Audits require read-only access to the log
  - Trivially offloaded to additional cores

- For infinite scalability
  - May replicate the log server
    - Master assigns event indexes
    - Slaves build history tree locally

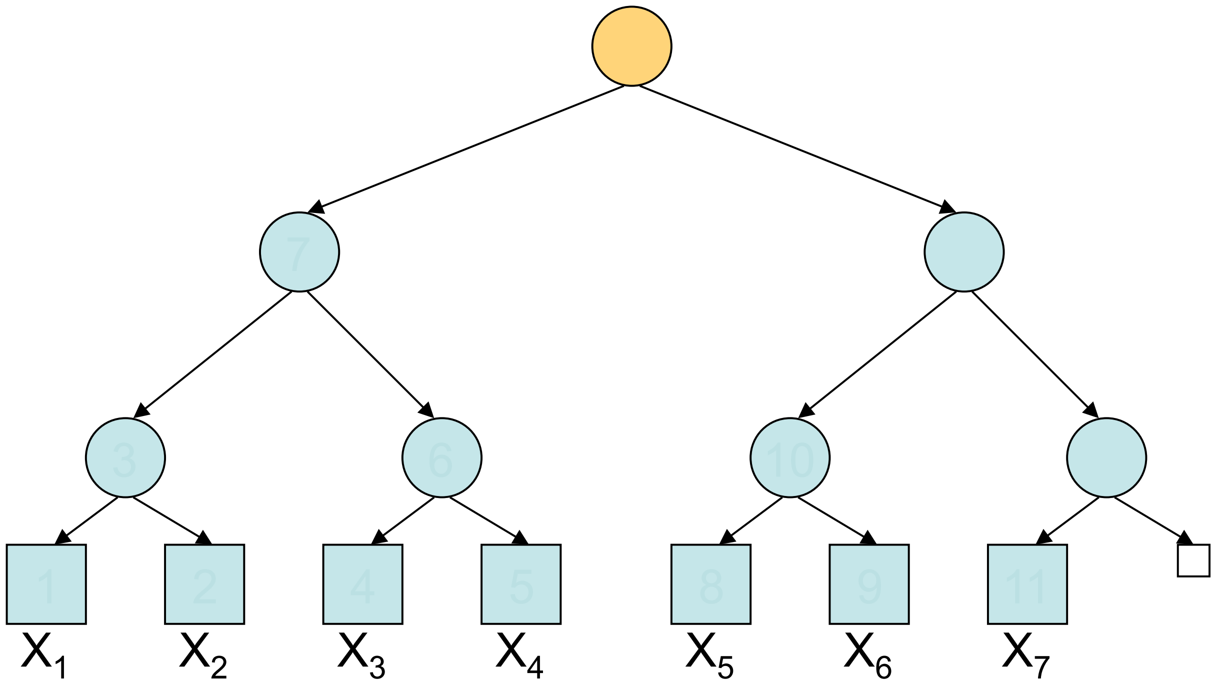
Increasing insert throughput

- Public key signatures are slow
  - 83% of runtime
- Three easy optimization
  - Sign only some commitments
  - Use faster signatures
  - Offload to other hosts
    - Increase throughput to 10k events/sec

More concurrency with replication

- Processing pipeline:
  - Inserting into history tree
    - O(1). Serialization point
    - Fundamental limit
      - Must be done on each replica
      - 38,000 events/sec using only one core
  - Commitment or proofs generation
    - O(log n).
  - Signing commitments
    - O(1), but expensive. Concurrently on other hosts

Storing on secondary storage



- Nodes are frozen (no longer ever change)
  - In post-order traversal
    - Static order
  - Map into an array

Tamper-evident logging

- New paradigm
  - Importance of frequent auditing
- History tree
  - Efficient auditing
  - Scalable
  - Offers other features

  - Proofs and more in the papers

Conclusion

- Presented two tamper evident algorithms
  - New PAD designs
    - Comprehensive evaluation
    - Monetary analysis
  - Tamper-evident history
    - New extensions for fast digital signatures
- Focused on efficiency in the real-world
- Code and technical reports

  http://tamperevident.cs.rice.edu