Efficient Data Structures for Tamper-Evident Logging

Scott A. Crosby
Dan S. Wallach
Rice University
Everyone has logs
Tamper evident solutions

• Current commercial 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

• Forward integrity
  – Events prior to Byzantine failure are tamper-evident
    • Don’t know when logger becomes evil
  – Clients are trusted

• Strong insider attacks
  – Malicious administrator
    • Evil logger
  – Clients may be mostly evil
    • Only trusted during insertion protocol
Limitations and Assumptions

• Limitations
  – Detect misbehaviour, not prevent it
  – Cannot prevent ‘junk’ from being logged

• Assumptions
  – Privacy is outside our scope
    • Data may encrypted
  – Crypto is secure
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
This talk

• Discuss the necessity of auditing
• Describe the history tree
• Evaluation
• Scaling the log
Tamper evident log

- Events come in
- Commitments go out
  - Each commits to the entire past
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 $C_n$
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 \( X_n \) ?

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 \( X_i \) ?
Problem

• We don’t trust the logger!
  – Logger signs stream of log heads
  – Each corresponds to some log

Does $C_{n-3}$ really contain the just inserted $X_{n-3}$?

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 $X_i$?
Solution: Audit the logger

• Only way to detect tampering
  – Check the returned commitments
    • For consistency
    • For correct event lookup

• Previously
  – Auditing = looking historical events
    • Assumed to infrequent
    • Performance was ignored
Solution

• Auditors check the returned commitments
  – For consistency
  – For correct event lookup

• 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 required in forward integrity threat model
Auditing is a frequent operation

- Every commitment must have a non-zero probability of being audited
Forking the log

• Rolls back the log and adds on different events
  – Attack requires two commitments on different forks disagree on the contents of one event.
  – If system has historical integrity, audits must fail or be skipped
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
  – Verify proper insertion
  – Lookup historical events

• Incremental auditing
  – Prove consistency between two commitments
Membership auditing a hash chain

• Is $X_{n-5} \Rightarrow C_{n-3}$?
Membership auditing a hash chain

- Is $X_{n-5} \cdot C_{n-3}$?
Membership auditing a hash chain

- Is $X_{n-5}$ in the chain?
Incremental auditing a hash chain

• Are \( C''_{n-5} \# C'_{n-1} \) ?
Incremental auditing a hash chain
Incremental auditing a hash chain
Incremental auditing a hash chain

\[ X'_{n-5}, X'_{n-4}, X'_{n-3}, X'_{n-2}, X'_{n-1}, C'_{n-1}, P \]
Incremental auditing a hash chain
Existing tamper evident log designs

- Hash chain
  - Auditing is linear time
  - Historical lookups
    - Very inefficient

- Skiplist history [Maniatis,Baker]
  - 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
Incremental auditing
Auditor

Diagram:

- $C_4$
  - $X_1$
  - $X_2$
  - $X_3$
  - $X_4$

- $C_3$
Incremental proof

Auditor

C₃

C₇

X₁

X₂

X₃

X₄

X₅

X₆

X₇
Incremental proof

- P is consistent with \( C_7 \)
- P is consistent with \( C_3 \)
- Therefore \( C_7 \) and \( C_3 \) are consistent.
Incremental proof

- P is consistent with \( C_7 \)
- P is consistent with \( C_3 \)
- Therefore \( C_7 \) and \( C_3 \) are consistent.
Incremental proof

• P is consistent with $C_7$
• P is consistent with $C_3$
• Therefore $C_7$ and $C_3$ are consistent.
Incremental proof

- 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 $X_3 \neq C''_7$

- 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

- \((\mathcal{W}, \mathcal{W}, \mathcal{W})\)
  - \(\mathcal{W}\): Type of attributes on each node in history
  - \(\mathcal{W}\): Aggregation function
  - \(\mathcal{W}\): Maps an event to its attributes

For any predicate \(P\), as long as:
- \(P(x) \text{ OR } P(y) \implies P(x \mathcal{W} y)\)
- 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

<table>
<thead>
<tr>
<th></th>
<th>$O(c_j)$</th>
<th>$O(c_i)$</th>
<th>$O(x_i)$</th>
<th>Insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>History tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td></td>
</tr>
<tr>
<td>Hash chain</td>
<td>$O(j-i)$</td>
<td>$O(j-i)$</td>
<td>$O(1)$</td>
<td></td>
</tr>
<tr>
<td>Skip-list history</td>
<td>$O(j-i)$</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
<td></td>
</tr>
<tr>
<td>[Maniatis, Baker]</td>
<td>$O(j-i)$</td>
<td>$O(n)$ or $O(n)$</td>
<td>$O(1)$</td>
<td></td>
</tr>
</tbody>
</table>
Skip list history [Maniatis, 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 Management – 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

• Machine
  – Dual-core 2007 desktop machine
  – 4gb RAM
Performance

- Insert performance: 1,750 events/sec
  - 2.4% : Parse
  - 2.6% : Insert
  - 11.8% : Get commitment
  - 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
    - Compression reduces performance and proof size by 50%
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
Partial proofs

- Can re-use node hashes from prior audits
  - (eg, incremental proof from $C_3$ to $C_4$)
Conclusion

- New paradigm
  - Importance of frequent auditing
- History tree
  - Efficient auditing
  - Efficient predicate queries and safe deletion
  - Scalable
- Proofs of tamper-evidence will be in my PhD Thesis
Questions
Historical integrity

$X'_{n-5}$  $X'_{n-4}$

$C'_{n-4}$
Historical integrity
Historical integrity
Historical integrity
Defining historically integrity

• A logging system is tamper-evident when:
  – If there is a verified incremental proof between commitments $C_j$ and $C_k$ ($j<k$), then for all $i<j$ and all verifiable membership proofs that event $i$ in log $C_j$ is $X_i$ and event $i$ in log $C_k$ is $X'_i$, we must have $X_i=X'_i$. 
Safe deletion

- Unimportant events may be deleted
  - When auditor requests deleted event
    - Logger supplies proof that ancestor was not important