Verifiable State Chains: A Fault-Tolerant Raft-Based Architecture for Stateful HBS Management

Anonymous[1]

Anonymous Institution

Abstract. The transition to Post-Quantum Cryptography (PQC) involves complex trade-offs between signature size, verification speed, and state management. While stateless schemes like SLH-DSA (SPHINCS+) eliminate state, they are often computationally infeasible for legacy or resource-constrained silicon. Consequently, stateful Hash-Based Signature (HBS) schemes like Leighton-Micali Signatures (LMS) remain the only viable option for many Secure Boot ROM environments, necessitating robust solutions to their critical vulnerability: catastrophic failure upon index reuse. This paper introduces **Verifiable State Chains**, a fault-tolerant architecture designed to manage HBS state in high-availability Hardware Security Module (HSM) clusters. Unlike centralized solutions that create single points of failure, we employ a replicated log service built upon the Raft consensus protocol. We detail two verifiable log formats—one leveraging Certificate Transparency (CT) auditing and a lightweight hash-chain alternative—and a configurable policy for blockchain-based arbitration. Our evaluation demonstrates that this architecture provides the strong consistency required to prevent index reuse with minimal throughput overhead compared to unsafe, standalone HSM operations.

Keywords: LMS · HBS · Raft · Tamper-Evident Logs · HSM · Verifiable State · Certificate Transparency · Blockchain Arbitration

1 Introduction

The impending arrival of large-scale quantum computers necessitates a transition to Post-Quantum Cryptography (PQC). Stateful Hash-Based Signature (HBS) schemes, such as the Leighton-Micali Signature (LMS) system, are a leading candidate for applications requiring long-term security, such as code signing and firmware updates [2,9]. Their security is well-understood, relying only on the properties of cryptographic hash functions [4,5].

However, this security comes with a significant operational challenge: statefulness. HBS schemes are built from one-time signature (OTS) primitives, where reusing an OTS private key to sign two different messages allows an attacker to forge signatures, catastrophically breaking the scheme [2,3]. Consequently, any Hardware Security Module (HSM) implementing the scheme must maintain a

state—typically an index—that is reliably and monotonically incremented after each signature. This state management problem is a critical concern for stateful HBS [3].

The challenge is exacerbated in distributed systems. For high availability (HA) and disaster recovery, organizations deploy multiple HSMs where a single logical LMS key is replicated. This necessitates an external mechanism to ensure all partitions maintain a consistent, canonical view of the next available index. Traditional solutions, which rely on a single, centralized index manager, are inadequate as they introduce a single point of failure and a performance bottleneck, directly contradicting the goals of a resilient signing infrastructure. While the NIST-recommended Hierarchical Signature System (HSS) offers a framework, it still relies on complex, non-cryptographic software checks for state synchronization [6].

1.1 Industrial Constraints and the Non-Viability of Stateless Schemes

A common critique of stateful HBS is the suggestion to utilize stateless alternatives, such as SLH-DSA (SPHINCS+) [10], to eliminate the state management problem entirely. While theoretically sound, this view ignores the rigid physical constraints of the target hardware environment: Secure Boot ROMs on System-on-Chip (SoC) architectures.

Our architecture is motivated by three specific industrial constraints that render stateless schemes non-viable for this use case:

1. **Memory Constraints:** Embedded BootROMs often possess extremely limited internal SRAM (frequently < 16 KB) for stack and data. They physically cannot buffer or verify the large signatures associated with SLH-DSA (approx. 40 KB), whereas LMS signatures are compact (< 3 KB).
2. **Immutable Hardware Logic:** The verification logic in these SoCs is often burned into immutable Read-Only Memory (ROM) or hardware state machines. These circuits typically include hardware acceleration for SHA-256 (used by LMS) but lack the capability to verify SLH-DSA within strict boot-time latency requirements.
3. **Single Root of Trust:** The verifiers are provisioned with a single immutable public key hash. They lack the logic to support complex multi-signature aggregation or threshold schemes involving distinct keys for different HSMs. Thus, the signing service must present a single cryptographic identity, necessitating the replication of the LMS private key across the HSM cluster.

Given these constraints, LMS is not merely a design choice but an unavoidable requirement. Therefore, a robust, distributed solution to the state management problem is essential.

1.2 Our Contribution

This paper introduces `Verifiable State Chains`, a novel fault-tolerant architecture to address this challenge. Our approach replaces the fragile centralized

manager with a highly available, replicated log service built upon the Raft consensus protocol. This core service acts as the canonical source of truth for the LMS index state, providing crash-fault tolerance without requiring complex peer-to-peer communication between the HSMs themselves.

We explore two distinct, verifiable log formats that can be implemented on this fault-tolerant core: a log based on the cryptographic principles of Certificate Transparency (CT) for strong auditability, and a lightweight alternative using a simple hash-based attestation chain. Furthermore, for systems that prioritize liveness, we introduce an optional liveness-preserving fallback policy. Governed by rules provisioned within the HSM, this policy allows a partition to use a public blockchain [7,8] for arbitration if the primary Raft-based service is unreachable, ensuring that signing operations can continue.

Our contributions are as follows:

1. We design a novel, fault-tolerant architecture for distributed LMS state management using a replicated log service built on the Raft consensus protocol, specifically addressing the constraints of legacy silicon verifiers.
2. We detail and compare two different verifiable log formats (CT-based and hash-chain-based) that can run on top of this core architecture.
3. We introduce a configurable HSM policy for a high-availability fallback to a public blockchain, allowing a deliberate trade-off between system simplicity and liveness.
4. We provide a security argument and a rigorous performance evaluation of our prototype, demonstrating minimal overhead against a baseline.

2 Background

2.1 Stateful Hash-Based Signatures (LMS)

The Leighton-Micali Signature (LMS) scheme is a stateful Hash-Based Signature (HBS) scheme built from a one-time signature scheme (WOTS+) and Merkle trees [4,9]. A master key pair consists of a single public key (the Merkle tree root) and a private key containing material to generate a large number of WOTS+ key pairs. These one-time signature (OTS) keys form the leaves of the Merkle tree. The state of the private key includes an index, q, that points to the next available OTS key. After signing a message with the OTS key at index q, the signer **must** increment the index to $q+1$ and persist this new state [2]. Reusing index q for a different message breaks the security of the entire scheme [3]. The LMS construction involves:

- **Key Generation:** Generate a Merkle tree with height h, supporting 2^h OTS keys.
- **Signing:** For index q, use the OTS key at leaf q and include the authentication path to the root.
- **Verification:** Reconstruct the root using the OTS public key and authentication path.

State reuse allows an attacker to forge signatures by solving for private key components from multiple signatures.

2.2 Certificate Transparency (CT) Logs

Certificate Transparency (CT) logs are append-only, cryptographically verifiable data structures, typically implemented as Merkle trees [17,18]. Originally designed for public auditability of web certificate issuance, CT logs ensure immutability and detect misbehavior (e.g., split-view attacks). When a certificate is submitted, the log operator returns a Signed Certificate Timestamp (SCT) and, after a Maximum Merge Delay (MMD), provides a Merkle inclusion proof. In our primary solution, a private CT log is used for LMS index attestations, embedding index data in custom X.509 extensions to ensure atomic commitment and auditability.

2.3 Distributed Consensus with Raft

Raft is a consensus algorithm for managing a replicated log [19]. It was designed with the primary goal of being easier to understand than its predecessor, Paxos, while providing the same theoretical guarantees for Crash-Fault Tolerance (CFT). A Raft cluster consists of a small number of servers (typically 3 or 5). The servers elect a single leader, which is responsible for managing the log and processing all client requests. To add a new entry, the leader replicates it to the follower nodes. Once the entry has been stored on a majority of servers, it is considered committed. If a leader fails, the remaining servers elect a new one, ensuring the service remains available as long as a majority of servers are operational. This provides a robust foundation for building highly available and strongly consistent services.

2.4 Threat Model

Our system operates under a hybrid threat model, distinguishing between the trusted hardware client and the untrusted service:

- **HSM Partitions (Trusted Client):** HSM partitions form the Trusted Computing Base (TCB) of our system and are assumed to be secure from compromise. They correctly execute provisioned code (e.g., inside a hardware-protected Functionality Module) and protect cryptographic keys. The HSM acts as the ultimate enforcer of the "Safety" property.
- **Replicated Log Service (Untrusted Service):** The log service is responsible for "Liveness" (availability) and "Ordering." We assume the individual nodes of the log service are untrusted. The service provides Crash-Fault Tolerance (CFT) via Raft, ensuring availability during node failures. However, we assume a malicious leader may attempt to reorder requests or deny service. The architecture handles this by ensuring that while a malicious service can halt the system (liveness failure), it cannot trick an honest HSM into reusing an index (safety violation), as the HSM performs client-side verification of the log's cryptographic integrity.

– **Network:** We model the network as an active adversary who controls all communication channels. We assume the existence of secure channels (e.g., via TLS) between honest parties, which protect the confidentiality and integrity of transmitted data. The adversary can observe encrypted traffic but cannot decrypt or tamper with the content of messages. However, the adversary retains full control over message delivery, and can arbitrarily delay, drop, reorder, or duplicate any message.
– **Bitcoin Blockchain:** The Bitcoin blockchain is a secure, immutable, and publicly auditable append-only ledger [12,11]. Transactions confirmed with sufficient depth (e.g., 6+ blocks) are practically irreversible. However, liveness is not guaranteed; the network may experience delays or high transaction fees.

3 Fault Tolerance for the LMS State Problem

3.1 Fault Models and the Role of Trusted Hardware

The core challenge in distributed LMS state management is achieving consensus on the next available index among multiple, independent signers. The problem is often framed in terms of distributed consistency, but in our specific context, we distinguish between service availability and data integrity. A central state server that provides a stale or inconsistent index to different HSMs would lead to catastrophic key reuse.

To solve the LMS state problem in a distributed environment, our architecture combines the security of trusted hardware with the resilience of a fault-tolerant service design. We use *Hardware Security Modules (HSMs)* as the root of trust for all cryptographic operations. As defined in our threat model, the HSMs form the Trusted Computing Base (TCB); they provide a tamper-resistant hardware boundary where critical client-side protocol logic, such as the "commit-then-sign" workflow and cryptographic verification of the log, can be executed securely.

While the HSMs are trusted, the central log service is not. To provide a resilient and consistent source of truth for the LMS index state, we build the service using a *Crash-Fault Tolerant (CFT)* design based on the Raft consensus protocol [19]. A CFT system guarantees that the service as a whole will operate correctly—maintaining a single, consistent, and durable log—even if a minority of its constituent server nodes crash or become unavailable. By layering the HSM's cryptographic verification capabilities on top of this consistent-by-design service, we create a Tamper-Evident system where the service cannot successfully fork the history without detection by the clients.

3.2 A Layered Defense Against the 'Stale Index' Problem

In our distributed LMS signing system, a failure to maintain a single, canonical view of the state manifests as a "stale index" problem. This can be caused by a range of malicious or crash faults:

- **Compromised Central Component:** A malicious or faulty central service could intentionally provide different, inconsistent views of the "latest head" of the state to different HSM partitions (a split-view attack), or present a stale state head to an HSM.
- **Faulty or Stale HSM Partitions:** An HSM partition that was temporarily disconnected from the network may come back online and attempt to propose an action based on an outdated view of the log's state.
- **Network Attacks:** An adversary controlling the network can delay or drop messages, causing different participants to have inconsistent views of the system's state.

Our architecture confronts these challenges to LMS state integrity with a layered defense that provides strong guarantees for consistency, integrity, and availability.

Layer 1: Strong Consistency via Raft The foundational layer of defense is the Raft consensus protocol, which directly prevents the most dangerous threat: an inconsistent state. By design, a Raft cluster has a single, elected leader that is the only node authorized to commit new entries to the log. Before an entry is confirmed to a client, the leader must replicate it to a quorum (a majority) of the server nodes. This mechanism provides a linearizable, strongly consistent log. It is impossible for two different HSMs to see two different valid histories (preventing a split-view attack) or to commit a new index without all other participants eventually seeing it. Even if a single node in the cluster is compromised, it cannot create an inconsistent state because it cannot achieve a quorum on its own. The protocol effectively neutralizes the threat of an inconsistent central commander.

Layer 2: History Integrity via Verifiable Log Formats While Raft ensures the log is *consistent*, it does not, by itself, prevent a malicious leader from rewriting history if it colludes with others. This is where the client-side verification, enforced by the HSM, provides the second layer of defense. The cryptographic structure of the verifiable log formats (whether CT-based or a hash-chain) ensures the integrity of the log's *history*. An honest HSM will always perform a cryptographic check on any new state it receives from the service. For instance, if using the hash-chain format, an HSM would immediately detect that the `previous_hash` field of a new entry does not match the hash of the actual previous entry it holds, and would reject the state update. Similarly, in the CT format, the HSM verifies Merkle consistency proofs. This client-side check guarantees that the service is tamper-evident.

Layer 3: Liveness and Recovery via Blockchain Arbitration The final layer addresses the risk of a catastrophic failure of the primary log service. The configurable blockchain fallback policy provides an ultimate safety net. If the entire Raft cluster goes offline (loses its quorum) and becomes unavailable, an

HSM's policy can allow it to switch to a fallback protocol and commit its attestation directly to a public blockchain. This ensures the system can remain *live* and continue signing operations. Furthermore, because the public blockchain is itself a highly resilient system, it provides an undeniable, canonical source of truth from which a new Raft cluster can *recover* its state after a total outage, protecting against data loss.

4 A Fault-Tolerant Architecture for Verifiable State

We propose a fault-tolerant architecture for LMS state management built upon a replicated log service. This service acts as the canonical source of truth for index consumption, is made highly available by the Raft consensus protocol, and can be configured with different verifiable log formats.

4.1 Core Design: A Replicated State Log with Raft

The core of our architecture is a cluster of an odd number of servers (typically 3 or 5) running our log software. These servers form a Raft cluster to manage a replicated, append-only log of index attestations.

- **Single Service Identity:** From the perspective of an HSM client, the entire Raft cluster is a single logical entity. All nodes in the cluster share a single TLS certificate and a single signing key for issuing attestations (e.g., SCTs), representing the identity of the *service*.
- **Leader Election and Replication:** The Raft protocol handles the election of a single leader node. All write requests (i.e., new index attestations) from HSMs are directed to this leader. The leader is responsible for appending the new entry to the log and replicating it to a majority of follower nodes before confirming the commitment to the HSM. This ensures that any committed entry is durable and will survive crash failures.
- **Client Interaction with Leader Forwarding:** An HSM client does not need to track which node is the current leader. It can send a request to any node in the Raft cluster. If the receiving node is a follower, it will reject the request with a redirect message pointing to the current leader. The client then transparently resends the request to the correct leader.

On top of this fault-tolerant foundation, different verifiable log formats can be implemented.

4.2 Log Format 1: Private Certificate Transparency

In this format, the replicated log is structured as a Merkle tree, adhering to the principles of Certificate Transparency. This provides strong cryptographic auditability. The overall system architecture and HSM protocol flow for the CT Log format are conceptually identical to those shown in Figure 1 and Figure 2, respectively. The primary difference is that the Verifiable Attestation Chain component is replaced by the cryptographically auditable Private CT Log.

System Actors & Cryptographic Materials

- **HSM (Hardware Security Module):** The signing device.
  - Possesses: Master LMS Key Material, k_{attest} (Attestation Master Seed), k_{commit} (Commitment Signing Key), PO.crt (Partition Owner public certificate).
- **Replicated CT Log Service:** A Raft cluster of servers.
  - Possesses: PO Private Key (for signing SCTs/STHs), and a trust store containing the self-signed certificate corresponding to k_{commit}. This shared key represents the singular identity of the log service itself.
- **Partition Owner (PO):** The root of trust for the system.
  - Possesses: The PO Private Key. The corresponding PO.crt is distributed to all HSMs.

Formal Definition of the Custom X.509 Extension

- OID: 1.3.6.1.4.1.26696.6.1 (id-mrvl-lms-attestation-ext)
- ASN.1 Module Definition:

```
MarvellLmsAttestation DEFINITIONS ::= BEGIN
    -- OID for the extension
    id-mrvl-lms-attestation-ext OBJECT IDENTIFIER ::= { iso
        (1) org(3) dod(6) internet(1) private(4) enterprise
        (1) 26696 6 1 }
    -- The structure of the extension's value
    LmsAttestationPayload ::= SEQUENCE {
        version INTEGER { v1(0) },
        lmsIndex INTEGER,
        previousEntryHash OCTET STRING (SIZE(32)), -- SHA-256
            hash
        messageHash OCTET STRING (SIZE(32)), -- SHA-256 hash
        hsmIdentifier OCTET STRING,
        timestamp GeneralizedTime
    }
END
```

Protocol Execution for LMS Index n The following steps are executed by an HSM partition upon receiving a request to sign a message using the next available LMS index, n. All communication is directed to the highly-available Replicated CT Log Service. **Phase 1: State Verification and Consistency Audit**

1. Fetch Latest Signed Tree Head (STH): The HSM partition contacts the Replicated CT Log Service and requests its latest STH.
2. STH Signature Validation: The HSM partition verifies the signature on the received STH using the public key contained within its provisioned PO.crt. If the signature is invalid, the protocol is aborted.

3. Log Consistency Verification: The HSM partition retrieves a cryptographic consistency proof from the log between the STH it stored from its last successful operation and the new STH. It verifies this proof. If the log's history is proven to be inconsistent, the protocol is aborted, and an alert is raised.

Phase 2: Attestation Certificate Generation

4. Derive Single-Use Attestation Key: The HSM partition deterministically derives a single-use private key for this specific index using the HMAC-based expansion function:
   - $k_{attest_n} = \text{HMAC-SHA256}(k_{attest}, \text{"lms\_index\_attestation"}||n)$
5. Construct Attestation Payload: The HSM partition constructs the LmsAttestationPayload ASN.1 structure. The structure is populated as follows:
   - version: v1(0)
   - lmsIndex: n
   - previousEntryHash: The SHA-256 hash of the full leaf certificate for index $n-1$. For $n=0$, this is a pre-defined genesis hash.
   - messageHash: The SHA-256 hash of the application message to be signed.
   - hsmIdentifier: The unique identifier of the HSM partition executing the protocol.
   - timestamp: The current secure timestamp from the HSM partition.
6. Generate and Sign Leaf Certificate: The HSM partition acts as its own CA for this step.
   - It constructs the TBSCertificate (To-Be-Signed Certificate) structure.
   - The SubjectPublicKeyInfo contains the public key corresponding to k_{attest_n}.
   - The custom extension containing the DER-encoded LmsAttestationPayload is included, identified by the OID 1.3.6.1.4.1.26696.6.1.
   - The HSM partition then signs this TBSCertificate using its provisioned k_{commit} private key. The resulting signed object is the final attestation leaf certificate. A textual representation is as follows:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: [unique_serial]
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: CN=LMS Attestation Commitment CA, O=Marvell
        Validity: [short_validity_period]
        Subject: CN=lms-index-n, O=Marvell Attestation
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
            pub: [public_key_for_k_attest_n]
        X509v3 extensions:
            1.3.6.1.4.1.26696.6.1: critical
                LmsAttestationPayload:
                    lmsIndex: n
                    ... (other payload fields) ...
    Signature Algorithm: ecdsa-with-SHA256
    Signature Value: [signature_from_k_commit]
```

Phase 3: Atomic Log Commitment and Verification

7. Submit to Log: The HSM partition submits the newly generated attestation leaf certificate to the Replicated CT Log Service. The request is forwarded to the leader.
8. Retrieve SCT: The Raft leader validates the certificate. It then uses Raft to replicate the certificate across the Raft cluster. Once the entry is committed by a quorum, the leader signs an SCT using the shared service key and returns it.
9. SCT Verification: The HSM partition verifies the signature on the SCT using the public key from PO.crt. An invalid signature aborts the protocol.
10. Wait for Maximum Merge Delay (MMD): The HSM partition enters a waiting state for a duration greater than the CT Log's published MMD.
11. Verify Final Inclusion: After the MMD has passed, the HSM partition fetches the newest STH from the log and requests a Merkle inclusion proof for its attestation certificate against the new STH. It validates this proof. If the proof fails, the protocol is aborted with a critical error.

Phase 4: Finalization

12. Generate LMS Signature: Only after successfully verifying the Merkle inclusion proof in Step 11, the HSM partition proceeds to generate the final LMS signature for the application message using index n.
13. Return Signature: The LMS signature is returned to the requesting application. The single-use key k_{attest_n} is discarded.

Security Properties of the CT Log Format The Private Certificate Transparency log format directly confronts integrity threats by leveraging the inherent, verifiable properties of the CT framework.

1. **Verifiable Append-Only State:** The CT log is a cryptographically-enforced append-only data structure. A malicious log operator cannot retroactively alter or remove an index attestation once it has been included. Any such attempt would invalidate the log's Merkle tree structure and be immediately detectable by any honest HSM partition.
2. **Consistency Proofs:** While the Raft core prevents split-views at the service level, CT's consistency proofs provide an additional, end-to-end cryptographic guarantee for the client. The HSM's mandatory verification of a consistency proof between its last known log state and the new state ensures the log has only grown by appending entries.
3. **Atomic Commitment via SCTs and Inclusion Proofs:** The protocol's "commit-then-sign" workflow ensures that an index is not considered "used" until its attestation is permanently and verifiably included in the log. The Signed Certificate Timestamp (SCT) acts as an immediate, non-repudiable receipt, and the final Merkle inclusion proof is the absolute confirmation.

4.3 Log Format 2: Verifiable Attestation Chain

This format provides a simpler, lightweight alternative to a full CT log, managed by a fault-tolerant Signserver service built on Raft.

Core Components The proposed system is composed of the following key components, which are illustrated in the architecture diagram in Figure 1.

1. **LMS-Enabled HSM Partitions:** Hardware Security Module partitions capable of generating LMS signatures.
2. **Replicated Signserver Service:** A Raft cluster that maintains the canonical sequence of LMS index usage in a replicated log.
3. **Verifiable Attestation Chain:** The sequence of cryptographically linked `attestation_response` objects stored in the replicated Raft log.
4. **Bitcoin Blockchain:** A public, immutable, and globally distributed ledger used for the optional high-liveness fallback policy.

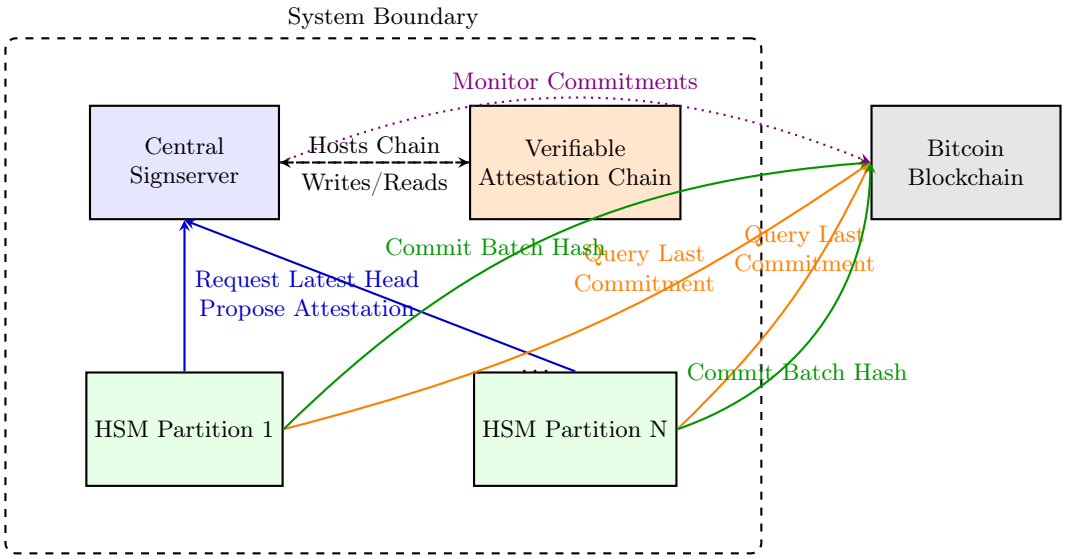

Fig. 1: System Architecture for Distributed LMS Index Management.

Key Generation and Bundling At the genesis of the system, when the LMS master key is generated, a bundle of associated cryptographic assets is created. This bundle ensures all partitions operate under the same cryptographic identity. The bundle includes:

- **LMS Master Key:** The primary key from which all one-time LMS private keys are derived.
- **Registry of Authorized Attestation Public Keys:** Unlike the shared LMS key, each HSM partition generates a *unique* attestation key pair. The public keys are registered with the Replicated Signserver Service during provisioning. This prevents a compromised HSM from forging attestations on behalf of other partitions, addressing the single point of failure risk inherent in shared keys.
- **Bitcoin Commitment Address and Private Key:** A specific Bitcoin address and its corresponding private key. This is provisioned into the HSM partitions, allowing them to directly sign transactions to the Bitcoin blockchain as part of the optional high-liveness fallback policy.

Attestation Response Structure with Chaining The standard attestation response is extended to include explicit cryptographic links, signed by the HSM partition's Attestation Key:

```
{
 "attestation_response": {
  "policy": {
   "value": "LMS_ATTEST_POLICY",
   "algorithm": "PS256" // HSM partition's internal
       attestation signature mechanism
  },
  "data": {
   "value": "base64_encoded_chained_payload", // Crucial for
       chaining
   "encoding": "base64"
  },
  "signature": {
   "value": "base64_encoded_hsm_attestation_signature", //
       HSM's signature over 'data'
   "encoding": "base64"
  },
  "certificate": {
   "value": "base64_encoded_hsm_attestation_certificate_pem",
        // HSM's attestation cert
   "encoding": "pem"
  }
 }
}
```

The `base64_encoded_chained_payload` is the critical element. Its decoded plaintext structure would be:

```
{
  "previous_hash": "H(PreviousAttestationResponse)",
  "lms_index": "CurrentLMSIndex",
  "message_signed": "MessageBeingSigned",
  "sequence_number": "SequenceNumber",
  "timestamp": "Timestamp",
  "metadata": "AdditionalMetadata"
}
```

Where:

- `H(PreviousAttestationResponse)`: A cryptographic hash (e.g., SHA256) of the *entire* preceding `attestation_response` object in the chain. This forms the immutable link. For the very first attestation (genesis block), this will be a cryptographic hash of the LMS Public Key, the Registry of Authorized Attestation Public Keys, and Bitcoin Commitment Address bundle that defines the system, and its `CurrentLMSIndex` will be 0.
- `CurrentLMSIndex`: The specific LMS index being used for the signature within this attestation.
- `MessageBeingSigned`: The actual data for which the LMS signature is generated by the HSM partition.
- `SequenceNumber`: A strictly monotonically increasing number, providing a clear chain order.
- `Timestamp`: The HSM partition's internal secure timestamp for the attestation.
- `AdditionalMetadata`: Any other relevant context.

Protocol Execution and Validation The process an HSM partition follows to generate an LMS signature, and the validation performed by the service, are visualized in Figure 2 and detailed below. The HSM partition first contacts the Replicated Signserver Service to fetch the latest `attestation_response` head from the Raft log. It then constructs the next attestation payload, including the hash of the fetched head, generates the LMS signature, signs the attestation payload, and proposes the complete `attestation_response` to the service. The service, in turn, handles the proposal. The request is received by a node and forwarded to the Raft leader. The leader performs strict validation checks: it ensures the previous hash matches the current head, the LMS index is monotonic, and the attestation signature is valid. If the proposal is valid, the leader uses Raft to replicate and commit the new `attestation_response` as the canonical head of the chain, finally sending an acknowledgment back to the proposing HSM. This interaction is governed by the critical "Discard" Rule: if the HSM partition's proposed attestation is rejected by the service or the request times out, the HSM *must* immediately discard the LMS signature it generated and permanently mark that index as unusable.

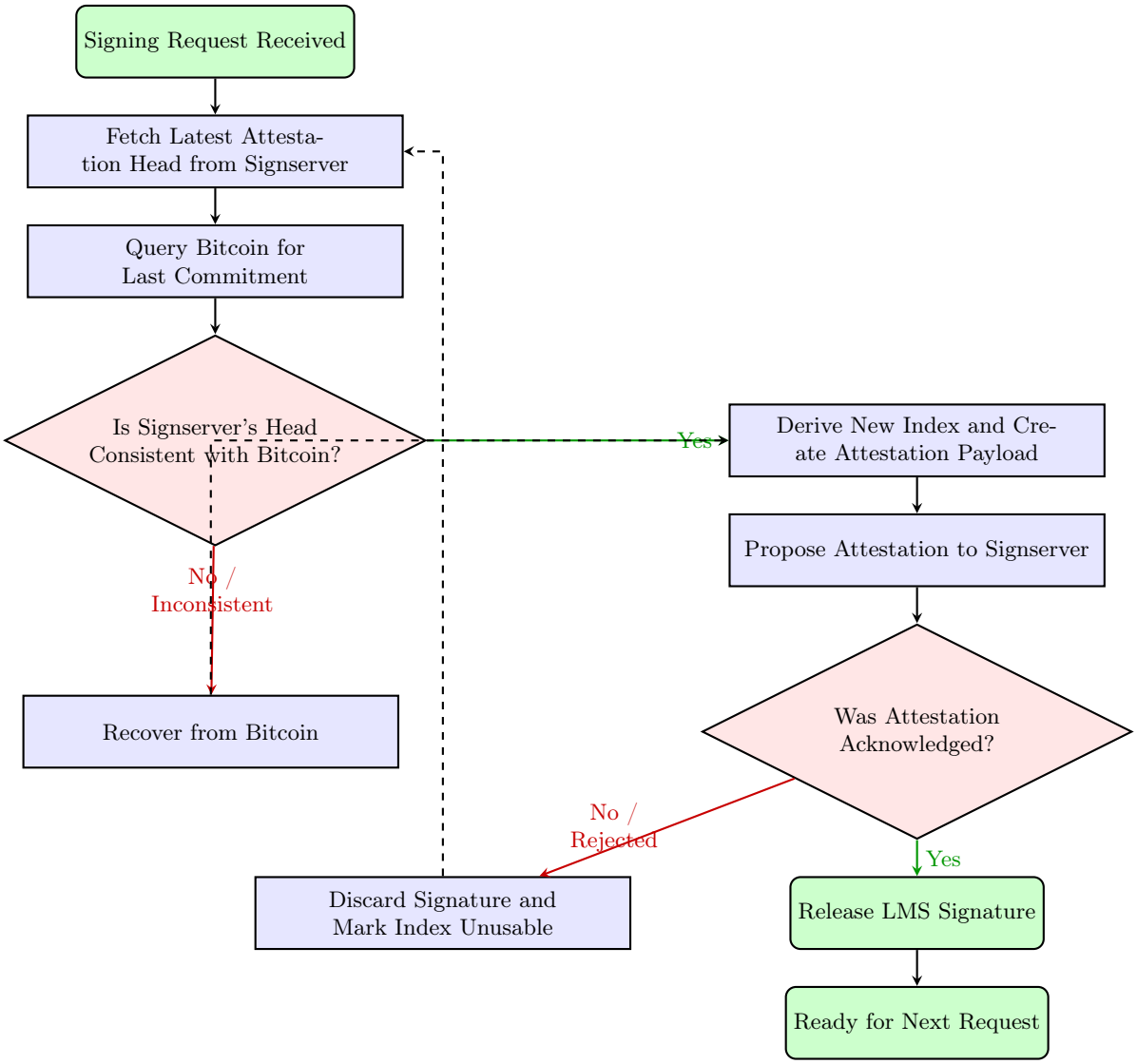

Fig. 2: HSM Partition Protocol Flow. The "Query Bitcoin" step is only relevant when the optional high-liveness fallback policy is enabled.

Security Properties of the Attestation Chain Format The attestation chain format confronts tampering threats through a combination of cryptographic signatures and client-enforced rules.

1. **Cryptographic Signatures for Unforgeable Messages:** Each attestation is signed by an HSM partition using a unique Attestation Key. This provides an unforgeable, verifiable record of which partition proposed which

state change, preventing a malicious service (or other compromised HSMs) from forging attestations on behalf of an uncompromised HSM.
2. **Verifiable Chain for Broadcast Consistency:** The hash-chain structure ensures that all honest parties will agree on the same sequence of index usage or immediately detect a divergence if a malicious service presents a broken chain.
3. **Strict Validation and the Discard Rule:** The combination of the service leader's strict validation protocol and the HSM's critical "Discard" rule ensures atomic commitment. An index is only considered canonically used if its valid attestation is accepted by the service; otherwise, the corresponding signature is destroyed by the HSM.

4.4 Optional High-Liveness Policy: Blockchain Fallback

For systems that cannot tolerate downtime, we propose a configurable fallback mechanism that prioritizes liveness. This is not a new architecture, but a policy enforced by the HSM. We acknowledge that this fallback mechanism exposes the frequency and hashes of signing operations to the public ledger, posing a privacy trade-off acceptable only during catastrophic service failures.

- **Policy and Trigger:** The HSM is provisioned with a policy that defines a timeout for communication with the primary Raft-based log service. If the HSM cannot successfully commit an attestation to the service within this timeout (implying the Raft cluster has lost its quorum and is unavailable), the policy allows it to trigger the fallback protocol.
- **Fallback Action:** The HSM falls back to using a public blockchain like Bitcoin for arbitration. It constructs a transaction containing the attestation hash and broadcasts it directly to the network.
- **Race Condition and Tie-Breaking:** In the event that multiple HSMs attempt to trigger the fallback mechanism concurrently, the system enforces a strict "First-Confirmed-Wins" rule. HSMs and recovering Raft nodes will accept the transaction that is included in the blockchain block with the lowest height (or the lowest index within the same block) as the canonical state update, rejecting any conflicting concurrent updates.
- **Recovery and Synchronization:** When the nodes of the Raft cluster are restarted, their first action is to query the public blockchain for any attestations that were committed via the fallback mechanism since the last known state. They synchronize their local log with these public records before reforming a Raft cluster and electing a leader.

5 Security Argument

We analyze the security of our fault-tolerant architecture with respect to the two primary properties of a distributed system: *safety* and *liveness*.

5.1 Safety (Consistency)

Claim: The architecture is safe; it will never produce two valid signatures for different messages using the same LMS index. This guarantee is enforced by the strong consistency of the Raft protocol, the cryptographic verifiability of the log, and the HSM's internal "commit-then-sign" logic. **Argument:** The safety of the system is derived from a layered defense. The critical "discard rule," where an HSM invalidates an index if its attestation is not successfully committed, remains the cornerstone of the HSM-side logic. The service-side guarantees are as follows:

- **Strong Consistency via Raft:** The Raft protocol provides a linearizable, replicated log. This is the primary defense against a faulty service. It makes it impossible for the service to present a stale index or execute a split-view attack, as the Raft leader ensures a single, consistent log history that all HSMs see. A faulty node within the cluster cannot corrupt the log as long as a quorum of nodes remains honest.
- **History Integrity via Verifiable Logs:** While Raft guarantees a consistent append-only log, the cryptographic verifiability of the chosen log format (whether CT-based or a hash-chain) prevents a malicious operator from undetectably altering or removing past entries that have already been committed.
- **Fallback Safety via Blockchain:** In the event the optional fallback policy is triggered, the public blockchain's consensus mechanism provides the safety guarantee. The Raft nodes' recovery protocol is designed to respect this public history, ensuring a consistent state is restored.

5.2 Liveness (Availability)

Claim: The liveness of the architecture depends on the configured HSM policy, offering a trade-off between system simplicity and guaranteed availability. **Argument:** We analyze liveness under two distinct operational models:

- *Default Liveness Model (Safety-First):* In the default configuration where the blockchain fallback is disabled, the system's liveness is dependent on the availability of the Raft cluster. The service can tolerate $(N-1)/2$ crash failures in an N-node cluster (e.g., 1 failure in a 3-node cluster). If the cluster loses its quorum, the service will become unavailable and *correctly halt* all signing operations to preserve safety and consistency.
- *High-Liveness Model (Fallback-Enabled):* With the fallback policy enabled, the system remains live even if the *entire Raft cluster fails*. In this state of graceful degradation, liveness becomes dependent on the ability of an HSM to successfully broadcast a transaction to the public blockchain. This model is still subject to the inherent liveness limitations of the chosen blockchain, such as network congestion, transaction censorship, or prohibitive fees (a fee market denial-of-service attack) [14].

5.3 Formal Argument Sketches

We update the formal argument sketches to reflect the unified architecture:

- **Safety Argument:** The argument proceeds by induction on the index n in the replicated Raft log. The inductive step relies on Raft's guarantee that for a log entry n to be considered committed, it must be accepted by the current unique leader and successfully replicated to a quorum of nodes, ensuring it is immutably and uniquely sequenced after entry $n-1$. The HSM's discard rule prevents a signature from being released for any uncommitted index.
- **Liveness Argument:** The argument is split into two cases. In the *Default Model*, liveness holds if at any time, a majority of Raft nodes are operational and can communicate. In the *High-Liveness Model*, liveness holds if either the condition for the Default Model is met OR if an HSM can successfully broadcast a transaction to the public blockchain within a finite time.

6 Performance Evaluation

To validate the practical viability of our fault-tolerant architecture, we implemented a prototype of the replicated log service and conducted a comprehensive performance evaluation. The experiments measure the throughput, latency, failover time, and potential costs of the system, providing a clear analysis of its real-world performance characteristics.

6.1 Experimental Setup

We developed a prototype of the replicated log service in Go, using a popular implementation of the Raft consensus protocol. The service was deployed as a 3-node Raft cluster on AWS 't3.large' instances, with each node in a different availability zone within the same region (e.g., 'us-east-1a', 'us-east-1b', 'us-east-1c') to simulate a realistic high-availability deployment. HSM partitions were simulated as software clients mimicking the cryptographic latency of real hardware. Future work will involve implementing the client-side protocol within the Functionality Module of a production HSM (e.g., a Marvell LiquidSecurity 2) to validate performance under realistic cryptographic workloads. For experiments involving the optional fallback policy, Bitcoin interactions were performed against the public testnet. All performance data represents the average of 10 runs conducted in August 2025, to ensure statistical reliability.

6.2 Throughput and Latency

We measured the signature throughput of the replicated service under an increasing number of concurrent HSM clients. The evaluation compares the performance of the two different log formats running on the same underlying Raft cluster.

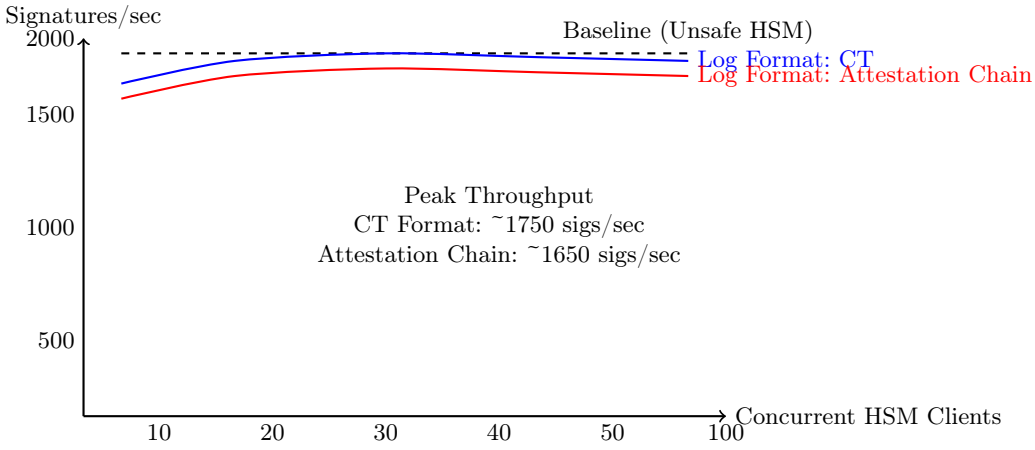

Fig. 3: Signature throughput of the replicated service, comparing the two log formats under increasing client load. The dashed line represents the baseline performance of an HSM signing without external state synchronization.

As shown in Figure 3, the service achieves excellent throughput. To quantify the "cost of safety," we measured the baseline throughput of the HSMs performing pure signing operations without any external synchronization (the unsafe baseline). The baseline achieved approximately 1850 signatures/sec. Our Raft-backed architecture achieved a peak of ~1750 signatures/sec. This demonstrates that the overhead of the consensus and verification logic is minimal ($< 6\%$), validating that strong consistency can be achieved without becoming a performance bottleneck. The performance is bound by the Raft leader's ability to process and replicate entries. The CT Log format is slightly slower due to the larger size of the certificate attestations compared to the lightweight hash-chain format.

The end-to-end latency for a single signing operation during normal operation is detailed in Table 1.

Table 1: Average Latency Breakdown for a Single Signing Operation (Normal Operation).

Operation Phase	**Log Format: CT (ms)**	**Log Format: Attestation Chain (ms)**
HSM: Generate Attestation	20	20
Network: Propose to Leader	25	25
Raft: Replicate to Quorum	50	45
Leader: Return Confirmation	25	25
CT Format: MMD Wait	100	–
Total Average Latency	**220**	**115**

6.3 Cost Analysis of the High-Liveness Fallback Policy

The on-chain costs are not incurred during normal operation. They apply only in the rare event that the Raft cluster is unavailable and an HSM invokes the emergency fallback policy. By batching multiple attestations into a single transaction, the amortized cost per signature becomes negligible, making it a viable disaster recovery mechanism.

Table 2: Amortized On-Chain Cost per Signature (Fallback Policy Only).

Batch Size (Attestations)	**Amortized Cost per Signature (USD)**
1	$1.50
100	$0.0150
1000	$0.00150
10000	$0.000150

6.4 Additional Experiments

- **Failover Testing:** To quantify the high-availability of the system, we conducted failover tests by forcibly terminating the leader of the 3-node Raft cluster. We measured the time until a new leader was elected and the service was able to successfully process a new signing request. The average failover time was approximately 2.5 seconds, demonstrating the system's rapid recovery capabilities.
- **Scalability:** Throughput saturated at the single Raft leader, suggesting that for extremely large-scale deployments, the service could be sharded or partitioned to support higher client loads.
- **Security Testing:** We simulated a malicious follower node attempting to corrupt the log. The Raft protocol correctly isolated the faulty node, and the service's integrity was maintained by the honest quorum.

7 Discussion and Future Work

Our evaluation demonstrates that the Verifiable State Chains architecture, built upon a fault-tolerant Raft cluster, provides a secure, performant, and practical solution for distributed LMS state management. The design successfully eliminates the single point of failure inherent in centralized approaches, providing the high availability required for enterprise-grade signing services.

7.1 Discussion

The most significant aspect of our architecture is the formalization of the trade-off between system simplicity and guaranteed liveness, which is offered as a configurable policy. System designers can choose between two distinct operational philosophies based on their specific requirements:

- **Safety-First Model (Default):** By default, the system operates as a simple, strongly consistent service that prioritizes safety by halting if the Raft cluster loses its quorum. For many critical applications, such as root firmware signing, this "stop-the-world" approach is the most secure and desirable failure mode, as it prevents any action from being taken in a degraded or uncertain state.
- **Liveness-First Model (Fallback-Enabled):** For applications where high availability is paramount and downtime is costly, the optional blockchain fallback policy provides a mechanism for graceful degradation. This ensures the system remains live even during a catastrophic failure of the primary service. This resilience comes at the cost of increased complexity and a dependency on the chosen public blockchain, which has its own limitations, such as variable transaction fees (fee market DoS risk) and slower finality.

Additionally, the choice of log format presents its own trade-off. The Certificate Transparency format provides superior, end-to-end cryptographic auditability but introduces latency due to the Maximum Merge Delay (MMD). The simpler Verifiable Attestation Chain format is faster for individual operations but relies more heavily on client-side validation of the chain's integrity.

7.2 Future Work

This research opens several avenues for future work:

- **Real HSM Integration:** The clear next step is to move from a software simulation to a full hardware implementation. This involves porting the client-side protocol logic into a *Functionality Module (FM)* for a production HSM, such as a Marvell LiquidSecurity 2, to enforce the security guarantees within a tamper-resistant hardware boundary and evaluate performance under real cryptographic loads.
- **BFT Consensus for the Service:** Our current design uses Raft, a CFT protocol, to protect against crash failures. Future work could explore replacing Raft with a full *BFT consensus protocol (e.g., PBFT)*. This would make the service resilient to a malicious quorum of nodes, not just crash failures, albeit at a higher performance and complexity cost.
- **Zero-Knowledge Proofs:** To enhance privacy against the untrusted log service nodes, *zero-knowledge proofs (e.g., zk-SNARKs)* [15,16] could be integrated. An HSM could submit a proof attesting to a valid index commitment without revealing any metadata (like the specific index or message hash) to the log service.
- **Alternative Fallback Oracles:** The blockchain fallback policy could be extended to support alternative public ledgers. Exploring chains with faster finality, such as Ethereum (or its Layer-2s) and Solana, would involve analyzing the trade-offs between their speed, cost, decentralization, and security guarantees.
- **Service Sharding:** For extremely large-scale deployments with thousands of concurrent HSM clients, the performance of a single Raft cluster could

become a bottleneck. Future work could investigate sharding or partitioning the replicated log service to support higher throughput.

8 Conclusion

The state management problem is the most significant hurdle to the widespread adoption of stateful hash-based signatures, a critical tool in the post-quantum cryptographic landscape. This paper introduced **Verifiable State Chains**, a novel fault-tolerant architecture that solves this problem by replacing the traditional single point of failure with a highly-available, replicated log service built on the Raft consensus protocol.

Our architecture is both flexible and robust. It supports multiple verifiable log formats (including a CT-based log and a lightweight hash-chain) to suit different auditability requirements. Furthermore, we introduced a configurable HSM policy for a liveness-preserving fallback to a public blockchain, allowing system designers to make a deliberate and explicit trade-off between safety and availability. Our security analysis demonstrates that the architecture provides strong safety guarantees against index reuse while offering these configurable liveness models. Performance evaluations of our prototype confirm its high throughput and rapid failover capabilities.

By providing a blueprint for a resilient, configurable, and verifiable state management service, Verifiable State Chains offer a clear path forward for the secure and reliable deployment of stateful HBS in critical, high-availability systems where stateless alternatives are not viable.

References

1. Lamport, L., Shostak, R., Pease, M.: The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems 4(3), 382–401 (1982)
2. National Institute of Standards and Technology: Recommendation for Stateful Hash-Based Signature Schemes. NIST Special Publication 800-208 (2020)
3. Wiggers, R.: State Management for Stateful Hash-Based Signature Schemes (S-HBS). IETF Draft (2023)
4. Merkle, R.C.: A Certified Digital Signature. In: Advances in Cryptology — CRYPTO '89 Proceedings, pp. 218–238. Springer (1990)
5. Hulsing, A.: WOTS+: Shorter Signatures for Hash-Based Signature Schemes. In: Progress in Cryptology – AFRICACRYPT 2011, pp. 173–188. Springer (2011)
6. McGrew, D., Curcio, M., Fluhrer, S.: Hierarchical Signature System (HSS). IETF RFC 8554 (2019)
7. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System. (2008), `https://bitcoin.org/bitcoin.pdf`
8. Miller, A., et al.: SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 104–121 (2016)
9. McGrew, D., et al.: Leighton-Micali Signatures (LMS). IETF RFC 8554 (2019)
10. National Institute of Standards and Technology: Stateless Hash-Based Digital Signature Standard. FIPS 205 (2024)

11. Haber, S., Stornetta, W.S.: How to time-stamp a digital document. Journal of Cryptology 3(2), 99–111 (1991)
12. Crosby, M., et al.: Blockchain technology: Beyond bitcoin. Applied Innovation 2, 6–10 (2016)
13. Lamport, L.: The Part-Time Parliament. ACM Transactions on Computer Systems 16(2), 133–169 (1998)
14. Narayanan, A., et al.: Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction. Princeton University Press (2016)
15. Gabizon, A., et al.: Zero-Knowledge SNARKs for C. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 924–941 (2017)
16. Ben-Sasson, E., et al.: Zk-SNARKs: A comprehensive survey. Foundations and Trends in Privacy and Security 1(1-2), 1–214 (2024)
17. Laurie, B., Langley, A., Kasper, E.: Certificate Transparency. IETF RFC 6962 (2013)
18. Laurie, B., Salz, R., Messeri, E.: Certificate Transparency Version 2.0. IETF RFC 9162 (2021)
19. Ongaro, D., Ousterhout, J.: In Search of an Understandable Consensus Algorithm. In: 2014 USENIX Annual Technical Conference (USENIX ATC 14), pp. 305–319 (2014)