

Signing Twice Is Forever: State-Management Discipline for Stateful Hash-Based Signatures Under Operational Faults

Arpan Sharma

arpansharma073@gmail.com

Independent Researcher

Research Article

Keywords: hash-based signatures, XMSS, LMS, state management, fault injection, post-quantum cryptography

Posted Date: September 4th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-10789492/v1>

License: © ⓘ This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

Signing Twice Is Forever: State-Management Discipline for Stateful Hash-Based Signatures Under Operational Faults

Arpan Sharma

Independent Researcher, Dallas, TX, USA

arpansharma073@gmail.com

ORCID: 0009-0006-9303-7102

Abstract

Stateful hash-based signatures (HBS), XMSS and LMS, are approved by NIST SP 800-208 and mandated first under NSA CNSA 2.0 for firmware and software signing. Reusing a one-time key index even once enables forgery, a failure mode no other approved family shares. SP 800-208 confines keys to hardware modules, yet the schemes also ship in general-purpose software libraries, where routine operations such as backups, snapshots, and failover resurrect spent state. Which software state-management disciplines actually prevent reuse, and at what cost, has not been measured. We build a replicated Java signing service over Bouncy Castle and subject four disciplines (load-at-boot, load-sign-persist, write-ahead reservation, batched leasing) to crash storms, split-brain concurrency, and snapshot rollback, using real SIGKILL and taking key reuse as ground truth. Only transactional claim disciplines eliminate reuse under crashes and concurrency; the ad hoc patterns reuse tens to hundreds of indices per run. Snapshot rollback defeats every within-store discipline, so rollback protection requires a monotonic anchor outside the store. Safety is nearly free for XMSS but catastrophic for LMS on Bouncy Castle 1.85, where the first signature after every rehydration rebuilds the Merkle tree (~ 230 ms at $h=10$). We reported this upstream; Bouncy Castle added tree-cache serialization in its 1.86 beta. Re-measured on the patched beta, the fix is real but selective: load-sign-persist falls from 250 to 7 ms, while write-ahead reservation stays near 226 ms, because building the cache for serialization now carries the rebuild. Batched leasing (131 μ s) remains the only safe and affordable LMS discipline on both versions.

Keywords: hash-based signatures; XMSS; LMS; state management; fault injection; post-quantum cryptography

1 Introduction

Hash-based signatures are the most conservative post-quantum signature option: their security reduces to the collision or preimage resistance of the underlying hash function alone, with no new number-theoretic assumptions. NIST SP 800-208 [1] approves the two *stateful* schemes XMSS [2] and LMS [3], and the NSA CNSA 2.0 suite [4] mandates them *first*, ahead of the lattice-based ML-DSA [5], for firmware and software signing, precisely because a conservative root of trust matters most for code that authorizes other code. NIST’s transition guidance deprecates quantum-vulnerable algorithms after 2030 and disallows them after 2035 [6], so these deployments are being built now.

Statefulness is the price of that conservatism. A stateful HBS private key is a finite pool of one-time signing keys indexed $0, 1, \dots, 2^h - 1$, and the scheme’s entire security rests on an invariant with no analogue in RSA, ECDSA, or ML-DSA: each index is used to sign at most once, ever. A

single reuse leaks enough of the one-time key to enable existential forgery [3, 7]. The signing key must therefore advance a persistent counter in lock-step with every signature it emits, and must do so correctly across crashes, replicas, and restores.

SP 800-208 is explicit that this is hard in practice. It requires key and signature generation to happen inside hardware cryptographic modules that do not allow keying material to be exported, and it warns that “seemingly routine operations could lead to the risk of one-time key reuse” [1], pointing to the state-management analysis of McGrew et al. [7] for the mechanisms: copying a private key, restoring one from backup, or reverting a virtual machine to a snapshot all reintroduce spent state. But the schemes also ship in general-purpose software libraries, Bouncy Castle among them, and real signing services built on such libraries are assembled from exactly the mechanisms the standard worries about: they run as replicas behind a load balancer, persist key state to shared databases, take periodic backups, and recover from crashes by restarting. The standard tells operators to keep state in hardware; it does not tell them, and the literature has not measured, which software disciplines survive which operational faults, or what the safe ones cost.

This paper fills that gap with a fault-injection study. We construct a replicated Java signing service and subject four state-management disciplines to three fault classes, using real process kills and store rollbacks, and measure one-time-key reuse directly as ground truth. Concretely, we contribute:

1. **A fault-injection methodology and open-source harness** (Section 3) that spawns real worker JVMs sharing a real transactional store, injects crashes, concurrency, and rollback from outside, and detects reuse as the appearance of one leaf index under two distinct messages, the exact precondition for forgery.
2. **A safety characterization** (Section 4) of four disciplines across three faults for both XMSS and LMS: transactional claim disciplines eliminate crash- and concurrency-induced reuse; the two common ad hoc patterns do not.
3. **An empirical confirmation of the rollback limit** (Section 4): no discipline whose state lives inside the store survives the store’s own rollback, establishing the copy hazard behind SP 800-208’s in-module requirement as a structural boundary rather than an implementation shortfall.
4. **A cost analysis, before and after an upstream fix** (Section 5). On Bouncy Castle 1.85 a Merkle-tree rebuild on the first signature after every rehydration makes the per-signature safe discipline unusable for LMS; we reported this (`bcbgit/bc-java#2365`), Bouncy Castle added tree-cache serialization to the LMS encoding (merged in the 1.86 beta), and we re-measure on the same host to show the fix is real but does not reach the write-ahead-reservation access pattern.
5. **Deployment guidance** (Section 6) mapping each discipline and fault class to concrete recommendations for HBS signing infrastructure.

2 Background and Related Work

2.1 Stateful hash-based signatures

Both approved schemes build a Merkle tree of height h over 2^h one-time signature (OTS) key pairs, WOTS+ for XMSS and LM-OTS for LMS. The construction descends from Merkle’s certified digital

signature [8], and XMSS added forward security under minimal assumptions [9]. The public key is the tree root; a signature at leaf q is the OTS signature under key q plus the authentication path of sibling nodes from leaf q to the root. The private key is the seed plus the current index q , and signing must atomically consume and advance q . Multi-tree variants (XMSS^{MT}, HSS) compose trees to enlarge the key space; the state-management problem is identical, and this study uses single trees of $h=10$ ($2^{10} = 1024$ OTS keys) to keep campaigns tractable while preserving the semantics.

The reuse hazard is quantitative. For LM-OTS with Winternitz parameter w , reusing one key across two distinct messages exposes intermediate Winternitz chain values from which a forgery on related messages can be assembled [3]. We therefore treat any index that signs two distinct messages as a realized security failure, not merely a warning.

2.2 Operational state management

SP 800-208 [1] requires stateful HBS keys to be generated and used inside hardware cryptographic modules that never export private keying material, a conformance rule aimed at the accidental-copy hazard: copying, backup restoration, and snapshot reversion all duplicate state. That treatment is being actively revisited; NIST has outlined plans to revise SP 800-208’s handling of key backups [10]. The schemes nonetheless ship in general-purpose software libraries, Bouncy Castle among them, and services built on those libraries keep state wherever the application puts it; that software regime is what this study measures. Prior deployment discussion of stateful HBS has focused on *where* state should live, hardware security modules and key-management services, and on non-reuse at the module boundary [7, 11]. Deployment is concrete: LMS is specified for signed content in CMS [12], the format behind signed firmware and software, and Kampanakis et al. evaluate LMS and SPHINCS+ for UEFI secure boot [13]. Distributed-systems treatments of monotonic counters and lease-based coordination exist independently in the reliability literature [14, 15], but their application to HBS index management has not, to our knowledge, been measured under fault injection. Empirical work on the stateful schemes has compared LMS and XMSS performance on embedded targets [16], and the stateless alternative (SPHINCS+, standardized as SLH-DSA [17]) has been benchmarked extensively [18]; neither line measures state-management disciplines under operational faults, which is the axis this study adds. It takes the stateful schemes CNSA 2.0 mandates first and asks what it takes to run them safely.

2.3 Threat and fault model

We assume an honest but fault-prone operator: no adversary controls the signing hosts, but processes crash, replicas run concurrently, and storage is restored from snapshots. The adversary is the standard EUF-CMA forger [19] who observes emitted signatures; our metric is whether operational faults hand that adversary a reused index. This isolates state-management failures from cryptographic ones.

3 Methodology

The harness (Figure 1) models a signing service as N replica processes sharing one transactional state store. Replicas are real JVMs, killed with real SIGKILL, in the spirit of Jepsen-style fault injection against real processes [20]; the store is SQLite in WAL mode with BEGIN IMMEDIATE transactions, standing in for the consistency semantics of a replicated database such as PostgreSQL or etcd. Both SP 800-208 families are exercised through the Bouncy Castle JCA providers: XMSS-SHA256 ($h=10$) and LMS-SHA256-M32-H10 with LM-OTS $w=4$. Bouncy Castle is the natural subject because it is

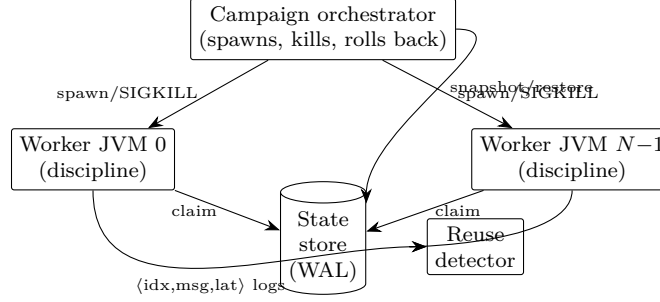


Figure 1: Fault-injection harness. Worker JVMs sign continuously against a shared transactional store; the orchestrator injects crashes, concurrency, and snapshot rollback; the detector reconstructs reuse from the union of per-signature logs.

Table 1: State-management disciplines under test.

Discipline	Persistence order	Concurrency control
naive-local	none (until shutdown)	none
persist-after-sign	sign, then write	none (last-writer-wins)
reserve-per-sig	claim 1, then sign	serialized claim + CAS
batch-lease-16	claim 16, sign ≤ 16	serialized claim + CAS

the mainstream Java provider that implements both schemes end to end; the JDK’s own providers verify HSS/LMS signatures but expose no key generation or signing for either scheme (on JDK 27 the SUN provider registers `HSS/LMS Signature` and `KeyFactory` services and no `KeyPairGenerator`). The reference C implementation of LMS/HSS [21] keeps its tree in memory for the life of the key and is not a drop-in for a replicated service, which is the setting studied here. The harness, campaign driver, per-signature logs, both Bouncy Castle versions’ campaign datasets, and the micro-benchmark probes behind Section 5 are public in the artifact repository [22].

3.1 State-management disciplines (independent variable)

Four disciplines span the space from naive to standards-aligned (Table 1). **naive-local** loads the key once and signs from memory, persisting only at graceful shutdown, modeling “read the key file at boot.” **persist-after-sign** reads state, signs, and writes the advanced state back with no concurrency control, the load-sign-persist CRUD pattern that looks correct. **reserve-per-sig** atomically claims a single index inside a transaction *before* signing (write-ahead reservation). **batch-lease- B** atomically leases a block of B indices, signs from it in memory, and re-leases when exhausted, the distributed leasing pattern SP 800-208 implies. We use $B=16$.

The transactional claim is a compare-and-set under a serialized transaction. Writing v for the stored version and s for the encoded state, a claim reads (s, v) , computes the advanced state s' and the handout shard, and commits

$$\text{UPDATE SET } s \leftarrow s', v \leftarrow v+1 \text{ WHERE } \text{version} = v, \quad (1)$$

persisting s' *before* the shard is used to sign. Algorithm 1 gives the leasing discipline; $B=1$ recovers write-ahead reservation.

Algorithm 1 Batched index leasing ($B=1$ is reserve-per-sig)

Require: key id k ; batch size B ; message m

```
1: static lease  $\leftarrow \perp$ 
2: if lease =  $\perp$  or REMAINING(lease) = 0 then
3:   lease  $\leftarrow$  CLAIM( $k, B$ ) {atomic: persist parent advanced by min( $B, \text{rem}$ ), return shard}
4: end if
5:  $\sigma \leftarrow$  SIGN(lease,  $m$ ) {consumes one index from the in-memory shard}
6: return (INDEX( $\sigma$ ),  $\sigma$ )
```

Table 2: Fault scenarios.

Scenario	Replicas	Injected fault
baseline	1	none (cost axis)
split-brain	4	concurrency only
crash-storm	2	SIGKILL + restart every ~ 2 s
snapshot-restore	2	store rolled back to a $t=6$ s snapshot

3.2 Fault scenarios (treatments)

Four scenarios (Table 2) apply to each discipline. In **baseline**, one worker signs 400 times with no faults, isolating cost. In **split-brain**, four workers sign concurrently with no fault injected; the concurrency itself is the fault. In **crash-storm**, two workers sign while the orchestrator SIGKILLS a random worker roughly every two seconds and restarts it. In **snapshot-restore**, two workers sign; the orchestrator takes an online store snapshot at $t=6$ s, then at $t=12$ s kills both, rolls the store back to the snapshot, and restarts, modeling backup or VM-snapshot recovery.

3.3 Ground truth: the reuse detector

Each worker appends one record per signature, $\langle \text{leaf index, message id, latency} \rangle$, flushed before the next iteration, so a killed worker loses at most its in-flight record (a conservative choice: a real caller may already hold that signature). The detector unions all logs, including those written before a kill or a rollback, and groups by leaf index. Writing $M(i)$ for the set of distinct messages signed under index i , the reuse count and the incident count are

$$R = |\{i : |M(i)| > 1\}|, \quad I = \sum_i (|M(i)| - 1). \quad (2)$$

$R > 0$ is a realized forgery precondition: two valid signatures under one one-time key exist in the wild. We report R as the primary safety outcome.

3.4 Environment and validity

All runs used Corretto JDK 24.0.2 on an Apple M4 (Mac16,10, macOS 27.0), with a fresh key pair per run (32 runs per Bouncy Castle version: 2 algorithms $\times$ 4 disciplines $\times$ 4 scenarios). We measure two Bouncy Castle versions on the same host: the 1.85 release, and the 1.86 beta (snapshot dated 2026-08-20) that carries the LMS fix discussed in Section 5. Bouncy Castle semantics the design depends on, that PKCS#8 round-trips carry the index, that signature encodings expose the leaf index (offset 0 for XMSS, offset 4 for LMS owing to HSS framing), and that `extractKeyShard( $n$ )` hands out the next n indices and advances the parent, are pinned by an assertion test that fails

loudly on any future Bouncy Castle change. Threats to validity: the single-host store models the consistency, not the network latency, of a replicated store, so absolute latencies are optimistic and the relative ordering of disciplines is the transferable result; SIGKILL timing relative to the sign/persist window is scheduler-dependent, so counts vary run to run while the zero-versus-nonzero safety verdict does not; and $h=10$ bounds run time without altering reuse semantics.

3.5 Experimental setup and reproducibility

The artifacts are pinned exactly. Bouncy Castle 1.85 is `bcprov-jdk18on 1.85` from Maven Central (SHA-256 `20af26bf...`); the 1.86 beta is the `bcprov-jdk18on-1.86-SNAPSHOT` jar from the Bouncy Castle betas page, retrieved 2026-08-20 (SHA-256 `8e872f94...`), recorded in the artifact repository alongside the host description (Apple M4, Mac16,10, macOS 27.0, Corretto JDK 24.0.2). The claim transaction of Eq. (1) is implemented directly on SQLite in WAL mode with an explicit serialized transaction; in outline:

```
BEGIN IMMEDIATE;
SELECT state, version FROM keystate WHERE key_id = ?;
-- decode; advance the parent by B; encode parent and shard
UPDATE keystate SET state = ?, version = ?
  WHERE key_id = ? AND version = ?;  -- must affect one row
COMMIT;                               -- only now is the shard released
```

`BEGIN IMMEDIATE` takes the write lock up front, so concurrent claimants serialize rather than race to the version check; the conditional `UPDATE` is the compare-and-set that makes a lost update impossible. The same two ingredients, a serialized writer and a versioned conditional update, are available as `SELECT ...FOR UPDATE` under a serializable transaction in PostgreSQL and as a compare-and-swap transaction in `etcd`, so the discipline transfers to a replicated store without change to its logic. The library semantics the detector depends on are pinned by an assertion suite (`HbsSemanticsTest`) that checks, for both schemes, that the leaf index can be read from a signature at the documented offset, that a PKCS#8 round-trip preserves the index, and that `extractKeyShard( $n$ )` hands out exactly the next n indices while advancing the parent; a future Bouncy Castle change to any of these fails the build rather than silently corrupting the reuse count. The full campaign on either library version regenerates from one command (`mvn package`, then the campaign runner), and the README documents the one packaging pitfall we hit, a shaded build that silently bundled the wrong Bouncy Castle version, together with the check that detects it.

4 Results I: safety

Table 3 is the core safety result: reused indices R per run for every discipline, fault, and algorithm. All `baseline` runs had $R = 0$ and are omitted. The fix of Section 5 affects only performance. Every cell was in fact run three times: the original committed campaign, and the two campaigns on the present host, on 1.85 and on the 1.86 beta of Section 5. The zero-versus-nonzero verdict was identical in all three runs for every one of the 24 cells, the eight claim-discipline cells under crashes and concurrency were zero in all three, and the nonzero cells ranged from 9 to 211 reused indices across runs. We report the 1.85 run on the present host here.

4.1 Transactional claims eliminate crash and concurrency reuse

Both `reserve-per-sig` and `batch-lease-16` achieved $R = 0$ across every `split-brain` and `crash-storm` run for both algorithms. The write-ahead ordering of Eq. (1) is why: the advanced

Table 3: Reused indices R per run (Eq. 2). X = XMSS-SHA256 $h=10$; L = LMS-SHA256-M32-H10. Bold = zero reuse.

Discipline	split-brain		crash-storm		snapshot-restore	
	X	L	X	L	X	L
naive-local	151	168	130	120	182	210
persist-after-sign	163	23	94	47	185	37
reserve-per-sig	0	0	0	0	173	9
batch-lease-16	0	0	0	0	155	96

state is committed before the index is used, so a crash after the claim wastes an index (safe) but can never reuse one, and concurrent claimants serialize on **BEGIN IMMEDIATE**. The waste is small: at most 59 of 1024 indices in the XMSS crash-storm lease run, and none for reserve-per-sig.

4.2 The ad hoc patterns fail

naive-local reused 120–210 indices per fault run, up to a third of all signatures issued, because a restarted or concurrent replica reloads the same stale starting index. More sharply, **persist-after-sign**, which *looks* correct, still reused 23–185 indices per run: it signs before it persists (a crash in that window re-issues the index) and its unconditional write loses updates under concurrency. Load-sign-persist is not meaningfully safer than no persistence at all. One caution follows from the cost results below: because the upstream fix makes **persist-after-sign** far faster on LMS, an operator who “fixes the performance” of this pattern lets it emit many more signatures per unit time and so expands rather than shrinks its reuse exposure in a fixed fault window. The re-run on the patched beta shows exactly this: LMS **persist-after-sign** reuse rises from 23–47 indices per run on 1.85 to 96–177 on the faster beta. Safety here comes from claim ordering, never from speed.

4.3 Snapshot rollback defeats every within-store discipline

Under **snapshot-restore**, all four disciplines, including the transactional ones, produced reuse ($R = 9$ –210). The mechanism is visible in the store’s final next-index falling below the count of distinct indices already emitted: signatures issued between snapshot and restore are in the wild, but the restored store re-hands those indices out. No claim discipline helps, because the claim is rolled back with the store that recorded it. If q_{snap} is the index at snapshot time and q_{max} the highest index actually signed before restore, then after restore the service re-issues every index in

$$[q_{\text{snap}}, q_{\text{max}}], \quad (3)$$

independent of the discipline. This is empirical confirmation that the copy hazard behind SP 800-208’s in-module requirement is structural: rollback safety requires a monotonic anchor *outside* the rolled-back store’s trust domain (an HSM counter, a secure element, or an append-only external log), not a better transaction against the store itself.

5 Results II: cost, and an upstream fix

Safety is necessary but not sufficient; a discipline must also be affordable. Table 4 gives baseline per-signature median latency for both Bouncy Castle versions, and here the two SP 800-208 families diverge dramatically.

Table 4: Baseline per-signature median latency (no faults), Bouncy Castle 1.85 and the 1.86 beta on the same host. Safe disciplines in bold. The 1.85 reserve-per-sig LMS run completed 255 of 400 signatures within a 180s budget; on the beta it completed all 400.

Discipline	XMSS		LMS	
	BC 1.85	BC 1.86	BC 1.85	BC 1.86
<code>naive-local</code> (unsafe)	7.01 ms	6.33 ms	0.14 ms	0.14 ms
<code>persist-after-sign</code> (unsafe)	7.05 ms	6.46 ms	250 ms	6.7 ms
<code>reserve-per-sig</code>	7.06 ms	6.29 ms	702 ms	226 ms
<code>batch-lease-16</code>	6.54 ms	6.30 ms	141 μ s	131 μ s

5.1 XMSS: safety is nearly free

For XMSS, all four disciplines sit within 6.3–7.1 ms per signature on either version. The transactional round trip is dwarfed by signing cost, and Bouncy Castle serializes XMSS’s BDS traversal state [23] in the PKCS#8 encoding, so rehydration carries no rebuild penalty: **`reserve-per-sig`**, which decodes the key on every signature, costs the same as **`naive-local`**, which decodes once (7.06 vs 7.01 ms on 1.85, 6.29 vs 6.33 ms on the beta; Table 4). On XMSS an operator can adopt write-ahead reservation with essentially no throughput penalty.

5.2 LMS on 1.85: a Merkle-tree rebuild on every rehydration

On Bouncy Castle 1.85 the per-signature safe discipline is catastrophic for LMS: **`reserve-per-sig`** runs at 702 ms per signature and **`persist-after-sign`** at 250 ms, against 141 μ s for the batched discipline. A dedicated micro-benchmark (six trials, isolated from the campaign) locates the cost. The PKCS#8 *decode* is cheap (~ 0.1 ms), but the first signature made with a freshly generated or rehydrated LMS key rebuilds the entire Merkle tree, ~ 230 ms at $h=10$, essentially the key-generation cost, while subsequent signatures on the same warm instance take ~ 0.4 ms (Figure 2, left group). Bouncy Castle 1.85’s LMS private-key encoding serializes only (version, type, OTS type, I , q , $q_{\max}$, seed) and never the tree cache, so every rehydration yields a cold instance that pays the full rebuild. The rebuild is $\Theta(2^h)$ OTS evaluations, so at production heights $h=15$ or $h=20$ it grows to seconds or minutes per cold signature. The ~ 230 ms figure is one rebuild; the per-signature disciplines trigger several. **`persist-after-sign`** pays roughly one rebuild per signature (hence 250 ms), while **`reserve-per-sig`** decodes, advances, re-encodes, and signs a cold key each time and pays about three (hence 702 ms), as the decomposition below shows. Batched leasing pays one rebuild per lease of 16 and amortizes it to 141 μ s per signature.

5.3 The upstream fix and what it does not reach

We reported this behavior upstream with a root-cause analysis and a proposed direction (bcgit/bc-java#2365) [24]. Bouncy Castle responded by adding tree-cache serialization to the LMS private-key encoding. The encoded key now appends the top six levels of the Merkle tree, nodes 1 to 63 (2,016 bytes plus a count), as optional, backward-compatible trailing data that older decoders never read; decoding primes the in-memory cache from them. A signature then needs only the part of its authentication path that lies below the cached frontier, about 32 leaf-level one-time public keys out of 1,024, which is why the first signature after a decode costs about 7 ms instead of a full rebuild. The change was merged and ships in the 1.86 beta [25]. We re-ran the entire campaign on that beta on the same host.

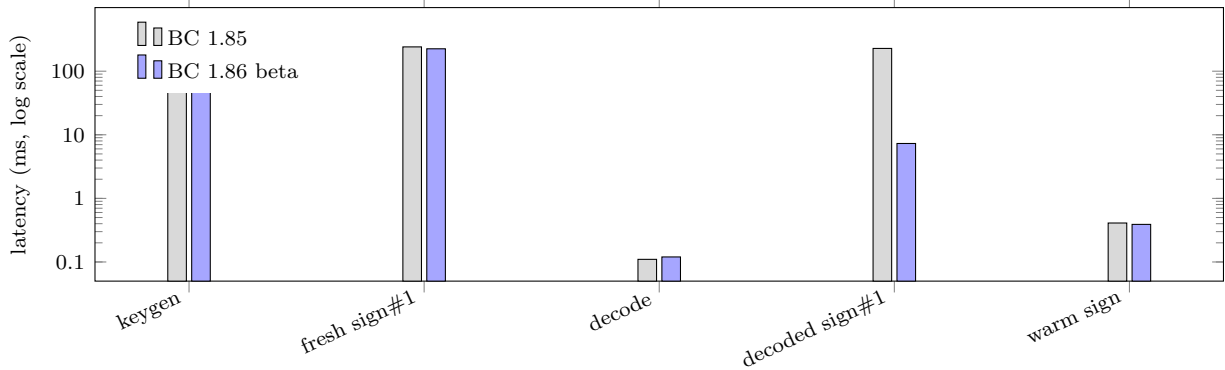


Figure 2: LMS ($h=10$) per-operation latency in Bouncy Castle 1.85 and the 1.86 beta (medians of six trials, log scale). The 1.86 tree-cache serialization removes the rebuild on the *decoded*-key first signature (~ 230 ms $\rightarrow$ ~ 7 ms) but not on the *fresh*-key first signature, which never passes through the encoding.

The fix is real but selective (Figure 2, right group; Table 4). For a key that is signed and then re-encoded, the *decoded*-key first signature drops from ~ 230 ms to ~ 7 ms, and the serialized state now carries the cache (the PKCS#8 encoding grows from 189 to 2214 bytes). **persist-after-sign**, which signs the key and then persists that same warm instance, benefits fully and falls from 250 to 6.7 ms per signature. **reserve-per-sig**, however, improves only from 702 to 226 ms and stays an order of magnitude off the leased discipline.

A per-signature decomposition explains why. On 1.85 the reservation pattern pays three separate tree builds per signature, each about 215 ms: encoding the advanced parent, encoding the single-index shard, and the signature itself. On the beta the signature drops to ~ 7 ms and the shard encode to under a millisecond, but encoding the advanced parent still costs ~ 236 ms. (The decomposition probe runs a few iterations in a fresh JVM, so its step costs sit several percent above the 400-signature campaign’s steady-state medians; the structure, three builds versus one, is the point.) The serialization the fix added made `getEncoded()` responsible for materializing the tree cache so it can be written, and the cached nodes include the root, which depends on every leaf, so an unpopulated cache means a full rebuild at encode time. **reserve-per-sig** advances the parent’s index through `extractKeyShard` without ever signing it, so the cache for the new index does not exist and `getEncoded()` rebuilds it on every claim. The rebuild did not disappear for this pattern; it moved out of the first signature and into the per-signature encode. **persist-after-sign** escapes it only because it signs before it encodes, and signing leaves the tree state current for the next index, so the encode that follows is cheap. The *fresh*-key first signature (immediately after key generation, before any encoding) is likewise unchanged at ~ 225 ms on the beta, since it never passes through the encoding the fix touches.

5.4 Batched leasing is the only practical safe LMS discipline

Leasing amortizes the cold-instance rebuild over B signatures: with $B=16$, LMS stays sub-millisecond (median 131–490 μ s across the safe scenarios on the two versions) with $R = 0$ under crashes and concurrency, at a worst-case waste bounded by one batch per kill. The upstream fix does not change the conclusion: **persist-after-sign** becomes cheap but stays unsafe, and **reserve-per-sig**, the safe per-signature discipline, stays expensive because it re-encodes the advanced key on every claim and, on the patched library, that encode now carries the tree rebuild. For LMS on current Bouncy

Table 5: Discipline $\times$ fault outcome (measured). $\checkmark$ = no reuse; $\times$ = reuse; LMS cost is baseline median on the 1.86 beta.

Discipline	crash	concurrency	rollback	LMS cost (1.86)
naive-local	$\times$	$\times$	$\times$	0.14 ms
persist-after-sign	$\times$	$\times$	$\times$	6.7 ms
reserve-per-sig	$\checkmark$	$\checkmark$	$\times$	226 ms
batch-lease-16	$\checkmark$	$\checkmark$	$\times$	131 μ s
external anchor (implied)	$\checkmark$	$\checkmark$	$\checkmark$	n/a

Castle, batched leasing is therefore not a mere optimization but a correctness-preserving necessity, the only discipline that is simultaneously safe and affordable.

6 Discussion: deployment guidance

Table 5 consolidates the recommendations. Three hold regardless of algorithm.

Use write-ahead claims, never load-sign-persist. The persist-after-sign pattern is dangerous precisely because it looks correct in code review; only a claim committed before the signature is emitted is safe under crashes and concurrency. Making it faster, as the upstream fix does, does not make it safe.

Keep LMS keys hot, and lease. Any architecture that persists an LMS key on every signature, per-request state written back to a KMS or database, or write-ahead reservation, pays a rebuild each time on 1.85, and, even on the patched beta, pays it whenever it advances and re-encodes the key without signing it first, which is exactly the write-ahead-reservation pattern. Lease index blocks into a long-lived signing process, where one rebuild is amortized over a whole batch of in-memory signatures.

Treat backups and snapshots of HBS state as key-compromise events, or anchor the index outside the store. No software discipline measured here survives rollback (Eq. 3); operational runbooks should forbid restoring HBS signing state from a backup and should provision a monotonic external counter where rollback cannot be prevented. This is the same boundary NIST’s planned revision of SP 800-208’s backup treatment is examining [10]; our measurements supply the empirical case that no within-store discipline crosses it.

What an external anchor looks like in practice. The anchor does not need to hold the key, only a counter that cannot go backwards. A TPM 2.0 non-volatile monotonic counter, incremented with `TPM2_NV_Increment` and readable by the signing service, is the most widely available such device [26]; a hardware security module holding the HBS key itself, as SP 800-208 requires for conformance, subsumes the counter. Batched leasing makes the anchor cheap to consult: the service increments the external counter once per lease of B indices and refuses to sign if the store’s index has fallen behind the counter, so the anchor is touched once per B signatures rather than once per signature, and the $B = 16$ discipline measured here pays that cost at most once every sixteen signatures. Measuring that round trip is the natural next experiment.

The rollback boundary also has an architectural reading: because no in-store discipline crosses it, HBS signing is one of the few workloads where the correct answer is to move the security-critical counter out of the application’s database entirely and into a device whose defining property is that it cannot be rolled back. This aligns with SP 800-208’s requirement of in-module state, and our results quantify why that requirement is not merely belt-and-suspenders: no in-store discipline

we measured survives all three fault classes, so a configuration that does must anchor the counter outside the store, as the standard’s hardware-module rule already arranges.

7 Threats to validity

Store fidelity. SQLite on one host reproduces the transactional consistency of a replicated store but not its network latency or partition behavior; absolute latencies are therefore optimistic and only the relative ordering of disciplines transfers. *Timing dependence.* Reuse counts depend on where SIGKILL lands relative to the sign/persist window and vary run to run; the safety verdict (zero versus nonzero R) is stable (identical across the three campaigns for all 24 cells), and the zero-reuse claims for transactional disciplines follow from the claim-before-sign ordering, not merely from the absence of observed incidents. *Library specificity and version.* The LMS cost findings are specific to Bouncy Castle; we measure the 1.85 release and the 1.86 beta, and the beta is a pre-release snapshot whose behavior could change before general availability. The re-measurement harness reproduces the comparison in one command against either jar. *Scale.* $h=10$ trees bound run time; taller trees and multi-tree HSS/XMSS^{MT} change capacity and rebuild cost but not the reuse semantics that drive the safety results.

8 Conclusion and Future Work

Stateful hash-based signatures are mandated first under CNSA 2.0, and their one defining hazard, index reuse, is an operational problem, not a cryptographic one. Measuring it directly, we find that only write-ahead transactional claim disciplines prevent reuse under crashes and concurrency; that the widespread load-sign-persist pattern is nearly as unsafe as no persistence at all; that store rollback defeats every within-store discipline and structurally requires an external monotonic anchor; and that on Bouncy Castle the safe disciplines are free for XMSS but, for LMS, collide with a per-instance Merkle-tree rebuild. We reported that rebuild upstream, Bouncy Castle added tree-cache serialization in response, and re-measuring on the patched beta shows the fix removes the cost from the sign-then-persist path but not from the write-ahead-reservation pattern, where the rebuild simply moves into the per-signature encode, leaving batched leasing the only LMS discipline that is both safe and cheap. The guidance is concrete and immediately actionable for the firmware- and software-signing infrastructure the post-quantum transition puts on the critical path.

Future work: quantifying the throughput cost of an external monotonic anchor (an HSM counter or append-only log) per lease; extending the campaign to multi-tree HSS/XMSS^{MT} and to taller trees where the LMS rebuild dominates; and replacing the single-host store with a networked PostgreSQL or Raft cluster to add partition faults.

Declarations

Funding. No funding was received for this work.

Competing interests. The author has no competing interests. The author reported the Bouncy Castle behaviour studied in Section 5 to the Bouncy Castle project as `bcgit/bc-java#2365` and has no affiliation with the project.

Data and code availability. The fault-injection harness, the campaign driver, the per-signature logs, the campaign datasets for both library versions, the micro-benchmark probes, and the assertion suite are publicly available at <https://github.com/Arpan0995/pqc-hbs-state-management> (the re-measurement dataset, probes, and the cache-structure probe are under `results/bc186-fix/`; latest commit `b04b8e2`).

Ethics approval. Not applicable; no human or animal subjects.

References

- [1] D. Cooper, D. Apon, Q. Dang, M. Davidson, M. Dworkin, and C. Miller, “Recommendation for stateful hash-based signature schemes,” NIST Special Publication 800-208, Oct. 2020. <https://doi.org/10.6028/NIST.SP.800-208>
- [2] A. Huelsing, D. Butin, S. Gazdag, J. Rijneveld, and A. Mohaisen, “XMSS: eXtended Merkle Signature Scheme,” RFC 8391, May 2018.
- [3] D. McGrew, M. Curcio, and S. Fluhrer, “Leighton-Micali Hash-Based Signatures,” RFC 8554, Apr. 2019.
- [4] National Security Agency, “Announcing the Commercial National Security Algorithm Suite 2.0,” Cybersecurity Advisory, Sep. 2022.
- [5] National Institute of Standards and Technology, “Module-lattice-based digital signature standard,” FIPS 204, Aug. 2024. <https://doi.org/10.6028/NIST.FIPS.204>
- [6] National Institute of Standards and Technology, “Transition to post-quantum cryptography standards,” NIST IR 8547 (Initial Public Draft), Nov. 2024.
- [7] D. McGrew, P. Kampanakis, S. Fluhrer, S.-L. Gazdag, D. Butin, and J. Buchmann, “State management for hash-based signatures,” in *Security Standardisation Research (SSR)*, 2016, pp. 244–260.
- [8] R. C. Merkle, “A certified digital signature,” in *Advances in Cryptology, CRYPTO ’89*, ser. LNCS, vol. 435, Springer, 1990, pp. 218–238.
- [9] J. Buchmann, E. Dahmen, and A. Hülsing, “XMSS: A practical forward secure signature scheme based on minimal security assumptions,” in *Post-Quantum Cryptography (PQCrypto 2011)*, ser. LNCS, vol. 7071, Springer, 2011, pp. 117–129.
- [10] J. Kelsey, “Stateful HBS backups: revising SP 800-208,” NIST presentation, Nov. 2024. [Online]. Available: <https://csrc.nist.gov/Presentations/2024/stateful-hbs-backups-revising-208>
- [11] P. Kampanakis and S. Fluhrer, “LMS vs XMSS: Comparison of two hash-based signature standards,” IACR ePrint 2017/349, 2017. <https://eprint.iacr.org/2017/349>
- [12] R. Housley, “Use of the HSS/LMS hash-based signature algorithm in the Cryptographic Message Syntax (CMS),” RFC 9708, IETF, 2025.

- [13] P. Kampanakis, P. Panburana, M. Curcio, C. Shroff, and M. M. Alam, “Post-quantum LMS and SPHINCS+ hash-based signatures for UEFI secure boot,” IACR ePrint 2021/041, 2021. <https://eprint.iacr.org/2021/041>
- [14] T. D. Chandra, R. Griesemer, and J. Redstone, “Paxos made live: An engineering perspective,” in *Proc. ACM PODC*, 2007, pp. 398–407.
- [15] M. Burrows, “The Chubby lock service for loosely-coupled distributed systems,” in *Proc. USENIX OSDI*, 2006, pp. 335–350.
- [16] F. Campos, T. Kohlstadt, S. Reith, and M. Stöttinger, “LMS vs XMSS: Comparison of stateful hash-based signature schemes on ARM Cortex-M4,” in *Progress in Cryptology, AFRICACRYPT 2020*, ser. LNCS, vol. 12174, Springer, 2020, pp. 258–277. IACR ePrint 2020/470. <https://eprint.iacr.org/2020/470>
- [17] National Institute of Standards and Technology, “Stateless hash-based digital signature standard,” FIPS 205, Aug. 2024. <https://doi.org/10.6028/NIST.FIPS.205>
- [18] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe, “The SPHINCS+ signature framework,” in *Proc. ACM CCS*, 2019, pp. 2129–2146.
- [19] S. Goldwasser, S. Micali, and R. L. Rivest, “A digital signature scheme secure against adaptive chosen-message attacks,” *SIAM J. Comput.*, vol. 17, no. 2, pp. 281–308, 1988.
- [20] K. Kingsbury, “Jepsen: distributed systems safety research.” [Online]. Available: <https://jepsen.io/>
- [21] Cisco Systems, “hash-sigs: a full-featured implementation of the LMS and HSS hash-based signature schemes.” [Online]. Available: <https://github.com/cisco/hash-sigs>
- [22] A. Sharma, “pqc-hbs-state-management: fault-injection harness and datasets,” GitHub repository, 2026. [Online]. Available: <https://github.com/Arpan0995/pqc-hbs-state-management>
- [23] J. Buchmann, E. Dahmen, and M. Schneider, “Merkle tree traversal revisited,” in *Post-Quantum Cryptography (PQCrypto 2008)*, ser. LNCS, vol. 5299, Springer, 2008, pp. 63–78.
- [24] A. Sharma, “LMS: first signature per LMSPublicKeyParameters instance rebuilds the whole Merkle tree ($\sim$ keygen cost); getEncoded() omits the tree cache so every rehydration pays it again,” Bouncy Castle bc-java issue #2365, Jul. 2026 (resolved Aug. 2026). [Online]. Available: <https://github.com/bcgit/bc-java/issues/2365>
- [25] Legion of the Bouncy Castle, “Bouncy Castle Java beta releases (LMS private-key tree-cache serialization, 1.86 snapshot).” [Online]. Available: <https://downloads.bouncycastle.org/betas/>
- [26] Trusted Computing Group, “Trusted Platform Module Library Specification, Family 2.0, Level 00, Revision 01.59,” Part 1: Architecture, and Part 3: Commands (TPM2_NV_Increment, Section 31.8), November 2019. <https://trustedcomputinggroup.org/resource/tpm-library-specification/>