POST-INCIDENT REPORT / 06 SEPTEMBER 2026

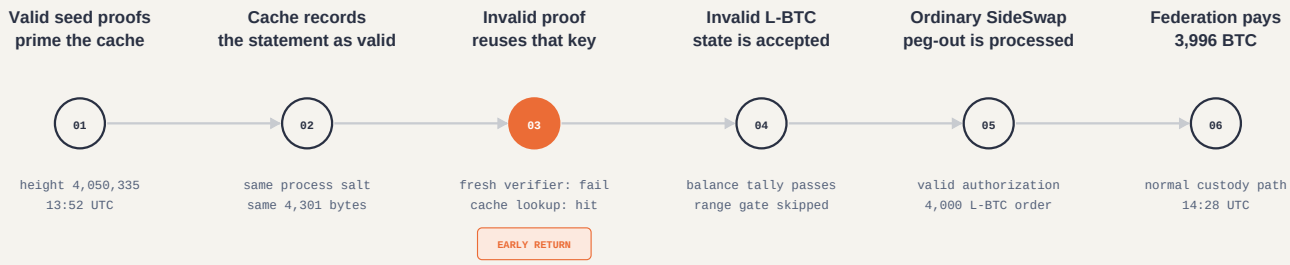
How an ambiguous cache key admitted invalid L-BTC

CAUSE

Elements cached range-proof success under a key formed by concatenating variable-length fields without boundaries. A valid seed and a different invalid target therefore became indistinguishable to the cache.

One validation shortcut became a 3,996 BTC payout

Observed sequence; SideSwap enters only after invalid L-BTC was accepted by Liquid.



Independent post-incident review by Alpen Labs using its internal Multi-Agent Security Harness. All reproduction artifacts referenced in the report are local-only. No transaction is ever signed or submitted.

SECTION 01

Executive Summary

On September 6, two outputs with valid rangeproofs were confirmed on Liquid, followed one block later by a crafted target output. Liquid consensus should have rejected the target because its rangeproof is invalid. The source and retained replay explain its acceptance with high confidence: the Elements validation cache could treat the target as the already-verified seed and return success before running the cryptographic check. The target's chain acceptance is observed; the production cache execution is strongly inferred. Outputs descended from that transaction reached SideSwap's peg-out service as a 4,000 L-BTC customer order; the federation later paid 3,996 BTC. This attack began with ordinary transaction-submission capability. It did not require theft of another user's credentials, compromise of SideSwap or its peg-out authorization key, compromise of federation Bitcoin keys, a SHA-256 collision, or a break in the rangeproof cryptography.

The cause was an ambiguous cache-key encoding introduced by Elements change c26d719. The change was intended to bind a cached rangeproof result to the proof, value commitment, effective asset generator, and output script. It fed those fields to SHA-256 as raw adjacent bytes, without lengths or fixed-size field hashes:

```
SHA256(process_salt || proof || value_commitment || asset_generator || scriptPubKey)
```

Because the proof and script are variable-length, different four-field statements can serialize to the same byte string when boundaries are not encoded. The retained incident comparison found identical encoded material for a valid statement and a different invalid statement. This was a collision in the encoding before hashing, not in SHA-256.

I reviewed the exact cache source, the retained public transaction bytes, the machine-derived transaction graph, the direct secp256k1-zkp verification results, and the actual-cache replay records. The retained local replay used the upstream `c26d719` cache wrapper: the seed passed fresh verification and was cached; the target failed fresh verification; after seed priming, the cache accepted the target. An independent Pedersen tally also passed for the target transaction, and spent-output edges connect its relevant outputs to the later peg-out. Production validator binary hashes and historical cache contents were not available, so the exact deployed build is strongly inferred rather than binary-proven. Confidential values also prevent a public determination of the exact amount created in the origin transaction. The 4,000 L-BTC figure is confirmed at the downstream SideSwap order, not at the origin.

The earliest exact vulnerable implementation verified in this review is `c26d719`, together with equivalent branch backports identified before the evidence cutoff. Available source history does not establish a first affected release or a fixed release. Elements 23.3.3 used an older, separately unsafe cache key, but the incident's exact seed and target do not collide under that older formula. No public revision correcting this ambiguous serialization was identified by the cutoff.

SECTION 02

Background

The security boundary

Liquid confidential transactions hide output amounts with Pedersen commitments. A commitment tally can prove that the input and output group elements balance, but that equation alone does not prove that every hidden amount is non-negative and within the allowed range. Each confidential output therefore carries a rangeproof. Elements verifies that proof against four pieces of context:

- the rangeproof bytes;
- the output's value commitment;
- the effective asset generator; and
- the output script, supplied as extra committed data.

The rangeproof is the boundary that prevents a balancing output from representing a negative value. If every proof is checked, an actor cannot make one output carry a negative offset and use that offset to enlarge another, spendable output while preserving the overall commitment equation.

The relevant topology is small:

Component	Role in the incident	Expected behavior
Liquid transaction submitter	Controls transaction fields and ordering of seed and target submissions	May submit transactions, but cannot make validators accept an invalid rangeproof
Elements validator	Checks confidential output proofs and the transaction commitment tally	Rejects an output whenever fresh rangeproof verification fails
Rangeproof cache	Reuses a prior successful verification for exactly the same statement	A cache hit must be equivalent to rerunning the complete verification
SideSwap peg-out service	Accepts L-BTC already recognized by Liquid and supplies valid peg-out authorization	Processes the customer order; it is not expected to re-run a different consensus history
Liquid Federation	Pays the recognized peg-out obligation on Bitcoin	Pays after a validly authorized peg-out is accepted

The incident actor needed no special account, validator access, SideSwap credential, or federation key. The required conditions were a deployed validator path equivalent to `c26d719`, the ability to construct the seed and target fields, and ordering that caused validators to see the valid seed before the invalid target while the cache entry remained available.

Why a cache exists

Rangeproof verification is computationally expensive. Elements caches successful checks so that a proof already verified during mempool admission does not necessarily need to be verified from scratch when a block containing it connects. That optimization is safe only if the cache identity uniquely represents every input that can change the verification result.

In `src/confidential_validation.cpp`, `CRangeCheck::operator()` passes the proof, value commitment, asset generator, and script to `CachingRangeProofChecker::VerifyRangeProof`. A false result becomes `SCRIPT_ERR_RANGEPROOF`. The normal policy is therefore clear: the output is valid only when verification of that complete statement succeeds.

```
if (!CachingRangeProofChecker(store).VerifyRangeProof(
        rangeproof, val->vchCommitment, assetCommitment,
        scriptPubKey, secp256k1_ctx_verify_amounts)) {
    error = SCRIPT_ERR_RANGEPROOF;
    return std::make_pair(error, "Range proof verification failed");
}
```

This entry point is in `src/confidential_validation.cpp`, `CRangeCheck::operator()`, in the assessed source snapshot. The defect is not in this call. It is in how the callee decides that a different statement has already been checked.

SECTION 03

Vulnerability Details

An attempted context binding without boundaries

Before September 1, the rangeproof cache key covered only the proof and value commitment. That omitted the asset generator and script even though both are inputs to verification. Change c26d719, merged through PR #1592, attempted to correct that omission by appending the asset generator and script.

The decisive implementation is `src/script/sigcache.cpp`, `SignatureCache::ComputeEntryRangeProof`, in the assessed source snapshot:

```
void SignatureCache::ComputeEntryRangeProof(
    uint256& entry,
    const std::vector<unsigned char>& proof,
    const std::vector<unsigned char>& commitment,
    const std::vector<unsigned char>& asset_commitment,
    const CScript& scriptPubKey) const {
    CSHA256 hasher = m_salted_hasher_range_proof;
    hasher.Write(proof.data(), proof.size())
          .Write(commitment.data(), commitment.size())
          .Write(asset_commitment.data(), asset_commitment.size())
          .Write(scriptPubKey.data(), scriptPubKey.size())
          .Finalize(entry.begin());
}
```

`Write` streams bytes into one hash state. It does not add a field tag or length. The result is therefore a hash of one byte string, not an unambiguous hash of a four-field tuple. A process-local salt precedes the fields, but the same salt is used for both statements. If their unsalted suffixes are identical, adding the same prefix preserves that equality.

The on-chain boundary shift

Let `P_s`, `C_s`, `H`, and `S_s` be the seed proof, seed value commitment, effective L-BTC generator, and seed script. Let the corresponding target fields be `P_t`, `C_t`, `H`, and `S_t`. The incident bytes satisfy:

```
P_t = P_s || C_s || H || prefix
S_s = prefix || C_t || H || S_t

therefore:

P_s || C_s || H || S_s
  = P_t || C_t || H || S_t
```

The two-byte `prefix` is `6a43`. In the seed it begins an `OP_RETURN` script; in the target it completes the longer proof. The same bytes are divided at different boundaries:

Cache field	Valid seed	Invalid target
Rangeproof	4,166 bytes	4,234 bytes
Value commitment	33 bytes	33 bytes
Effective asset generator	33 bytes	33 bytes
Script	69 bytes	1 byte
Total hashed suffix	**4,301 bytes**	**4,301 bytes**

The bug: field boundaries were never encoded

The seed and target are different statements, but their raw concatenations are identical.

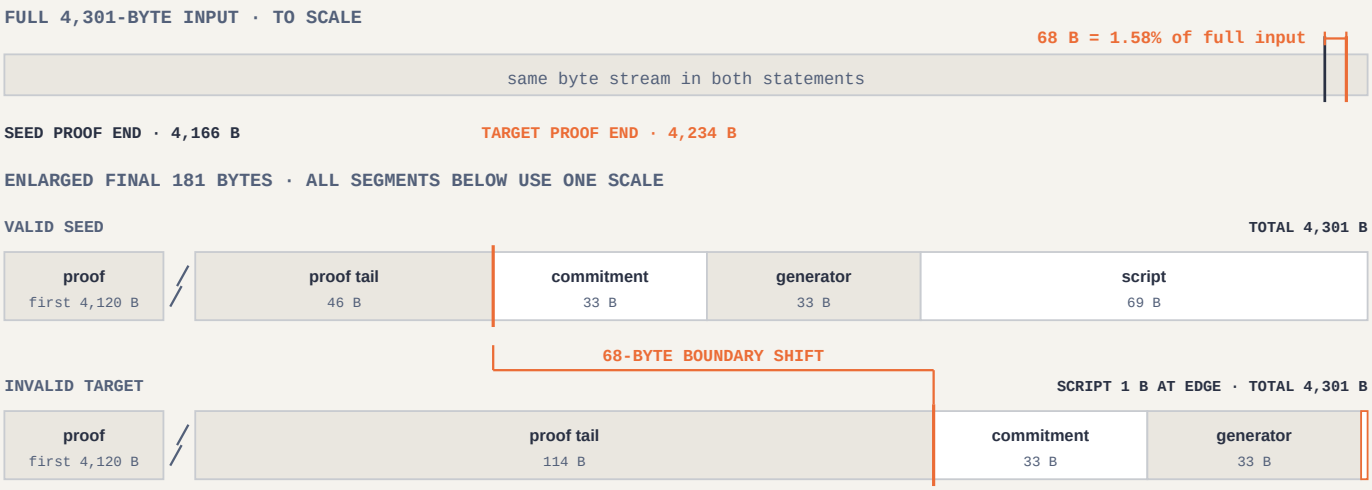


Figure 2. The bars mark the four semantic fields, and the accent line marks the shifted boundary. The byte stream has the same length and contents in both rows, but Elements assigns different meanings to its segments before verification.

The retained collision reproducer confirms byte-for-byte equality and records the common unsalted material hash as `82b0b8ccf8c743171f2bc8d80bb9982a81cfca583e8dfbe65563cf095e99c01a`. It also confirms a useful negative control: the seed and target do not collide under the older `proof || commitment` formula. See c26d719-cache-collision.json.

Two redundant valid seed outputs carry the same statement:

- 271147...7ec5:0
- 71c93d...f411:0

Both were confirmed at Liquid height 4,050,335 at 13:52:10 UTC. The invalid target is output 1 of f24a4b...183f, confirmed one block later at height 4,050,336 at 13:53:10 UTC.

A cache hit bypasses all decisive checks

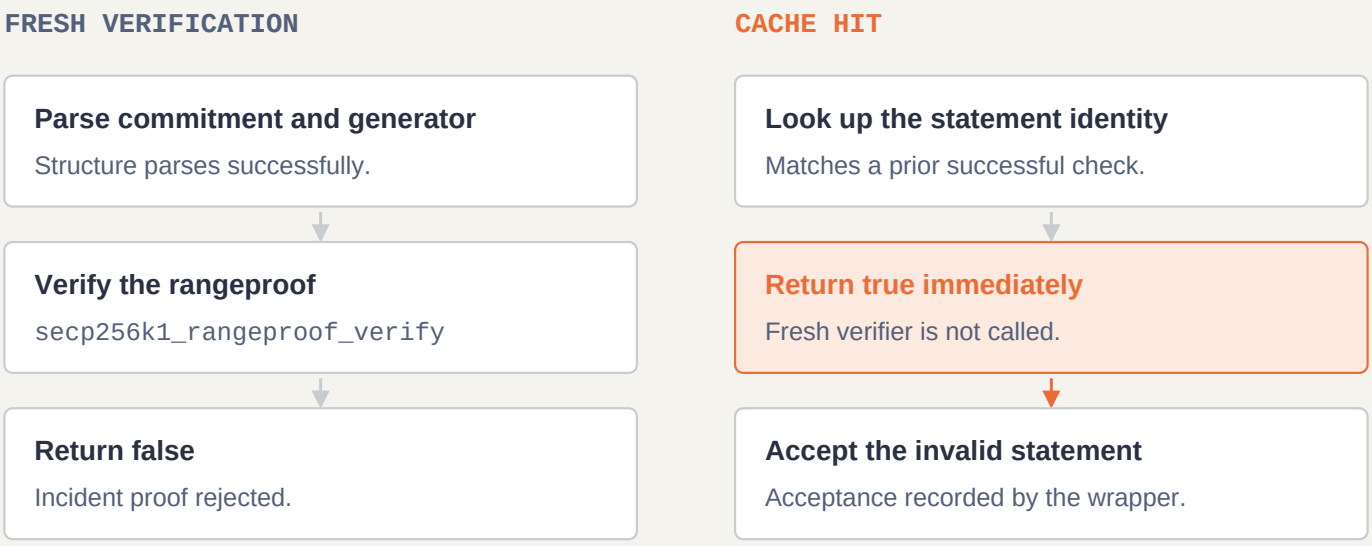
The second half of the cause is control flow. `CachingRangeProofChecker::VerifyRangeProof`, in `src/script/sigcache.cpp`, computes the ambiguous key and returns `true` immediately when it is present:

```
uint256 entry;
rangeProofCache.ComputeEntryRangeProof(
    entry, vchRangeProof, vchValueCommitment,
    vchAssetCommitment, scriptPubKey);

if (rangeProofCache.Get(entry, !store)) {
    return true;
}

if (vchRangeProof.size() == 0) {
    return false;
}

// Commitment and generator parsing follow here.
// secp256k1_rangeproof_verify follows after that.
```

On a hit, Elements skips proof non-emptiness, commitment parsing, generator parsing, `secp256k1_rangeproof_verify`, and the later minimum-value rule. The cache identity is therefore a consensus decision, not merely a performance hint.

The same incident proof receives different answers

The fresh rejection and cached acceptance are results of retained component checks.



A cache hit skips both cryptographic verification and the later minimum-value policy.

Production use of this bypass is strongly inferred; live cache records were unavailable.

Figure 3. The defective key lets a different statement enter at the cache-hit branch. The accent path never reaches fresh cryptographic verification.

The retained replay models the observed ordering against the actual upstream cache translation unit:

1. A valid seed enters the mempool path, passes fresh verification, and stores its cache entry.
2. Block validation consults the same entry with storage disabled and marks it eligible for later collection.
3. Elements' `cuckoocache` still returns entries marked for collection until space is needed, so the entry remains findable after that lookup.
4. The target computes the same key. The checker returns `true` without verifying the target proof.

The two identical seeds plausibly increased priming reliability across validators, but mempool arrival and live cache state are not recorded on-chain. Their exact operational role is therefore inferred. The cache behavior itself is source-confirmed and reproduced.

How the bypass permits inflation

The target transaction `f24a4b...183f` has one input and four outputs. Its overall Pedersen commitment tally passes: the retained direct check records one positive term, four negative terms, and `tally_ok: true`. That result proves the group elements balance; it does not prove each hidden amount is in range.

Output 1 declares the explicit L-BTC asset, uses a confidential value commitment, and has the unspendable script `6a` (`OP_RETURN`). Its rangeproof fails a fresh secp256k1-zkp verification. Because the asset is explicit, this output does not require a surjection proof. With its rangeproof bypassed, its commitment can supply the negative offset needed to balance an enlarged positive L-BTC commitment in a spendable output. Output 0 is spendable and its own rangeproof is valid. Output 2 is also spendable and has a valid rangeproof in the retained direct check; its surjection proof was present but was not independently rechecked in this review. Output 3 is the explicit 58-satoshi L-BTC fee.

This establishes the consensus inflation mechanism without revealing the confidential values. It does not establish that exactly 4,000 L-BTC was created at `f24a4b...183f`. The confirmed 4,000 L-BTC amount appears later, when SideSwap says the customer submitted the peg-out order.

Source and release context

The public title of `c26d719` was "fix: range proof cache bind to asset and scriptpubkey." The change modified the cache implementation but did not add a regression test for ambiguous field boundaries. Equivalent changes were identified on the maintained `elements-23.x` and `elements-23.3.x` branches before the cutoff. PR #1599 merged the backport associated with preparation for `23.3.4rc2` on September 6.

This history must not be read as a verified release range. Elements 23.3.3 used the older `proof || commitment` identity, which omitted verification context and was separately unsafe; the incident pair does not collide under it. The on-chain pair specifically targets the `c26d719` field order. That is strong evidence that accepting validators ran `c26d719` or equivalent code, but production binary hashes are unavailable. No public corrected revision or fixed release was identified by the evidence cutoff.

SECTION 04

Exploitability Analysis

Narrow demonstrated primitive

Under the verified cache implementation, an actor can make one valid rangeproof statement populate a cache entry and then present a different, invalid statement that has the same concatenated key material. A cache hit changes the invalid statement's result from rejection to acceptance. This crosses the consensus validation boundary for confidential output values.

The retained controls isolate that primitive:

Control	Observed result	What it rules out
Valid seed checked without cache	Accepted	The primer is not relying on an invalid seed proof
Target checked without cache	Rejected	The target is not independently valid
Seed then target through actual cache wrapper	Target accepted	The acceptance depends on cached identity reuse
Seed and target under old cache formula	Different keys	The incident pair is specific to the new field order
Target Pedersen tally	Passes	The transaction can reach the rangeproof gate without failing value balance
181-transaction incident corpus	783 valid proofs, one invalid proof	The target is the only fresh rangeproof failure in that retained corpus

The actual-cache results are retained in actual-c26d719-cache-replay.json. The uncached batch results are in fresh-rangeproof-verification.json, and the independent tally is in f24a4b-pedersen-tally.json.

Required conditions and practical constraints

The demonstrated path requires:

1. Validators using the `c26d719` serialization or an equivalent implementation.
2. A valid seed statement and an invalid target whose raw concatenations match.
3. Seed processing before target processing in the same process-local cache lifetime.
4. The seed entry remaining findable when the target is checked.
5. A target transaction whose other consensus checks, including its commitment tally, pass.

These are meaningful constraints, but the public chain shows deliberate-looking preparation: two identical valid seed statements, confirmation one block before the target, exact cross-field byte equations, a passing target tally, and rapid consolidation. Accidental construction is not a competitive explanation. The missing production mempool logs and cache snapshots prevent measurement of how many validators were primed, how the transactions first propagated, or whether both seeds were operationally necessary.

Observed downstream impact

Spent-output edges in the retained public transaction corpus connect both relevant outputs of `f24a4b...183f` to the peg-out transaction `ce4cae...88f2`:

```
f24a4b:0 -> 3875a6 -> c6ea58:vin2
f24a4b:2 ----------> c6ea58:vin3

c6ea58:0 -> 46f117 -> ce4cae:vin4
c6ea58:1 -> 328924 -> ce4cae:vin5
```

The complete machine-derived edge set is in inflation-to-pegout-path.json.

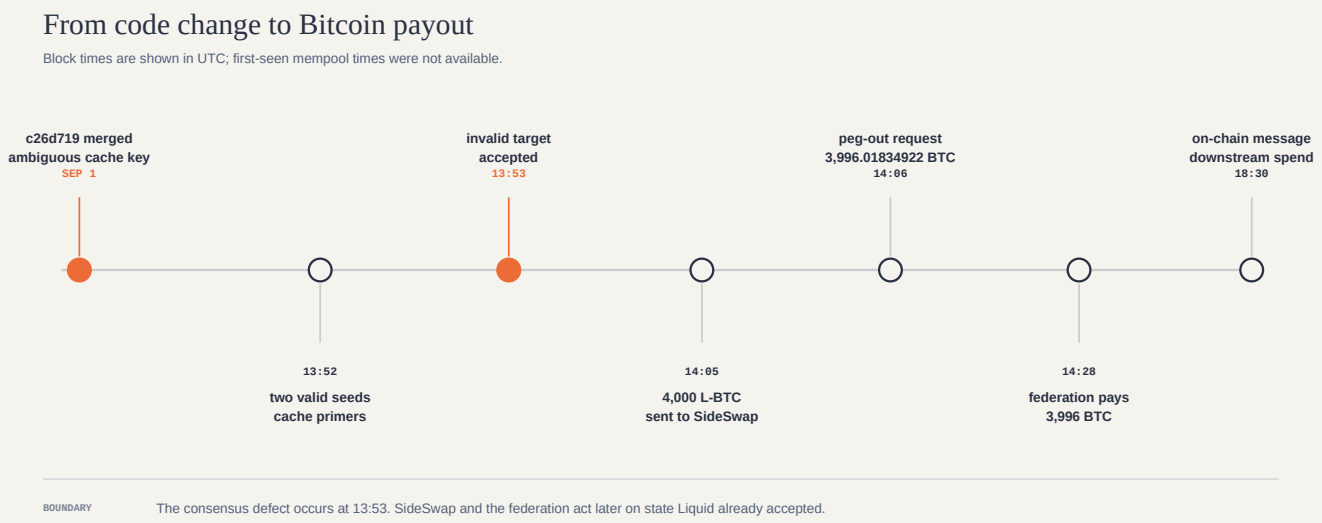


Figure 4. Times are block timestamps except SideSwap's 14:05 order time, which comes from the operator statement. They are not exact first-seen times.

Time (UTC)	Event	Evidence classification
13:52:10	Two valid seed outputs confirmed in Liquid block 4,050,335	Observed on-chain; fresh proofs reproduced as valid
13:53:10	`f24a4b...183f` confirmed in block 4,050,336	Observed on-chain; output 1 reproduced as invalid without cache
13:54:10-14:05:10	Relevant outputs consolidated toward the peg-out	Observed spent-output graph
14:05	Customer sent 4,000 L-BTC to SideSwap's peg-out service	Accepted official SideSwap account
14:06:10	`ce4cae...88f2` requests 3,996.01834922 BTC	Observed on Liquid
14:28:56	Federation transaction pays the requested amount	Observed on Bitcoin; consistent with SideSwap's 3,996 BTC statement
18:30:10	`c103de...9a19` carries the later whitehat message	Observed on Bitcoin; downstream of the payout, not the federation custody spend

SideSwap's official statement says its service handled the order normally, burned the L-BTC with valid peg-out authorization, and that neither its system nor authorization key was compromised. That account is consistent with the transaction graph and is accepted for this report. Private SideSwap service records were not inspected. The causal failure was upstream: Elements had already admitted the invalid state before SideSwap received the coins.

Alternative explanations

- **SHA-256 collision:** Excluded. The seed and target byte streams are identical before SHA-256.
- **Broken secp256k1-zkp verification:** Excluded. Fresh verification correctly accepts the seeds and rejects the target.
- **SideSwap or peg-out key compromise:** Not required, inconsistent with the adopted operator account, and unsupported by the chain path. The authorization can be valid for coins already accepted by Liquid.
- **Federation Bitcoin-key compromise:** Not required. The Bitcoin payment follows the recognized peg-out path.
- **Random malformed data or database corruption:** Inconsistent with the exact boundary equations, duplicate primers, one-block ordering, passing tally, and consolidation path.
- **A different Elements defect:** Logically possible without production binaries, but it would have to coexist with public bytes that exactly reproduce this source-level bypass. It is not a competitive cause on the present evidence.

SECTION 05

Local Reproduction

Retained artifacts

The reproduction is local and read-only with respect to both networks. It uses public confirmed transaction bytes and vendored Elements and secp256k1-zkp source. No transaction was constructed, signed, or submitted.

Artifact	Purpose
reproduce_incident_cache_collision.py	Reconstructs both four-field statements and proves their concatenated bytes are equal
verify_rangeproofs.c and verify_incident_rangeproofs.py	Verify incident-corpus rangeproofs directly without the Elements cache
incident_cache_replay.cpp and run_incident_cache_replay.py	Build and execute the actual cache-wrapper path against the public seed and target
verify_pedersen_tally.c and verify_incident_tally.py	Reconstruct and verify the target transaction's commitment tally
trace_incident_path.py	Reconstruct spent-output edges from the target to the peg-out
exact-root-cause-manifest.json	Pins hashes and roles for the source, scripts, public transaction bytes, and result records

Reproducing the byte collision

From the report directory, with Python 3 and the retained repository layout:

```
cd ..
python3 analysis_tools/reproduce_incident_cache_collision.py \
  --hex-dir evidence/chain/ce4cae-ancestry/hex \
  --elements sources/elements \
  --verification evidence/chain/ce4cae-ancestry/fresh-rangeproof-verification.json \
  --output post-incident-report/reproduced-cache-collision.json
```

The decisive retained output is:


```
{
  "seed_fresh_check_valid": true,
  "target_fresh_check_valid": false,
  "c26d719_material_byte_for_byte_equal": true,
  "pre_c26d719_material_equal": false,
  "c26d719_material_sha256": "82b0b8ccf8c743171f2bc8d80bb9982a81cfca583e8dfbe65563cf095e99c01a"
}
```


This command parses retained public transactions and writes one local JSON result. It does not contact a node or network.

Replaying the vulnerable control flow

The native replay requires the pinned Elements source tree, a C and C++20 toolchain, the platform SDK expected by the helper, and the retained public fixture. Its driver refuses a different Elements revision or modified tracked source. On the recorded run, every compile and link step exited successfully. The actual wrapper returned:


```
{
  "actual_c26d719_entries_equal": true,
  "seed_fresh_check_valid": true,
  "target_fresh_check_valid": false,
  "seed_mempool_result_cached": true,
  "entry_survives_seed_block_lookup": true,
  "invalid_target_accepted_after_priming": true
}
```


This is a controlled local reproduction of the validation decision, not a claim that a new transaction was accepted by production during testing. The exact deployed validator binaries and their historical in-memory cache state remain unavailable.

SECTION 06

Remediation

Immediate containment

1. Keep block production, peg-ins, peg-outs, and exchange L-BTC deposits and withdrawals paused until validators have a safe build and operators agree on state recovery.
2. Preserve validator binaries, build provenance, process start times, mempool logs, validation logs, functionary records, SideSwap order records, and available unblinding data. The historical cache itself is volatile.
3. Do not deploy `c26d719` or its direct branch backports as the fix. They contain the vulnerable serialization.

Repair the cache invariant

The emergency-safe change is to disable rangeproof cache hits and run the complete verifier for every statement. A durable cache can remain only if its identity is an unambiguous encoding of every verification input and its hit means that the complete decision would be identical.

One suitable design is to hash each semantic field separately, then hash fixed-size field digests in tagged, versioned slots. The following is illustrative proposed C++, not an inspected upstream fix:

```
static uint256 HashField(Span<const unsigned char> field)
{
    uint256 digest;
    CSHA256().Write(field.data(), field.size()).Finalize(digest.begin());
    return digest;
}

void SignatureCache::ComputeEntryRangeProof(
    uint256& entry,
    const std::vector<unsigned char>& proof,
    const std::vector<unsigned char>& commitment,
    const std::vector<unsigned char>& asset_commitment,
    const CScript& scriptPubKey) const
{
    const uint256 proof_hash = HashField(proof);
    const uint256 value_hash = HashField(commitment);
    const uint256 asset_hash = HashField(asset_commitment);
    const uint256 script_hash = HashField(scriptPubKey);
    static constexpr unsigned char version = 1;

    CSHA256 hasher = m_salted_hasher_range_proof;
    hasher.Write(&version, 1)
          .Write(proof_hash.begin(), 32)
          .Write(value_hash.begin(), 32)
          .Write(asset_hash.begin(), 32)
          .Write(script_hash.begin(), 32)
          .Finalize(entry.begin());
}
```

Canonical length-prefixing of every variable field is another valid design. Whichever design is selected should also:

- validate commitment and generator structure before a lookup;
- include an explicit cache domain and version;
- cache only the result of the complete proof and minimum-value decision;
- ensure all validators restart after deployment so pre-fix entries cannot remain; and
- avoid sharing identities across consensus rules that can produce different answers.

No fixed release had been verified by the evidence cutoff, so the snippet above must not be treated as shipped remediation.

Regression tests

Add the two public seed outputs and the invalid target as a permanent regression fixture. The test must assert that:

1. Both seeds pass a fresh check.
2. The target fails a fresh check.
3. The two identical valid statements share a repaired cache identity; the different invalid target has a distinct identity.
4. Processing a seed through mempool and block paths does not change the target result.
5. The result is unchanged across process restart and cache-disabled validation.
6. The exact old-key and `c26d719` counterexamples remain covered so neither omitted context nor ambiguous boundaries can return.

Property tests should vary proof and script lengths and cover field-boundary ambiguity. Identical verification statements may reuse a result; different statements must not inherit an incorrect cached success. Canonical serialization must preserve field boundaries; a design using field hashes additionally relies on hash collision resistance.

Chain and economic recovery

A fixed cache prevents future acceptance but does not repair an already-accepted invalid state. Operators should revalidate with the cache disabled from before height 4,050,335; a corrected validator should reject `f24a4b...183f` at height 4,050,336. A full-chain uncached reindex is needed to look for earlier uses of this or the older omitted-context weakness.

Network recovery then requires an explicit choice: a coordinated reorg to a state before the invalid transition, or a narrowly specified historical state repair. Restarting patched software alone cannot make the existing invalid block valid. Recovery planning should also:

- trace every descendant of the origin outputs, not only the SideSwap path;
- reconcile L-BTC supply against Bitcoin reserves;
- use authorized unblinding and accounting data to determine the amount created;
- separate returned or recovered funds, unrecovered Bitcoin, legitimate post-incident transactions, and recapitalization; and
- publish affected binary hashes, the repaired invariant, regression coverage, and the recovery rule.

Downstream velocity limits and reserve-relative peg-out limits may reduce loss from a future consensus failure, but they are containment controls. They do not repair invalid L-BTC admitted by validators, and this incident does not by itself establish a reason to rotate SideSwap's authorization key.

SECTION 07

Summary

The incident actor used ordinary Liquid transaction access to exploit a consensus-critical cache identity. Elements change `c26d719` hashed a variable-length proof and script beside two commitments without encoding field boundaries. Two different statements therefore became the same cache entry. The retained replay shows the target taking an early success path and skipping the rangeproof verifier that would otherwise reject it. The same mechanism is strongly inferred to explain the observed production acceptance.

Direct uncached verification, an actual-cache replay, an independent Pedersen tally, and the public spent-output graph establish the failure and its path to the later peg-out. SideSwap then processed a 4,000 L-BTC order with valid authorization, and the federation paid 3,996 BTC. The evidence does not show compromise of SideSwap, its authorization key, federation Bitcoin keys, SHA-256, or secp256k1-zkp.

The exact deployed validator binaries, origin amount, total inflation, and any earlier use remain unknown. No fixed release was verified by the cutoff. The next decisive work is therefore to deploy an unambiguous or disabled cache, restart every validator, revalidate history without the cache, recover to an explicitly agreed valid state, and complete the supply and reserve reconciliation.