Hardening Lightning
Harder, Better, Faster Stronger

Olaoluwa Osuntokun
@roasbeef
Co-Founder, Lightning Labs
Table of Contents

I. Overview of Lightning's Security Model
II. Hardening Contract Breach Defense
III. Reducing Client Side History Storage
   A. New Channel Design!
IV. Scaling Outsourcing (WatchTower++)
V. On-Chain Succinctness
   A. Better Fee Control
Lightning’s Security Model

- Lightning uses the underlying chain as **dispute mediation** system
  - Contract **creation+enforcement** happen on-chain
  - Contract **execution** happens off-chain
  - Secure chain acts as the **trust anchor**
- “The easy way ☑️, or the hard way ☐”
  - **Optimistically** we only require on-chain **enforcement** in the case of a **dispute**.
- In this case of a **dispute**, we assume can write to chain **“eventually” * (T)**
  - **T = csv_value**
  - **Configurable**, determines chain watching frequency
- We assume participants don’t **collude** with **pool-operators/miners**
  - **Strong non-censorship** assumption

### The Layers of Lightning

- **Application Layer (Layer 4)**
- **End-to-End Routing Layer (Layer 3)**
- **Channel Link Layer (Layer 2)**
- **Blockchain (Layer 1)**
Hardening Contract Breach Defense (Strategy)

- **Active breach defense:**
  - Watching chain in order to *refute invalid state attestation*
  - **Provide evidence** to chain showing *violation of signed contract*
  - “Blockchain as judge” model

- **In face of large backlog** (or low fee rate):
  - Possibility unable to confirm *justice tx* in time
  - Failure to provide *evidence*, allow *cheater to succeed 😞*

- Cheater *locked* into *fixed fee* due to commitment structure

- **Cheater** only has *access* to their *active balance*

- Defender has access to *entire balance* in channel

- **Scorched Earth** approach:
  - Strategy: iteratively *siphon* cheater’s *funds* to *fees*
  - End game: *all cheater’s funds go to miners, defender made whole*
    - Cheater needs to pay *fees > balance* to “succeed”
  - Assumes widespread replacement by fee rate!
Reducing Client Side History Storage

- **Lightning contract execution** is local:
  - Clients need to **store current state**
  - Prior states for possible on-chain breach remedy enforcement
  - “Shadow chain” only manifested on mainchain in event of breakdown
- **Channels state transitions:**
  - Update to current **commitment** (add/remove HTLCs, new contracts, etc)
  - Each state transition produces a new state
  - Need to **keep track of portion of prior state** in order to be able to enforce
- **Goal:** reduce per-channel state scales entire system
  - Allows for “lighter” light clients
  - High throughput backbone nodes may see considerable savings
- **Minimally**, one needs to **store the current state**
  - **Required** to be able to go on-chain for enforcement/delivery
● Commitment Invalidation
  ○ Each state transition, **invalidates** the prior state
  ○ **Critical** for safety of bi-di channels
  ○ Incentives promote forward progression
    ■ If you violate the contract, you get **slashed**!
    ■ **Worst-case enforcement**: party attempts to claim prior state
  ○ Naive method: keep all prior states!
    ■ In **high-velocity** scenario, quickly becomes **untenable**
    ■ Would then require **frequent resets** to abandon prior state **unnecessary** on-chain control traffic!

---

State #1  State #2  State #3

---

State Sequence
Reducing Client Side History Storage - History of Succinct Commitment Invalidation

- History of prior commitment invalidation mechanisms
  - **Decrementing** sequence locks (utilizes **BIP 68**)
    - **How**: use relative time-locks s.t latest state can go in **before** prior states
    - **Drawback**: limits number of possible **updates**
  - Commitment **invalidation tree** (used in **Duplex Payment Channels** (cdecker))
    - **How**: structure commitments in **tree** s.t **parent must be broadcast before leaf**
      - Roots have **decrementing** time lock w/“kick-off” allows for indefinite lifetime
    - **Drawback**: at **cost of increased on-chain footprint**
  - Commitment **Revocations** (hash or key based, current channel design)
    - **How**: must reveal secret of **prior state** when accepting new state
    - **Drawback**: **MUST critically store** $O(\log N)$ of remote party, more complex key derivation
      - **Asymmetric state**
- Goal: new commitment design with **symmetric state**, constant storage for prior states
Lightning 1.0 uses **key-based** commitment invalidation:
- Each party has multiple **static EC points** (we call em “basepoints”)
- Static points **tweaked** with fresh randomness for each state
- Randomness derived from a **verifiable PRF**
  - Currently used multidimensional hashchain: \( \text{shachain}(k, i) \)

**Revocation Key Derivation** (your state, my revBase):
- \( R_i = \text{revBase} \cdot \text{sha256}(\text{revBase} || \text{commitPoint}_i) + \text{commitPoint} \cdot \text{sha256}(\text{commitPoint}_i || \text{revBase}) \)
- \( \text{commitPoint}_i = \text{shachain}(k, i) \cdot G \)
- Relies on fact that: \( P = A + B = (a+b) \cdot G \)

**Key Revocation**:
- Reveal \( \text{commitSecret} = \text{shachain}(k, i) \)

**Key Validation**:
- \( \text{revPriv}_i = \text{revokeBasePriv} \cdot \text{sha256}(\text{revBase} || \text{commitPoint}) + \text{commitSecret} \cdot \text{sha256}(\text{commitPoint} || \text{revBase}) \)

**Cons** (M=num states, M = num_historical_htlcs)
- **Client Storage**: \( C + o(\log k) \)
- **Outsourcer Storage**: \( O(M) + O(N) + \log(k) \)
- Complex key derivation
Reducing Client Side History Storage - OP_CHECKSIGFROMSTACK

- Current signature validation operations in bitcoin assume an **implicit** message:
  - `message = sighash(
    transaction=vm.tx,
    sighash = sig.sighash_flags,
  )`
- Unable to verify signatures on **arbitrary** messages!
- Enter **OP_CHECKSIGFROMSTACK (CSFS)**
  - Args: `<message_hash> <sig> <pubkey>`
  - Returns: 1 if valid triple else, false
- Use cases:
  - "**Blessed**" structured messages
  - Output **delegation**
  - Oracles
Reducing Client Side History Storage - Signed Sequence Commitments

- **Signed Sequence Commitments**
  - **Generic** commitment state invalidation scheme
  - **Openings** of **signed** commitments **replaces revocation keys**

- **State transition:**
  - Reveal prior **commitment opening:**
    - open(c) = (n, r)
    - n = state number as int
    - r = randomness
  - Embed c within commitment output script
  - Sign next commitment:
    - c = commit(r, n++)
    - s = sig(A+B, c)

- **Enforcement:**
  - Present: (sig, c, n, r), s.t verify(sig, key, c) && open(c) == (n’, r) && n’ > n
  - “I know of an opening to a signed commitment (by broadcaster) of a newer seqno”
  - Each state creates new **signed sequence**, only need to **store latest** one!

- **Pros**
  - Now **O(1)** storage for client and outsourcer
  - Simpler client side implementation
  - Uses **same** BOLT #2 state machine
Scaling Outsourcing - Review of State Outsourcing

- LN assumes **decentralized** mining, **on-chain liveness**
  - On-chain censorship major issue
  - CSV value $T$ acts as **time-based security parameter**
    - Configurable on a channel to channel basis
- If **unable** to be **eternally vigilant**, can outsource to **WatchTower**
  - Under current design:
    - For commitments:
      - Send initial base points (needed to construct witness script template)
      - For each state send a **new signature** for justice transaction
      - Send description of justice tx assumed both sides use BIP 69
    - For HTLCs
      - Encrypt opaque blob with txid[:16]
  - Various **compensation/authentication** mechanisms possible
    - ZKP’s for authentication
    - Pay-per-state, only provide bonus upon action, subscription, etc
Scaling Outsourcing - Lighter Outsourcers

- Outsourcers + Signed Sequence Commitments
  - Only need single \((c, n, s, r)\) tuple per-client
  - Able to skip outsourcing states!
    - Elekrem/shachain approach have strict ordering requirement

- New Scheme:
  - Authenticate in sybil-resistant manner
    - Ring sig, send channel proof, post time-locked bond, etc
  - Authentication serves as user_id
  - For state \(N > N'\):
    - \(O_{blob} = \{\text{revType} || (c, n, s, r) || \text{revInfo}\}\)
    - Send \(O_{blob}\) to outsourcer, it replaces prior state

- As is, constant state if letting outsourcer take all funds:
  - If client wants cheater’s funds, still need sig to enforce structure
  - \(\text{revInfo}\) describes what justice tx looks like
    - Fees
    - Total compensation for outsourcer
    - Pubkey scripts
Ideally, outsourcers are **compensated:**
- They **provide a service**, require running node (light client / full-node)
- Two routes: **pay-per-state** or **retribution bonus**

**Retribution Bonus:**
- Outsourcer **negotiates** up front **% of cheater’s funds**
- **Details** (structure of justice tx) **contained** in revInfo
- Sig/covenant **fails** if **structure violated**
- Ideally, breaches are unlikely, so may not be enough

**Pay-Per-State:**
- **Pay** outsourcer for **each state** sent (able to batch, etc)
- Use **outsourcer** specific **e-cash tokens** (credits: Anton Kumaigorodski)
  - Pay outsourcer to for **batch of tokens** (over LN ideally)
  - **Unblinded** receiver tokens, unlink for initial payment
- Send token along with **o_blob** for each state

**Why not both?**
Lightning wallets have **additional storage requirements** compared to regular on-chain wallets
- **Static** state:
  - Set of open channels, derived keys, etc
- **Dynamic** state:
  - Optionally delegated to third party outsourcing

Overload outsourcer by also sending **encrypted static state backup**
- Re-use prior authentication in case of data loss
- Able to restore all prior channels + history
- Upon reconnect ascertain if on prior state
  - BOLT #2 addition, detect during chan state synchronization

Who watches the watchers?
- Use proof-of-retrievability protocol over static+dynamic states
- Outsource random challenges, provider notifies client of outsourcer infidelity
On-Chain Succinctness - 2-Stage HTLCs

- In current commitment design (BOLT#3) **CSV+CLTV decoupled** in HTLC’s:
  - Prior issue where if CSV is large, CLTV in total hop must be >
  - Solved by making HTLC claiming a 2-stage state machine
    - Off-chain multi-sig covenants
    - Attest (broadcast) -> Delay (csv) -> Claim (sweep)
  - Cons:
    - Requires distinct transaction for each HTLC
    - Must store signature for each HTLC
    - New state updates require signing+verifying N sigs (for each HTLC)
- Solution:
  - Use actual covenants in HTLC outputs!
  - Eliminates sig+verify w/ commitment creation
  - Eliminates sig storage of current state
  - Add independant script for HTLC revocation clause (reuse commitment invalidation technique!)
- Stop gap: `sighash_single + sighash_acp`
  - Allows 2-stage HTLC transitions to be coalesced into single transaction
Recent work on **multi-party channels**:
- *Scalable Funding of Bitcoin Micropayment Channel Networks*

Multi-party channels via **hierarchy** of multisig and **regular bi-di channels**:
- **Hook:**
  - Master multisig channel creation
- **Allocation**
  - Fan-out multisig funds distribution
- **Commitment**:
  - Regular multisig 2v2 channels

Keybased revocations blow up state:
- Paper’s solution: use invalidation tree for root allocation replacement
- **Cons**: large number of on-chain transactions on worst case

**Reuse Signed Sequence Commitments!**
- **Symmetric** state, maintain **constant** revocation state
On-Chain Succinctness - Fee Control

- Dangling commitment issue:
  - You need to **force close** due to channel peer inactivity
  - Currently, no way to adjust commitment fee
  - All your outputs are **time-locked** can’t use CPFP!
  - Solution:
    - **CPFP reserve hook**:
      - Channel creation specifies reserve value
      - Make portion explicit using distinct outputs w/ no delay
      - Allows either party to **anchor** a commitment

- Participants must **guess fees** ahead of time:
  - Initator uses **update_fee** message to regulate
  - Massive swings may leave **fees insufficient**
  - Solution:
    - Don’t add apriori fee to commitment transaction
    - Instead, use liberal sighash flags to allow for fee paying inputs
Thank You!

Any questions?