Towards a Formal Foundation for Blockchain ZK Rollups

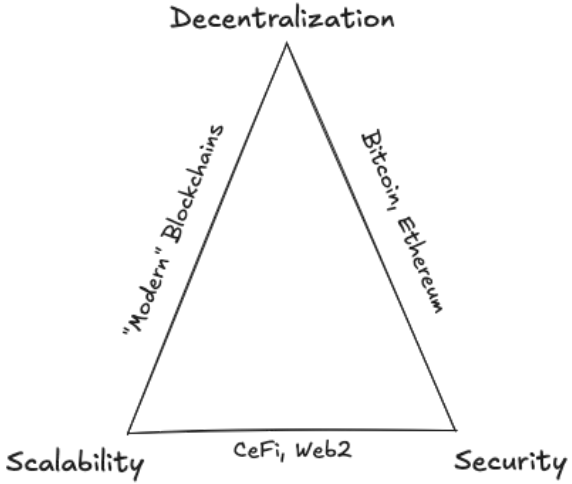
Stefanos Chaliasos, *Denis Firsov, Ben Livshits*

Imperial College London, ZKSecurity, IOHK, Eclipse Labs

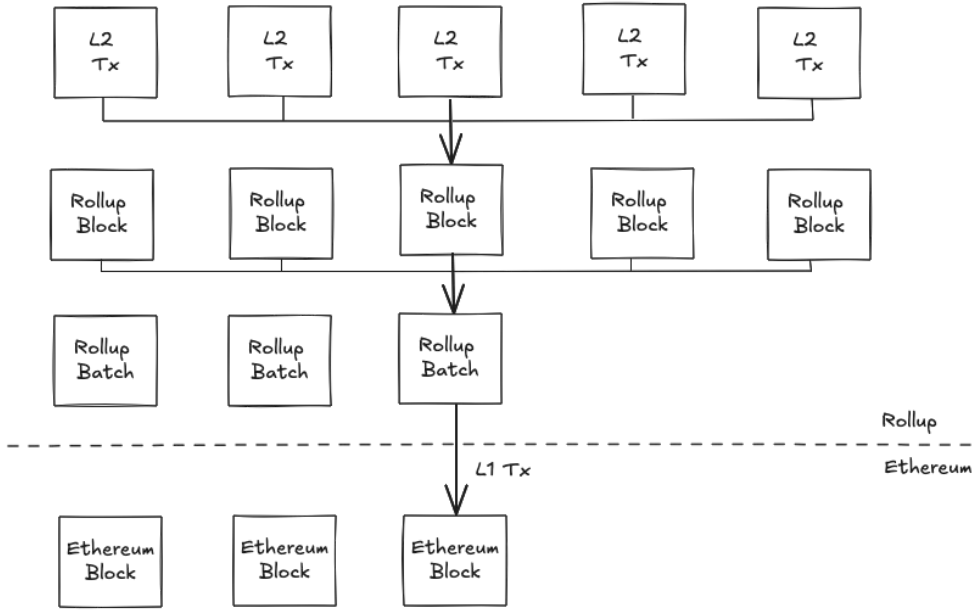
Mar 24, 2025, ZKProof Sofia

Blockchain Scalability

- Public blockchains like Ethereum and Bitcoin face throughput limitations (tens of TPS).
- The scalability trilemma: can't simultaneously optimize for security, decentralization, and scalability.
- Rollups perform computation off-chain, *inheriting L1 security* by publishing data and verifying proofs on-chain.

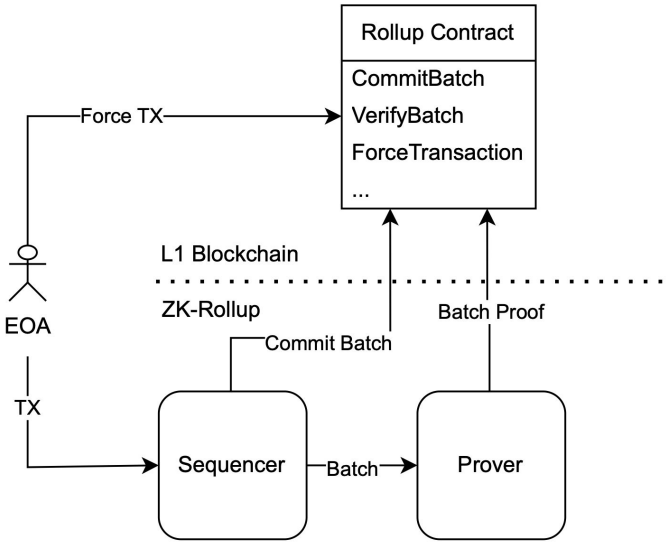


Rollups at a glance



ZK Rollups

- **Execution**: Transactions processed on L2, then compressed, posted, and verified to L1.
- **ZK (Validity) Proofs:** Cryptographic proofs assure correctness of L2 state transitions.
- **Benefits (*vs Fraud Proofs*)**: Faster finality and better compression.
- **Key Roles**:
  - **Sequencer**: Orders and batches L2 transactions.
  - **Prover**: Generates validity proofs for correctness of state transitions.
  - **L1 Contracts**: Verifies proofs, keeps canonical order of *finalized* states.



Real-World Incidents

Example of L2 concerns:

- **Blast Incident**: quick system upgrade to censor an attacker's address.
- **Linea Chain Halt**: block production suspended to blacklist compromised addresses.

Key Takeaways:

- Centralized L2 control can lead to censorship.
- Mechanisms for forced transactions and safe upgrades are crucial to preserve user security.

Intuition:

- L2s should inherit L1 censorship resistance.
- User funds should be as safe as they are in the L1.

Contributions

1. **Formal Model of Validity Proof Rollups** via Alloy specification.
2. **Three Key Mechanisms**:
    a. **Forced Queue**: ensures user transactions can't be censored.
    b. **Safe Upgradeability**: guaranteeing user can exit before malicious changes.
    c. **Safe Blacklisting**: no retroactive transaction "freezes"
3. **Counterexample-Driven Design**: illustrate and fix pitfalls
4. **Stronger Security Guarantees**: closer alignment with L1

Security Assumptions / Adversary Model

- **Validity Proofs are Correct**
  - The ZKPs used are complete and sound, both the prover and the verifier are bug-free.
- **Malicious Sequencer**
  - Censors user transactions or reorders them arbitrarily (in the L2)
- **Compromised Admins**
  - Deploy sudden, malicious rollup upgrades to divert or freeze funds
- **Honest & Malicious Users**
  - Honest: just want unstoppable transactions
  - Malicious: spamming for DoS
- **L1 is Trusted and Secure**
  - L1 finalizes transactions, enforce contract code, etc.

User-Centric Security Goals

Weak Liveness:

- If L2 is finalizing new blocks, forced transactions must be included eventually.

Upgrade Safety:

- Users have the opportunity to exit before an undesirable or malicious upgrade is enforced.

No Retroactive Censorship:

- Once forced transactions are in the queue, they can't be blocked by subsequent blacklisting or protocol changes.

Alloy for Design-Time Formal Analysis

- **Relation Logic** to describe systems state and operations.
- **SAT-Based** engine finds counter examples in bounded state space.
- Perfect for ***protocol-level*** logic verification.
- Not verifying source code but analyzing correctness of mechanisms like forced queue and upgradeability

Model

- **Inputs**: represent transactions submitted by end-users.
- **Block**: ordered sequence of Inputs
- **Commitment and proof**: We track two fields for both commitments and proofs: state and diff. The state is represented as an ordered sequence of blocks, and the diff refers to the current block for which we commit to the L1 contract and produce a proof.
- **L1**: models the L2 rollup's representation on the L1.

```
var sig Input{}
var sig Block {  var block_inputs : seq Input }

var abstract sig ZKObject {
  var state : seq Block,
  var diff : one Block
}
var sig Proof extends ZKObject{}{
  not state.hasDups
  diff not in state.elems
}
var sig Commitment extends ZKObject{}{
  not state.hasDups
  diff not in state.elems
}
one sig L1 {
  var finalized_state : seq Block,
  var commitments : set Commitment,
  var proofs : set Proof,
}{ not finalized_state.hasDups }
```

Strawman Validity Proofs Rollup Model

- **rollup_process**:
  - Enforces proof-commitment matching finalize new blocks.

Issue: No forced queue; no safe upgrade → vulnerable to censorship and malicious upgrades.

Forced Queue Mechanism

- **ForcedInput** posted on L1.
- Must be included in next finalized block if L2 wants to progress.
- **FIFO** ensures older forced transactions get finalized first.

Properties:

- **Guaranteed Inclusion**: If a block is finalized, it must include the queue's head.
- **No Skips**: The queue empties in order.

Formal Forced Queue Mechanism Properties (Alloy)

- **Guaranteed Processing**
  - If the forced queue is non-empty and a new state is finalized, then the head of the forced queue must be processed and removed.
- **Forced Queue Stable**
  - If the finalized state did not change, then the forced queue did not decrease.
- **State Invariant**
  - If the forced queue is non-empty and did not change, then the finalized state remains unchanged.
- **Forced Inputs Progress**
  - Forced inputs that were not processed move closer to the head of the forced queue.
- **Order Preservation**
  - Forced inputs retain their relative order within the queue.
- **Finalization Confirmation**
  - If an input was in the forced queue and then disappeared from it, it was finalized.

```
always (
 (some L1.forced_queue and some (L1.finalized_state'
    ↪ - L1.finalized_state))
   implies
     L1.forced_queue.first.tx in new_finalized_inputs
    and not L1.forced_queue.first.tx = L1.
    ↪ forced_queue'.first.tx
)
```

Secure Blacklisting

Design Dilemma:

- Some L2s want optional blacklisting for compliance or policies.
- But blacklisting can freeze forced queue if the head is disallowed

Solution:

- Only allow blacklisting to apply *after* forced queue items are processed
- Or require blacklisting changes to also go through the forced queue (or a safe upgrade)

Upgradeability

- **Announce Upgrade** on L1.
- **Timeout Window** for users to force-exit if they disagree.
- **Process All Forced TXs in Queue**
- **Deploy Upgrade**

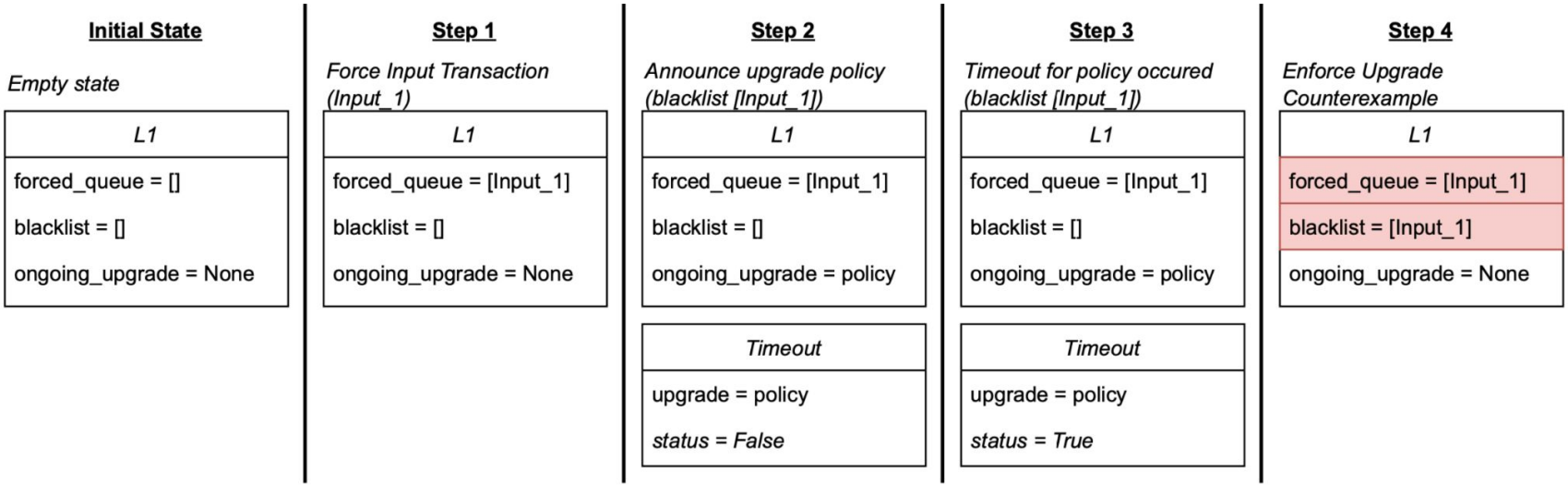
Result: Ensure no user is *left behind* with blocked forced transactions at upgrade time.

Notes: Blacklisting should be applied through upgrades, after the timeout the L2 processes *all* forced transactions and then apply the upgrade.

Counterexamples in Naive (common) Designs

- **Immediate Blacklisting**:
  - If forced queue already has blacklisted items, L2 get stuck.
- **Only Time-Based Upgrades**:
  - Attackers can spam the forced queue.
  - Honest users can’t finalize exits before the forced upgrade triggers.
- Alloy models produce explicit step-by-step states showing how the system breaks

Alloy Checks and the Counterexample Trace



Performance and Scope

- Alloy's *bounded* analysis: typically small scopes
- **Small-scope hypothesis**: many design flaws appear even in small examples

Scope 5

Mechanism	Lines of code	No. of Clauses	Solve time (sec)
Simple	295	1,203,661	9.099
Forced Queue	529	1,996,273	136.104
Blacklist	721	2,200,981	399.724
Upgradeability	970	2,468,911	388.620

Future Directions

- From source code to Alloy.
- From Alloy to source code.
- **Pragmatic Approach**: Design Spec in Alloy → FV and/or prop testing based on the invariants that have been formally proved with Alloy.
- What about security councils?

Thank You!

https://arxiv.org/pdf/2406.16219

https://github.com/StefanosChaliasos/zk-rollup-security