Breaking Blockchain Consensus via Boundary-Seeking Fuzzing of Inter-Client State Serialization

Hanzhi Liu, Yanju Chen, Hongbo Wen, Yu Feng
University of California, Santa Barbara *and* University of California, San Diego

Motivation: The Consensus Nightmare

- **The Threat:** Diverse clients (Prysm, Lighthouse, etc.) must agree on every byte of the state.
- **The Vulnerability:** Discrepancies in SSZ Serialization allow "non-canonical" but logically valid inputs to de-synchronize the network.
- **The Impact:** A single encoding bug can trigger a chain split, risking $30M+ ETH staked.

The Problem: The "Semantic Gap"

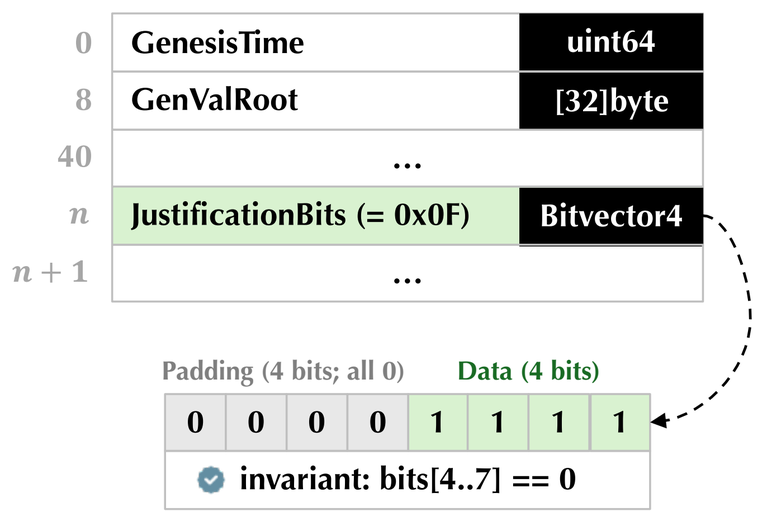
- **Why Fuzzers Fail:** Traditional tools (AFL++, LibFuzzer) focus on code coverage, but consensus bugs hide in the logic layer, not the crash layer.
- **Sparse Signals:** Valid and subtly invalid inputs share code paths, making coverage-guided search "blind".

Motivating Example: The `Bitvector` Dirty Padding Vulnerability

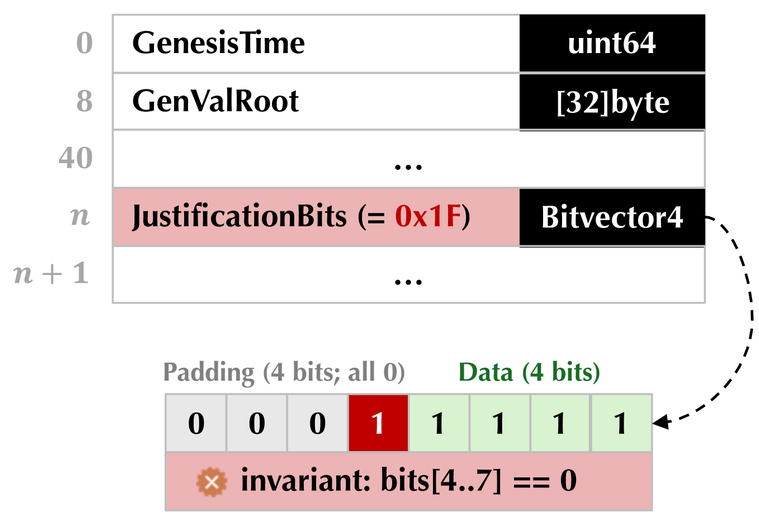
```go
type Bitvector4 []byte
func NewBitvector4() Bitvector4 {
    return make([]byte, 1)
}

type BeaconState struct {
    GenesisTime uint64
    GenValRoot  [32]byte
    // ... (Slot, Fork, Roots, etc.)
    JustificationBits Bitvector4
    // ... (more fields)
}
```

(a) SSZ Schema



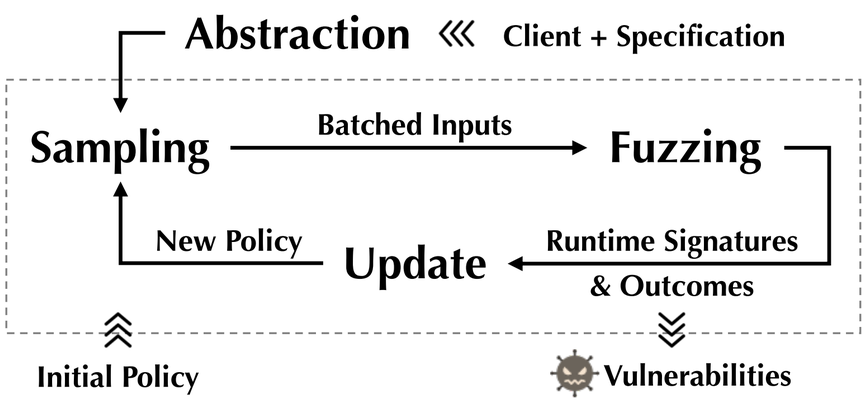
(b) A Canonical Input



(c) A Vulnerable Input

The SSZ schema (a) defines a `Bitvector4` fitting into one byte. The canonical layout (b) requires zero padding (`0x0F`). A buggy layout (c) with a dirty bit (`0x1F`) might be accepted by lenient clients, resulting in a different state root and a consensus split.

Core Innovation: How does BOLT address the problem?

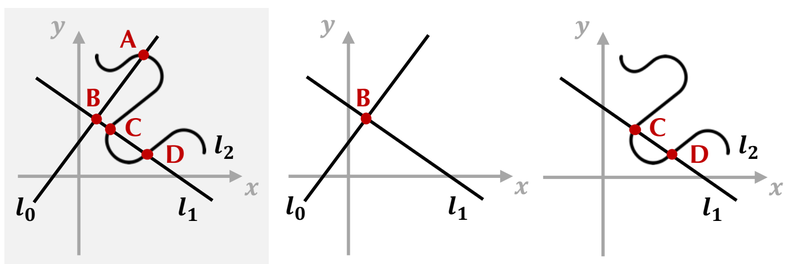


- **Input Partitioning:** Automatically derives a structured search space from SSZ types (`uint64`, `List`, `Bitvector`).
- **Constraint Tightening:** Uses a specialized operator ($\sqsupset$) to generate points exactly on and just over the validity boundaries.

- **Distributional Reward:** Instead of looking for crashes, BOLT measures the Kullback-Leibler (KL) Divergence of execution signatures.

$$r_i = \underbrace{\lambda_f \cdot \mathrm{D_{KL}}(P_i \parallel P_{i-\varepsilon:i})}_{\text{Novelty Search}} - \underbrace{\lambda_r \cdot \mathrm{D_{KL}}(P_{i-\varepsilon:i} \parallel P_i)}_{\text{Trust Region}} + \underbrace{\lambda_g \cdot R_i}_{\text{Result}}$$

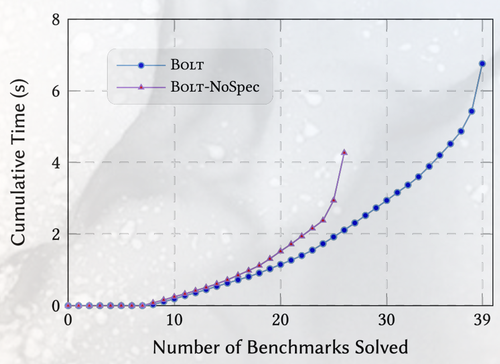
- **Result:** High rewards drive the fuzzer toward unexplored behavioral patterns and semantic edge cases.



(a) all boundaries (b) boundaries of l_0 and l_1 (c) boundaries of l_1 and l_2

Benchmarks, Performance and Ablation Study

- **Results:** BOLT identified 100% (39/39) of historical SSZ-related vulnerabilities in the benchmark suite.
- **Efficiency Metric:** Achieved a Time-to-Exposure (TTE) of less than 1ms for multiple complex logic bugs, performing significantly faster than general-purpose coverage-guided baselines.
- **Ablation Study:** Ablation results demonstrate that removing protocol-guided exploration leads to a 33% drop in detection capability and slower convergence.



Zero-Days Found

- BOLT identified 12 previously unknown vulnerabilities in production clients (Prysm, Lighthouse, Lodestar, Teku), including a critical divergence that breaks the consensus.

#	Vulnerability	Client (Issue)
1	Deserializing short/empty input	Lodestar [10] (p)
2.1	Bitvector dirty padding	Teku [20] (#10173)
2.2	Bitvector dirty padding	Prysm (fastssz) [13, 17] (p)
2.3	Bitvector dirty padding	ethereum/py-ssz [18] (p)
2.4	Bitvector dirty padding	dynamic-ssz [6] (#54)
3.1	Boolean canonicalization	fastssz / Prysm [13, 17] (#222)
3.2	Boolean canonicalization	dynamic-ssz [6] (#53)
4.1	Union trailing data accepted	Lighthouse [8] (#64)
4.2	Union trailing data accepted	Lodestar [10] (p)
4.3	Union trailing data accepted	ethereum/remerkleable [19] (p)
5	Variable-length container gap	ethereum/remerkleable [19] (p)
6	Null-bitlist trap	dynamic-ssz [6] (#56)