CS 3551: Advanced Topics in Distributed Information Systems - Building Dependable Infrastructure

Day 4: "Randomized Testing of Byzantine Fault Tolerant Algorithms"

Dr. Amy Babay, Fall 2024

University of Pittsburgh

Department of Computer Science

School of Computing and Information

The Problem

- In *theory*, BFT protocols guarantee correctness despite arbitrary behaviors from faulty nodes and temporary network delays/loss/disconnections
- But, protocols may have **bugs**
  - Logic / protocol bugs
  - Implementation bugs
- **Tools** to test correctness in the presence of both Byzantine node faults and network faults are lacking
  - Most testing tools focus on network and/or crash faults
  - State space of possible faults is very large, so generating effective test cases is challenging

Contribution

- **ByzzFuzz** is a tool to automatically find bugs in BFT protocol implementations
- Introduces **small-scope** mutations to effectively find bugs while limiting the state space (so that testing can be done in a reasonable amount of time)
- **Claim**: "the first automated testing tool that managed to discover previously unknown Byzantine fault tolerance bugs in production blockchain systems"

Approach - High Level

- Randomly inject faults with characteristics designed to quickly find bugs
- **Network faults**: *partitions*, where each network partition is isolated from all others
  - E.g. A&B can talk to each other, and C&D can talk to each other, but A&B can't talk to C&D
- **Process faults**:
  - **Message omissions**: don't send a specific message
  - **Structure-aware mutations**: manipulate message fields, not arbitrary bits
  - **<u>Small-scope mutations</u>**: keep field values *close* to their original/correct values
    - Numbers: increment or decrement by 1
    - Hashes: apply increment/decrement mutation to value before hashing, or use a hash from previous round
- Apply faults to an entire **round** (protocol step, e.g. "pre-prepare for view 1 and sequence number 1")
  - Retransmissions allowed once the sender has sent/received a message in a later round

Approach - Implementation

- Randomly generates faults to inject based on input parameters:
  - ***c* rounds with process faults**: randomly select round and subset of processes to receive mutated message
  - ***d* rounds with network faults**: randomly select round and partition
- **Network interception layer** intercepts all messages
  - For each message, determines if it should be dropped or mutated based on generated faults; randomly generates mutations

Results

- Claim: ByzzFuzz effectively detects Byzantine fault tolerance bugs in consensus implementations (RQ1)
- Evidence:
  - Detects already **known protocol bugs** from the literature:
    - PBFT liveness violation with read-only optimization
    - Ripple termination and agreement violations with insufficient UNL overlap
  - Finds **new protocol bugs**
    - New variant of Ripple agreement violation
    - ”Potential” termination violation in Tendermint (assumes messages can be buffered indefinitely and guaranteed to arrive eventually)
  - Finds **new implementation bugs**
    - Ripple termination violation (not checking hash values correctly)
    - 3 bugs in simple non-production PBFT implementation

Results

- Claim: ByzzFuzz finds more bugs than a simple baseline fault injector (RQ2)
  – Baseline fault injector: “arbitrarily injects network and process faults without the restriction to round-based structure-aware small-scope mutations”
- Evidence:
  – Only the Tendermint “potential” termination violation and the known Ripple termination violation were found by baseline fault injector

Results

- Claim: Small-scope message corruptions are effective in finding bugs (RQ3)
- Evidence: found bugs described; "any-scope" mutations are less successful in finding agreement violations

faults	T		V		I		A		Total	
baseline	41		0		0		0		41	
$c = 0,\ d = 1$	34		0		0		0		34	
$c = 0,\ d = 2$	53		0		0		0		53	
	ss	as	ss	as	ss	as	ss	as	ss	as
$c = 1,\ d = 0$	1	1	4	4	0	0	2	2	4	4
$c = 1,\ d = 1$	32	30	2	2	0	0	4	2	36	31
$c = 1,\ d = 2$	58	57	2	2	0	0	3	4	61	61
$c = 2,\ d = 0$	3	3	6	6	0	0	4	4	7	7
$c = 2,\ d = 1$	35	41	6	6	0	0	4	1	40	45
$c = 2,\ d = 2$	53	66	3	3	0	0	5	3	59	69

PBFT

faults	T		V		I		A		Total	
baseline	2		0		0		0		2	
$c = 0,\ d = 1$	11		0		0		0		11	
$c = 0,\ d = 2$	20		0		0		0		20	
	ss	as	ss	as	ss	as	ss	as	ss	as
$c = 1,\ d = 0$	9	21	0	0	0	0	1	0	10	21
$c = 1,\ d = 1$	27	20	0	0	0	0	0	0	27	20
$c = 1,\ d = 2$	19	23	0	0	0	0	1	0	20	23
$c = 2,\ d = 0$	31	25	0	0	0	0	0	0	31	25

Ripple

Future Work - Discussion

- Generalized “plug-and-play” approach
  - Or, at least step-by-step process to apply the framework
  - Apply to: Network interception layer, Output formatting / analysis
  - Are changes to message structure needed?
- Apply to other protocols
  - Prime
  - PBFT but many different implementations – what are the most common bug types?
  - Multileader / Leaderless – are there fewer bugs? (since most observed violations seem to arise from Byzantine leader behavior)
- How can we use ML / AI in BFT testing?
- Expanding fault scenarios
  - Asymmetric partitions are realistic for blockchain
  - Can we better quantify the impact of small-scope mutations? What if we compare against other types of mutation (min/max, addition/subtraction)? See message mutation strategy in “Turret: A Platform for Automated Attack Finding in Unmodified Distributed System Implementations”
  - Consider trade-off between expanding scenarios and runtime / time to find violations