Interactive symbolic testing with TLA$^+$, Apalache, and LLMs

igor@konnov.phd – Vienna, Austria

TLA$^+$ Community Meeting — Turin, Apr 12, 2026

APALACHE

Symbolic model checker for TLA+

2016 2017 2018 2019 2020 2021 2022 2023 2024 2025 2026

informal SYSTEMS

TLA+ FOUNDATION

TU WIEN

W|W|T|F

INTERCHAIN

Inria

vienna business agency

Epoch 1

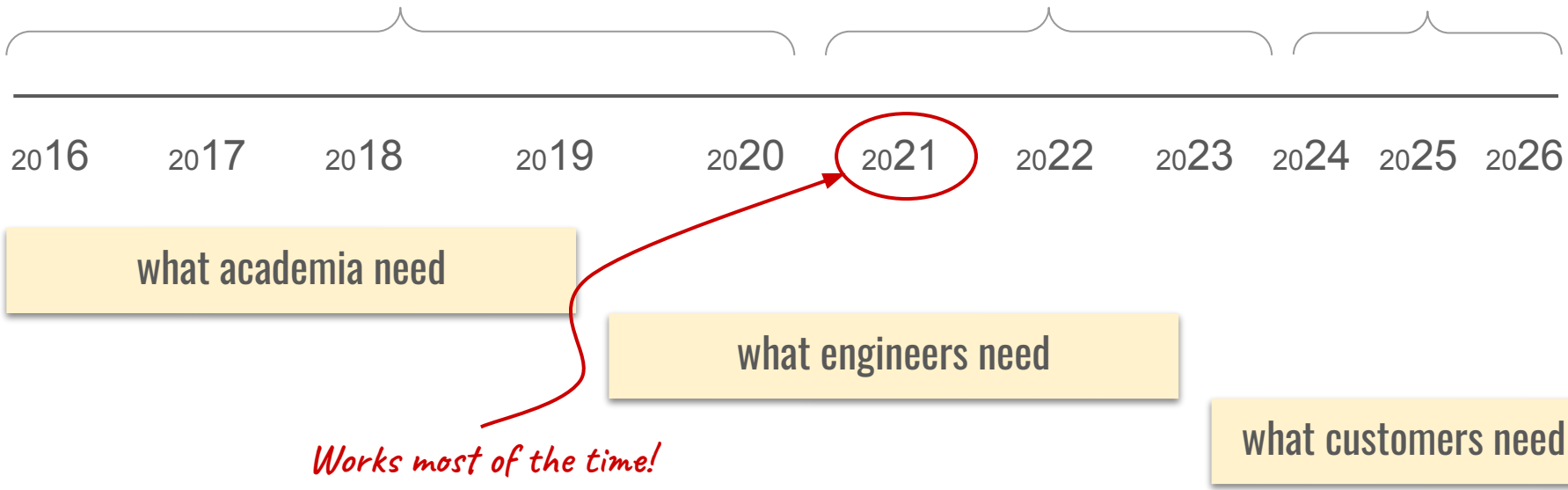
1. SMT transpiler
2. Type checker
3. Bounded model checker

Epoch 2

1. Random. symbolic execution
2. Data generators

Epoch 3

1. Parallelization
2. **DIY search**
3. **Test harnesses**

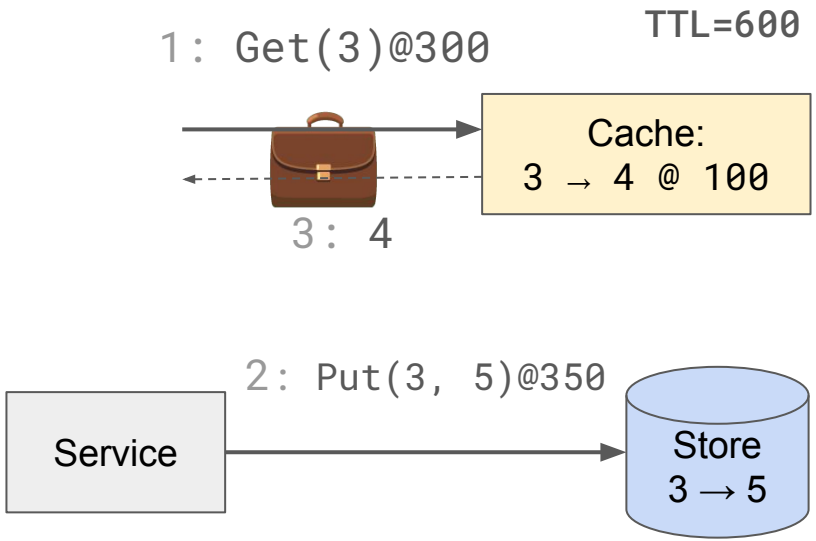


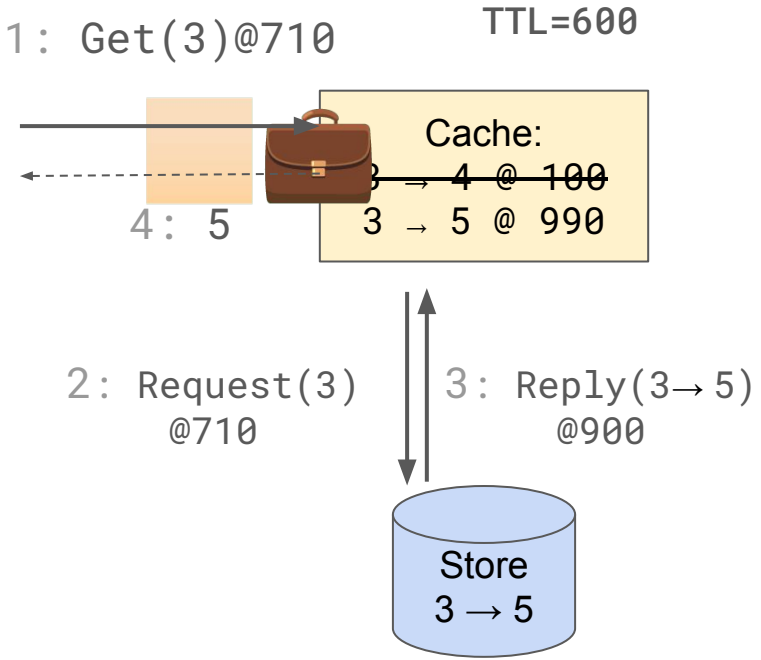
This talk

1. **New API for symbolic search** in Apalache
2. **Experiment 1**: Test TFTP with Copilot + Sonnet 4.5
3. **Experiment 2**: Test ZooKeeper with Claude & Codex

Symbolic search with Apalache

Cached key-value store





```
65  \* Another service updates the value in the store directly.
66  Put(key, val) ≜
67      Put::
68      ∧ store' = [ k ∈ DOMAIN store ∪ {key} ↦
69              IF k = key THEN [val ↦ val, ts ↦ now] ELSE store[k]
70          ]
71      ∧ writes_history' = Append(writes_history, [key ↦ key, val ↦ val, ts ↦ now])
72      ∧ UNCHANGED ⟨cache, requests, replies⟩
```

```
74  \* The client retrieves the value for a key from the cache. No cache miss.
75  GetOk(key) ≜
76      GetOk::
77      ∧ key ∈ DOMAIN cache
78      ∧ LET entry ≜ cache[key] IN
79          ∧ now < entry.ts + TTL
80          ∧ UNCHANGED ⟨store, cache, requests, replies, writes_history⟩
```

```
91  \* The store replies to the cache with the value for a key.
92  > SendReply(req) ≜…
103 \* The cache receives a reply from the store and updates its cache entry.
104 > RecvReply(req) ≜…
```

```
82  \* The client misses the cache, so a request is sent to the store.
83  \* The cache entry is invalidated.
84  GetMiss(key) ≜
85      GetMiss::
86      ∧ key ∈ DOMAIN cache ⇒ (now > cache[key].ts + TTL)
87      ∧ requests' = requests ∪ { [key ↦ key, ts ↦ now] }
88      ∧ cache' = [ k ∈ DOMAIN cache \ {key} ↦ cache[k]]
89      ∧ UNCHANGED ⟨store, replies, writes_history⟩
```

Checking invariants

```
KEYS ≜ 0..(2^32 - 1)
VALUES ≜ 0..(2^32 - 1)
TIMESTAMPS ≜ 0..(24 * 60 * 60)
DELTA ≜ 300
```

Large state space for state enumeration and simulation!

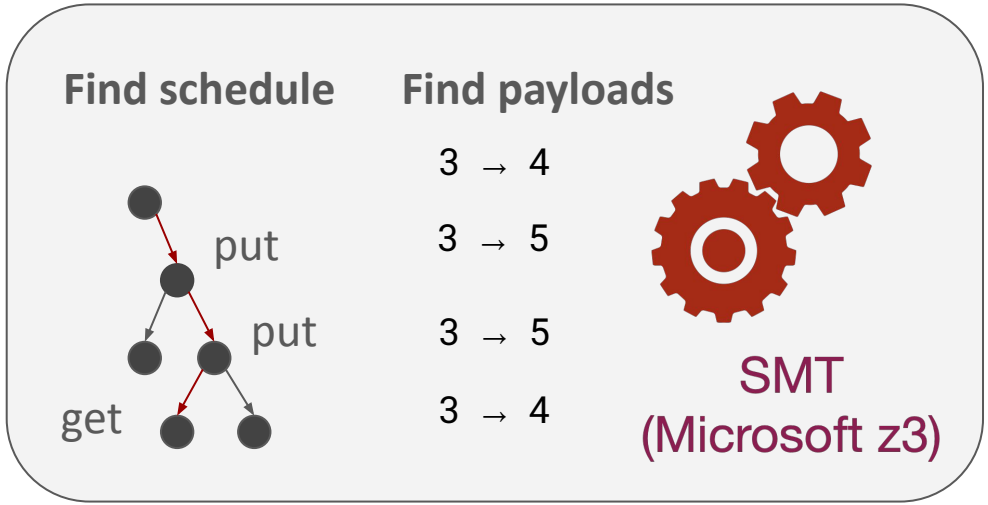
Closer to code configurations

```
126  Inv1 ≜
127      ∀ key ∈ DOMAIN cache:
128          ∧ key ∈ DOMAIN store
129          ∧ LET cache_entry ≜ cache[key]
130                store_entry ≜ store[key] IN
131            (cache_entry.ts ≥ store_entry.ts) ⇒ cache_entry.val = store_entry.val
```

Epoch #1: Bounded model checking (symbolic)

Explore all paths up to *k* steps, e.g., 20

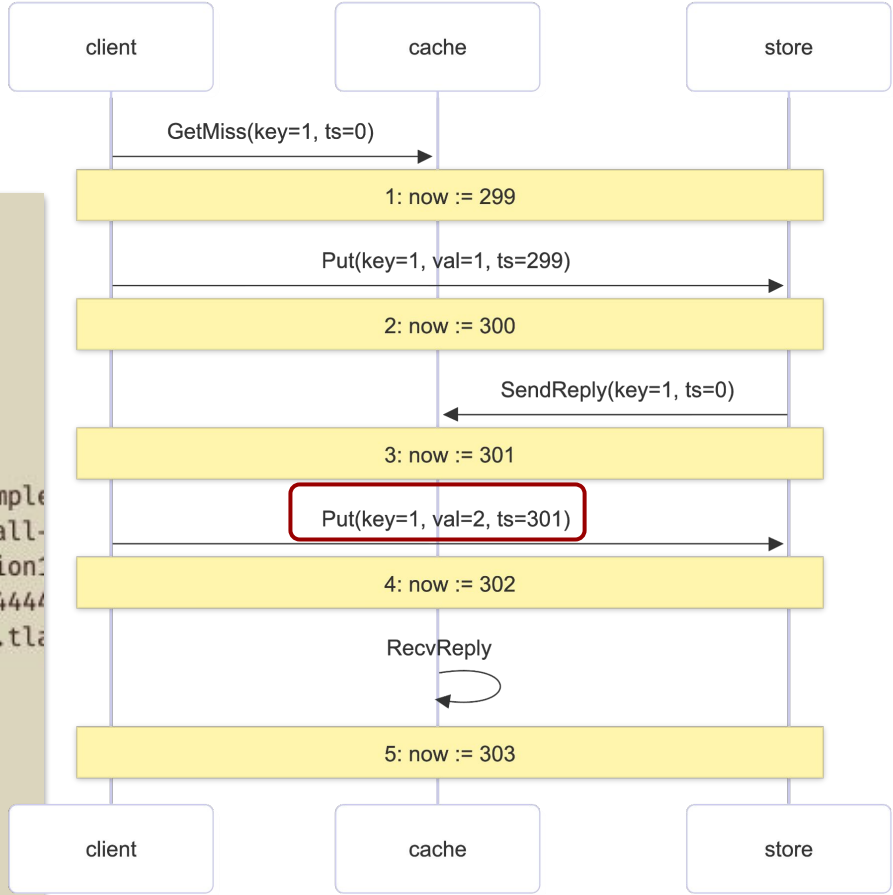
$ apalache-mc check



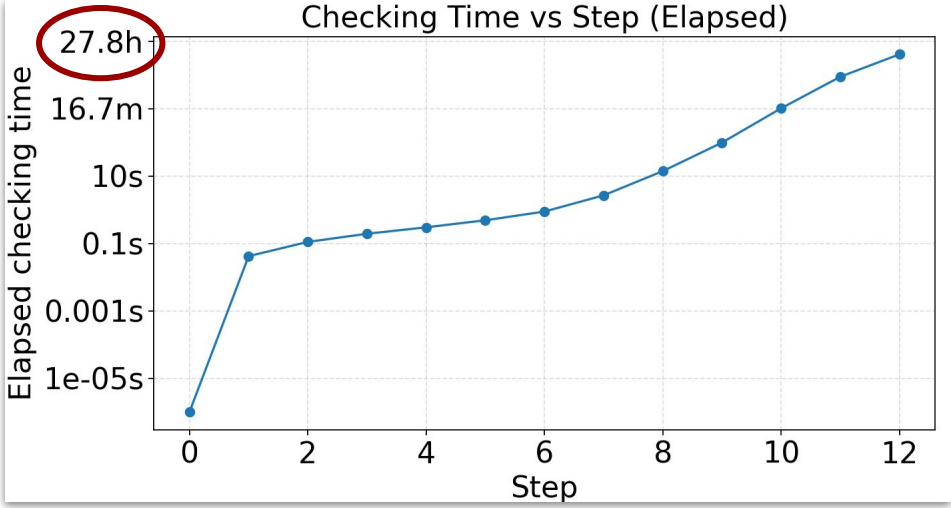
Kind of symbolic
breadth-first search

Inv1 violated

```
State 5: Checking 2 state invariants
State 5: state invariant 0 holds.
State 5: state invariant 1 holds.
State 5: Checking 2 state invariants
State 5: state invariant 0 holds.
State 5: state invariant 1 holds.
State 5: Checking 2 state invariants
State 5: state invariant 0 holds.
Check the trace in: /Users/igor/devl/all-apalache/apalache-example
12-06-45_4444455848424297195/violation1.tla, /Users/igor/devl/all-
he-out/MC.tla/2026-04-08T12-06-45_4444455848424297195/MCviolation1
les/simple-kv-cache/_apalache-out/MC.tla/2026-04-08T12-06-45_4444
ll-apalache/apalache-examples/simple-kv-cache/_apalache-out/MC.tla
1.itf.json I@12:06:47.777
State 5: state invariant 1 violated.
Found 1 error(s)
The outcome is: Error
Checker has found an error
It took me 0 days  0 hours  0 min  2 sec
Total time: 2.258 sec
```



Better invariant



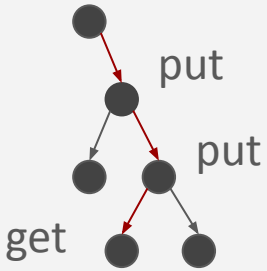

```
167  Inv4 ≜
168      ∀ key ∈ DOMAIN cache:
169          ∧ key ∈ DOMAIN store
170          ∧ LET cache_entry ≜ cache[key] IN
171              ∃ i ∈ DOMAIN writes_history:
172                  LET write ≜ writes_history[i] IN
173                  ∧ write.key = key
174                  ∧ write.val = cache_entry.val
175                  ∧ write.ts ≤ cache_entry.ts
176                  \* if there is a later write to the same key with a different value, then
177                  \* it cannot be older than TTL time units
178                  ∧ ∀ j ∈ DOMAIN writes_history:
179                      LET other_write ≜ writes_history[j] IN
180                      ∨ other_write.key ≠ key
181                      ∨ other_write.val = write.val
182                      ∨ other_write.ts ∉ (write.ts + 1)..(cache_entry.ts - TTL)
```

How deep shall we check it?

Epoch #2: Randomized symbolic execution

Exponential slowdown

Find schedule

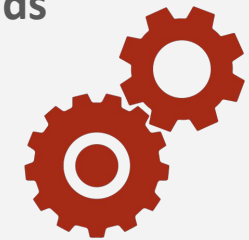


Find payloads

3 → 4

3 → 5

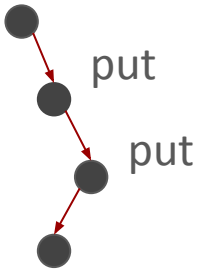
3 → 5



SMT
(Microsoft z3)

Bounded model checking

Random schedule



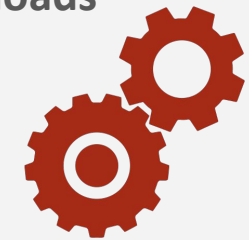
Deeper traces

Find payloads

3 → 4

3 → 5

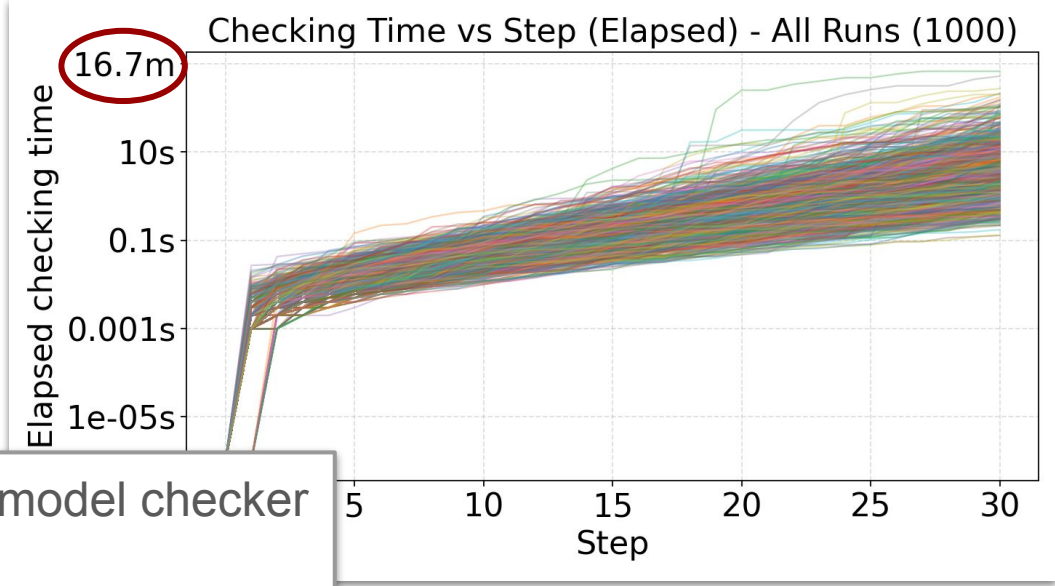
3 → 4



SMT
(Microsoft z3)

Randomized symbolic execution

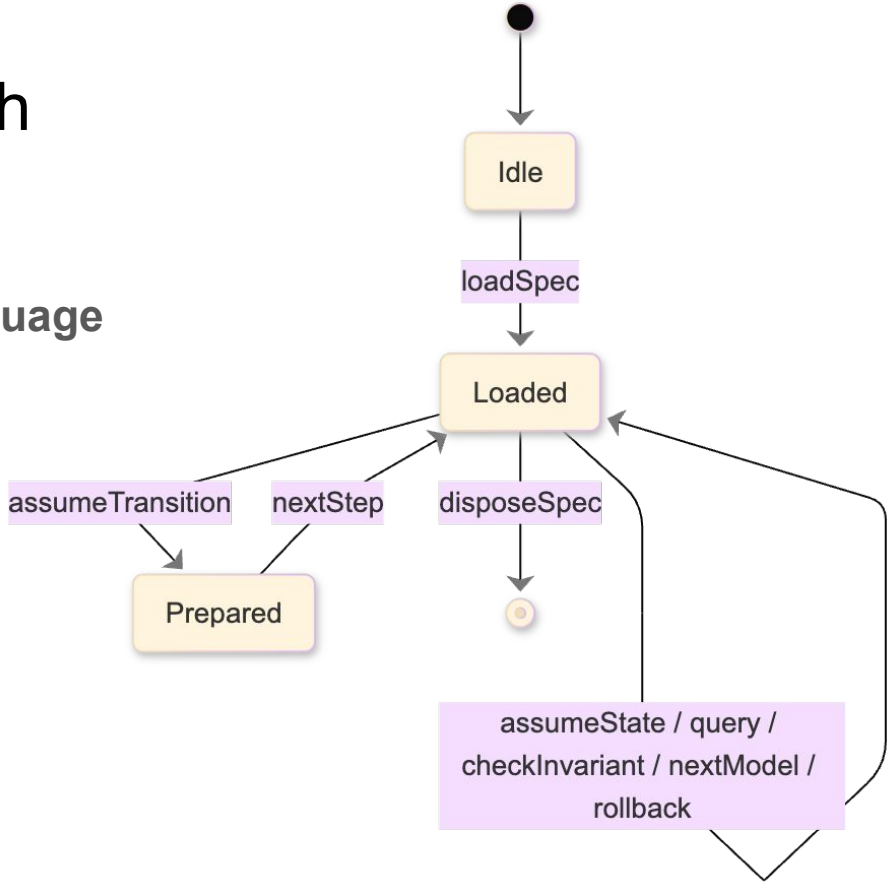
$ apalache-mc simulate



- ✅ Finds bugs faster than bounded model checker
- ✅ Easier for Z3 – mostly propagation
- ✅ Trivially parallelizable
- ❌ Hard to customize

Epoch #3: programmatic search

- ✅ Write your own search scripts in **any language**
- ✅ **Prioritize schedules** as you like
- ✅ **Feedback** to Apalache and Z3
- ✅ **Query** for traces and **enumerate** models



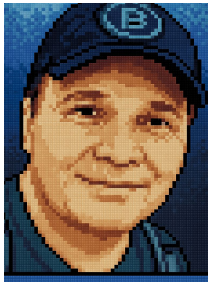
Good fit for AI tools!

Experiment 1: Symbolic Testing of TFTP

Hypothesis: AI tools can generate test harnesses

Setup:

1. The implementation is fixed (tftp-hpa, atftpd, …)
2. **Human** writes the TLA+ spec and refines it
3. **AI tool** writes the test harness and refines it
4. Harness runs the SUT and Apalache



Initial specification effort

Did not survive testing!

1. Relatively straightforward with my experience
2. Most of my time went into switching between RFCs
3. Generating repetitive type definitions with LLMs
4. Playing with falsy invariants to produce examples

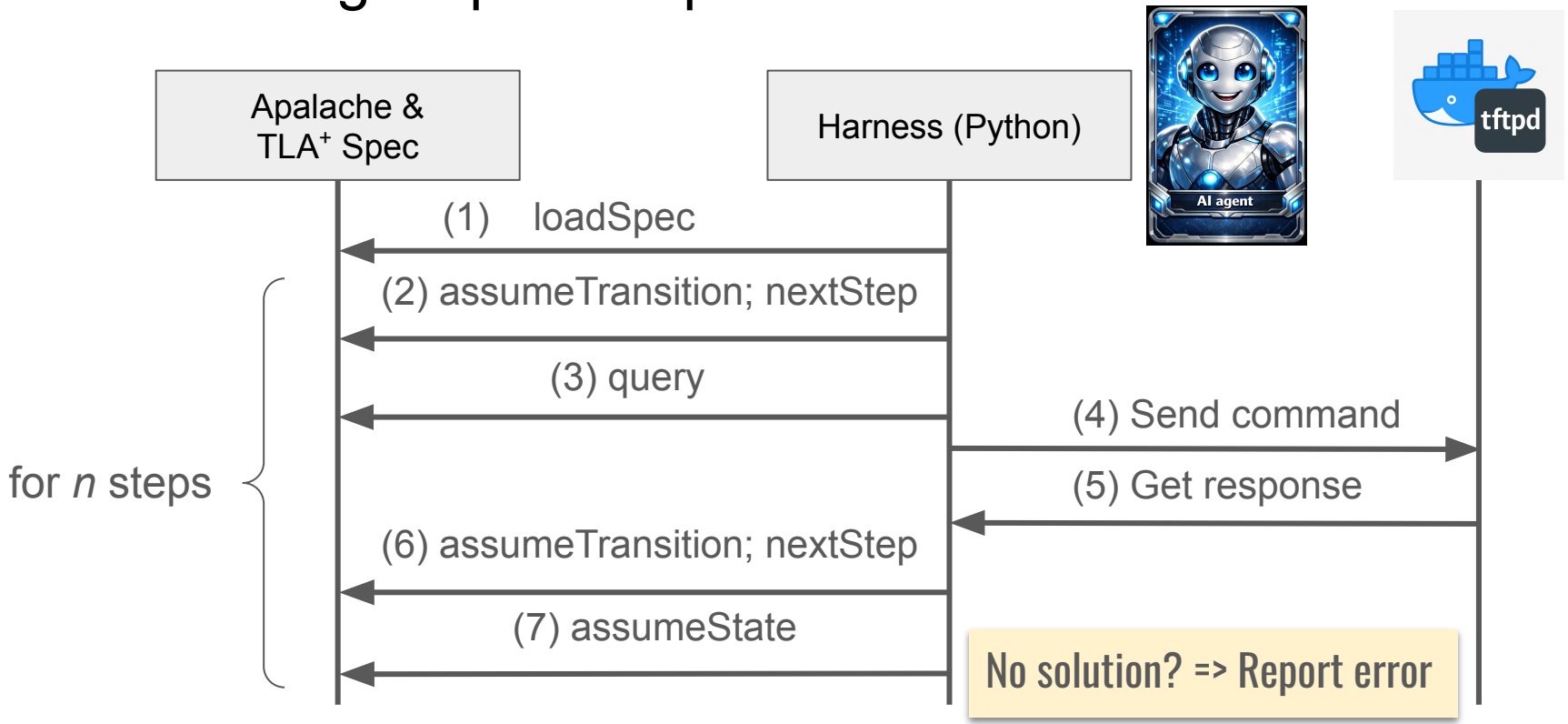
[github.com/konnov/tftp-symbolic-testing]

13 hours ≈ 2 days

Question #1 from every engineer:

How does it connect to the implementation?

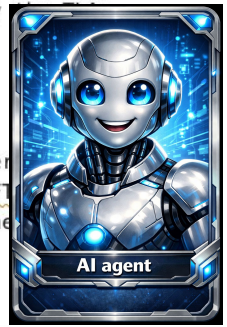
New testing loop with Apalache



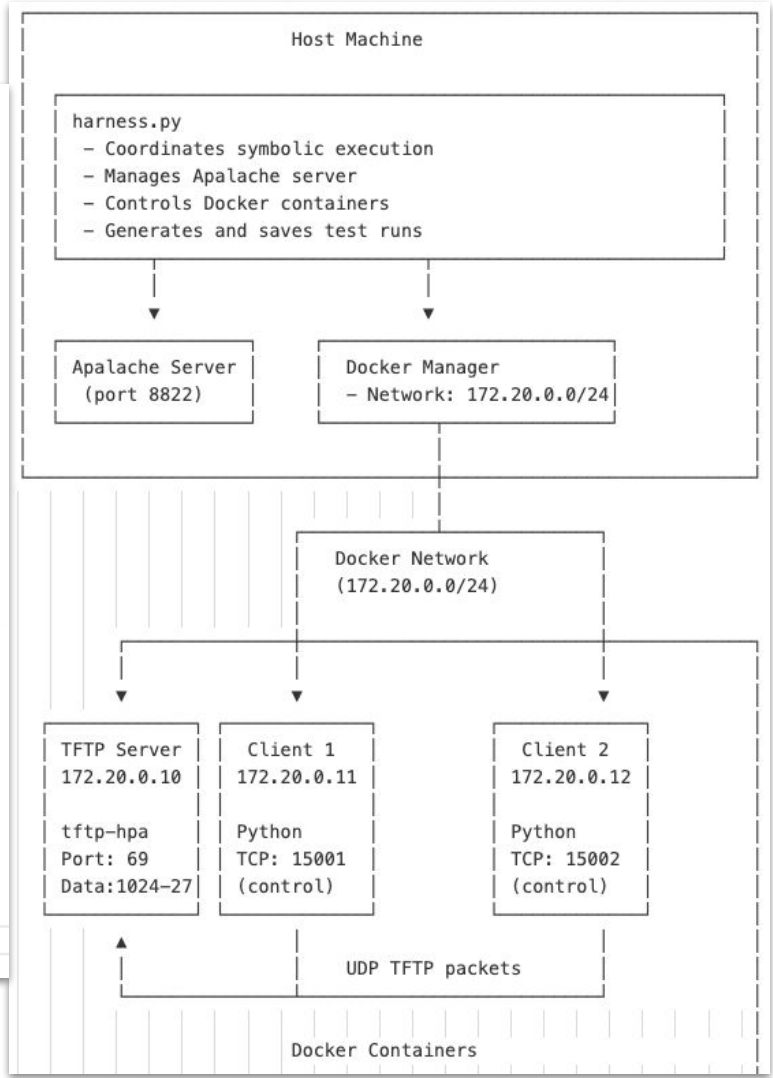
prompt-test-harness.md > ## Requirement 2

```
1  You are a protocol testing engineer. Your goal is to use the TLA+ specification
2  of TFTP to create a test harness that produces new tests by analyzing symbolic
3  executions of the protocol via JSON RPC of Apalache (see [client.py][]).
4
5  Below are the requirements of this project. Do not change this text unless
6  explicitly instructed to do so.
7
8  ## Requirement 0
9
10 The TLA+ specification of the TFTP protocol is provided in the [spec][] file.
11 The test harness should utilize this specification to generate symbolic
12 executio                                                              y
13 specific
14
15 ## Requirement 1
16
17 The test harness must be implemented in Python and should inter
18 using JSON RPC to retrieve symbolic execution traces of the TFT
19 Python script is located at [test-harness](./test-harness). The
20 should use Poetry to pull in dependencies.
21
22 ## Requirement 2
23
24 The communication with Apalache must be done using JSON RPC calls. The API is
25 implemented in the [client.py][] file, which provides functions to send requests
26 to Apalache and receive responses. The harness should utilize these functions to
27 obtain symbolic execution traces for the TFTP protocol.
28
29 ## Requirement 3
```

Detailed prompt – 5 hours of work



Overengineered



Debugging the harness

≈ 1 week
≈ 200 premium requests (Copilot)

1. Lots of debugging
2. Explaining Claude how to handle packets
3. Analyzing logs
4. Fixing bugs in the generated harness code
5. Claude lost in abstractions (e.g., clients in docker vs. clients in spec)

Lesson: design proper architecture upfront!

FIX #1: Error on RRQ is sent from the ephemeral port (not 69)

...

FIX #10: The server should send default timeout if it's not specified in the options

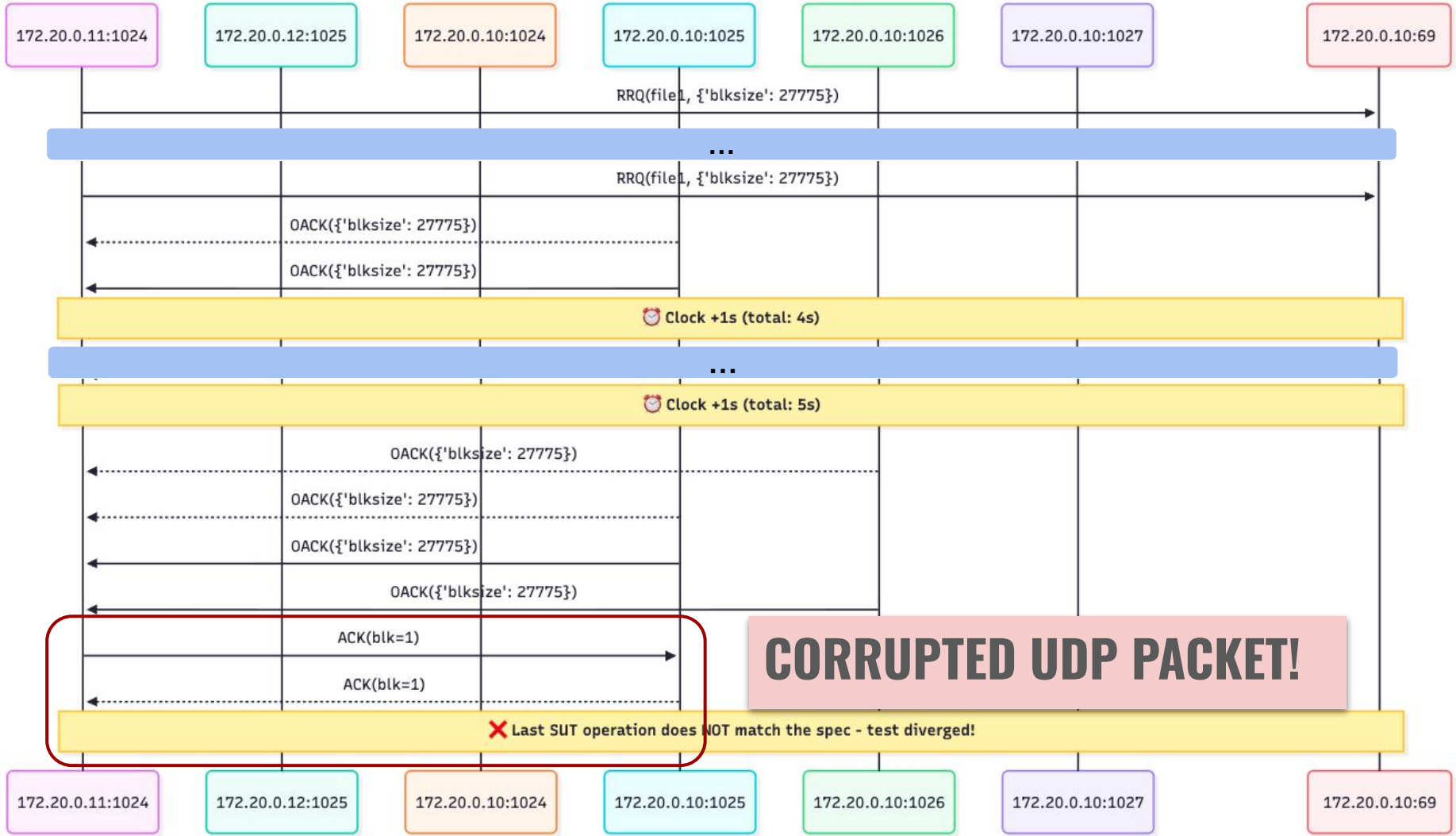
FIX #11: The server may send invalid (e.g., outdated) packets

FIX #12: My understanding of TFTP timeouts was wrong

FIX #13: Handle ERROR packets

FIX #14: Receive only one OACK message

FIX #15: Use clientIP-clientPort-serverPort triplets


172.20.0.11:1024
172.20.0.12:1025
172.20.0.10:1024
172.20.0.10:1025
172.20.0.10:1026
172.20.0.10:1027
172.20.0.10:69
RRQ(file1, {'blksize': 27775})
...
RRQ(file1, {'blksize': 27775})
OACK({'blksize': 27775})
OACK({'blksize': 27775})
⏰ Clock +1s (total: 4s)
...
⏰ Clock +1s (total: 5s)
OACK({'blksize': 27775})
OACK({'blksize': 27775})
OACK({'blksize': 27775})
OACK({'blksize': 27775})
ACK(blk=1)
ACK(blk=1)
❌ Last SUT operation does NOT match the spec - test diverged!
CORRUPTED UDP PACKET!
172.20.0.11:1024
172.20.0.12:1025
172.20.0.10:1024
172.20.0.10:1025
172.20.0.10:1026
172.20.0.10:1027
172.20.0.10:69

Lessons from LLM-generated harnesses

October 2025

Copilot + Sonnet 4.5

1. Surprisingly, **it works** – **not effortlessly** though
2. **Do not trust** the generated harness – TODOs and bugs inside
3. **Do not let** an LLM define **its own formats** – it goes wild
4. Log yourself and **define your log format** – or, face wild regexes
5. **Generate visualizations** that suit your needs – it's **amazing**!

Effort: 2 weeks, 2 impl. bugs

Better, use structured formats – JSON

It’s harder to write a more precise spec

The verification engineer’s mindset:

- The specification usually **overapproximates** the implementation
- **Reachability**: if the impl. reaches a state *s*, then the spec. reaches α(s)

Conformance testing (roughly):

- If the spec. executes action *A*, then impl. must be able to execute *A*
- If the impl. executes action *A*, then spec. must be able to execute *A*

Quality jump Q4 2025 – Q1 2026

Claude Code Opus + Sonnet 4.6

Codex GPT 5.4

```
→  zookeeper-testing git:(main) ✗ claude
   Claude Code v2.1.81
   Sonnet 4.6 · Claude Pro
   ~/devl/zookeeper-testing

❯ Extract TLA+ specifications from the source code of ZooKeeper, write a test harness, run it
```

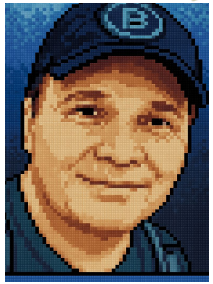
APACHE ZooKeeper™

Experiment 2: Symbolic Testing of ZooKeeper

Hypothesis: AI tools can generate the spec **and** the harness

Setup:

1. The implementation is fixed (ZooKeeper)
2. **AI tool** writes the TLA+ spec and refines it
3. **AI tool** writes the test harness and refines it
4. Harness runs the SUT and Apalache



“Almost done”

More autonomy. Sometimes, LLMs work for 1-2 hours

Better efficiency. 5 protocols + harness after 1 week

No magic. Human review & guidance needed

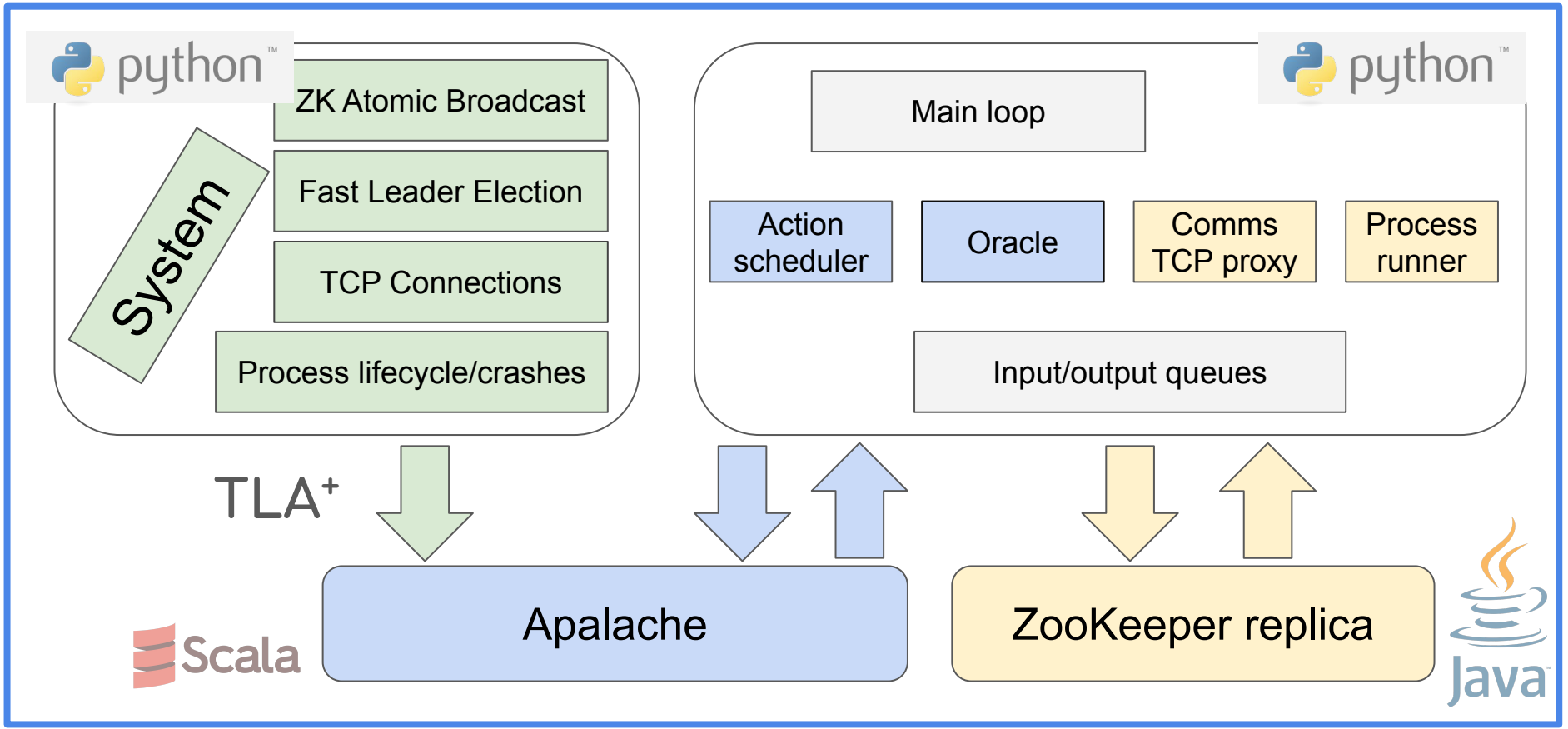
Be careful. Sometimes, AI tools cut corners badly

Effort: ~1 month, no impl. bugs so far

March 2026

Codex GPT 5.4

Claude Sonnet/Opus 4.6


python
ZK Atomic Broadcast
System
Fast Leader Election
TCP Connections
Process lifecycle/crashes
python
Main loop
Action scheduler
Oracle
Comms TCP proxy
Process runner
Input/output queues
TLA+
Scala
Apalache
ZooKeeper replica
Java

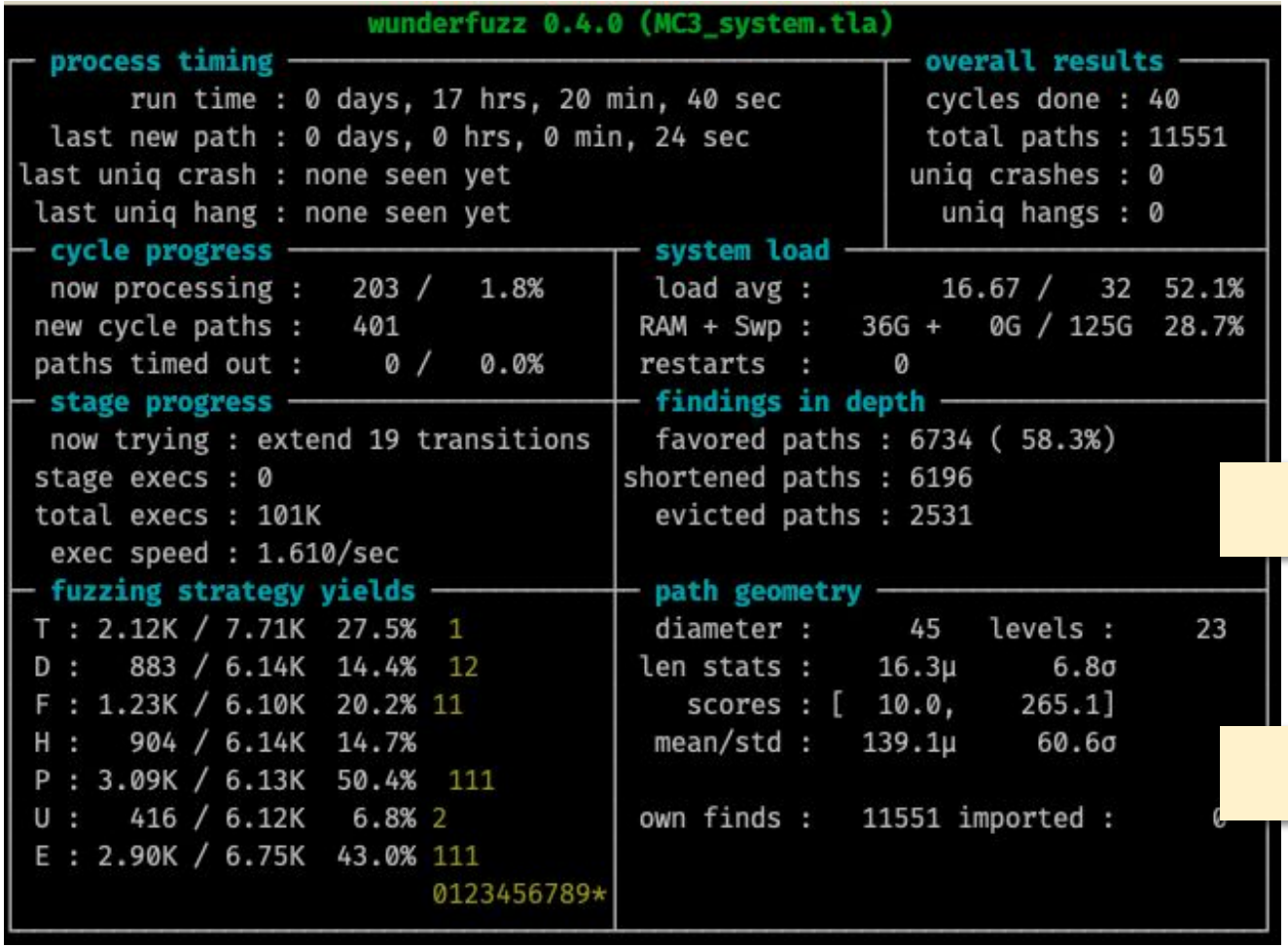
Biasing actions

action	total	views	avg	max
fle_rcv_notification	43.86	17	2.58	8.00
fle_send_notification	34.82	17	2.05	6.00
fle_h				
fle_h				
fle_r				

action	total	views	avg	max
zab_client_connect	8.65	27	0.32	1.00
zab_client_get_children	8.44	27	0.31	1.00
zab_client_dele				
zab_client_exis				
zab_client_crea				
zab_client_get_data	8.44	27	0.31	1.00
zab_client_set_data	8.44	27	0.31	1.00

```
View ≜
    WSMkTuple4(proc_status, fle_role, zab_state, zab_sync)
```

action	total	views	avg	max
zab_leader_info	9.17	29	0.32	1.00
zab_diff	9.17	27	0.34	1.00
zab_uptodate	9.17	27	0.34	1.00
zab_newleader	9.17	27	0.34	1.00
zab_follower_info	8.65	27	0.32	1.00
zab_ack_epoch	8.65	27	0.32	1.00
zab_ack_newleader	8.65	27	0.32	1.00
zab_proposal	8.44	27	0.31	1.00
zab_forwarded_request	8.44	27	0.31	1.00
				0
				0
				0
zab_snap	8.44	27	0.31	1.00
zab_trunc	8.44	27	0.31	1.00



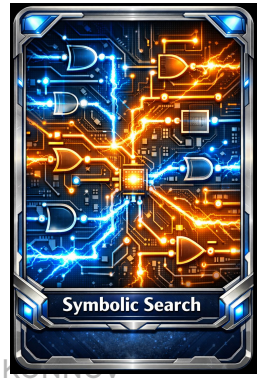
Symbolic fuzzer for TLA+

Contact me to learn more

Takeaways

APALACHE

1. Write your **own symbolic search** scripts!
2. **Connect your spec with the code**, as you like it
3. **LLMs can help** you with automation
4. **Feedback loop** for AI tools
5. You know better **what matters to your project**
6. Easy to **parallelize**



igor@konnov.phd protocols-made-fun.com

Independent security and formal methods researcher 🇦🇹 Vienna, Austria — 2024-...

- Verifying consensus protocols and smart contracts
- Improving Apalache, working on new tools

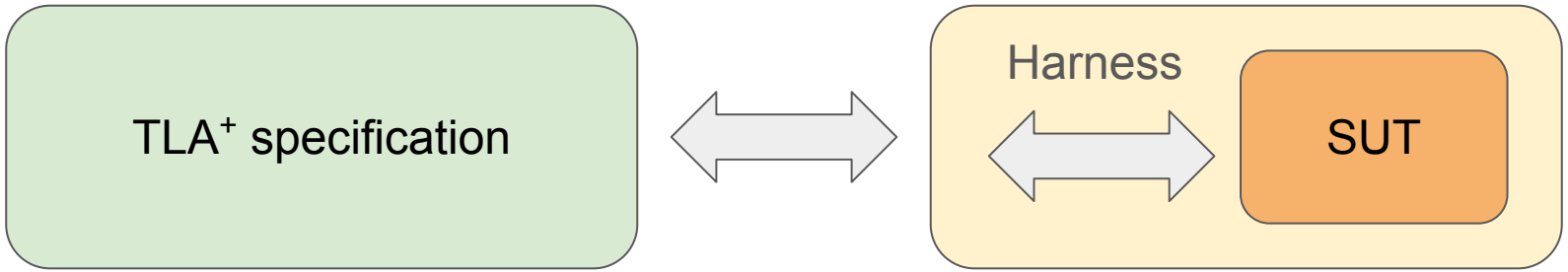
Principal research scientist at Informal Systems (Web3 – blockchains) 🇦🇹 — 2019-2023

- Leading development of Apalache and Quint

🇦🇹 🇫🇷

Postdoc, assistant professor & permanent researcher at TU Wien & Inria — 2011-2019

More slides!



Black-box testing: communication over the network, no instrumentation

Minimal mapping: define the labels of TESTER and SUT

Modular: test components in isolation, the rest is the spec and the checker

Area	Lines
Python spec sources	4692
Generated TLA+ spec	2527
Runtime harness	9482
Test suite	6127
Helper scripts	1026
Harness + tests + scripts	16635

Python spec breakdown

File	Lines
spec/zab.py	1628
spec/fle.py	1481
spec/system.py	1224
spec/tcp.py	204
spec/process.py	90
spec/config.py	65
Total	4692

Looking for witnesses

Example Category	Count
Commit progress	4
Recovery reached	4
Recovery mode exercised	3
Forwarding path exercised	3
Proposal pipeline state	5
Client-visible tree state	3
Total	22

Commit history 1

Foundation March 13-17, 2026

Initial simulation, system boundary, harness, leader election

Protocol depth March 18-April 1, 2026

ZAB, recovery modes, forwarding, witness examples

Stabilization April 6-10, 2026

Scheduler tuning, oracle fixes, invariants, persistent sessions

Commit history 2

239 commits across **28 active days**

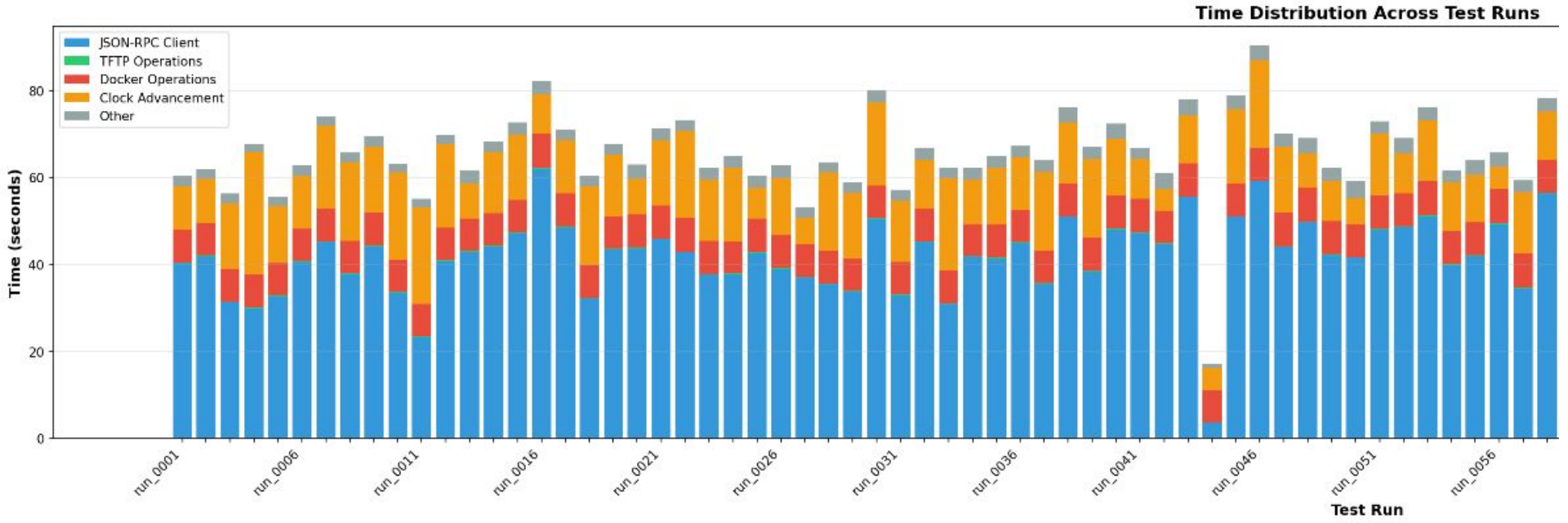
Average pace: 8.5 commits/day

Peak activity: 25 commits on March 30, 2026

Code churn: 28.7k insertions, 13.1k deletions

Produce 100 episodes, 100 steps each

```
$ ./harness.py --docker --steps=100 --tests=100
```



Testing with TLA+

1. Nagendra et. al. ***Model guided fuzzing of distributed systems*** (2025)
2. Cirstea, Kuppe, Merz, Loillier. *Validating Traces of Distributed Systems Against TLA+ Specifications* (2024)
3. Chamayou et. al. *Validating System Executions with the TLA+ Tools* (2024)
4. Jordan Halterman. *Verifiability Gap: Why We Need More From Our Specs and How We Can Get It* (2020)
5. Jessie Davis et al. *eXtreme Modelling in Practice* (2020)
6. Kupriyanov, Konnov. *Model-based testing with TLA+ and Apalache* (2020)
7. Pressler. *Verifying Software Traces Against a Formal Specification with TLA+ and TLC* (2018)

Except [6], all use TLC