ORACLE

Reverse-Engineering with TLA$^+$

Calvin Loncaric
Verification Engineer

Background: TLA$^+$ at Oracle Cloud Infrastructure (OCI)

2015

First TLA$^+$ written at OCI

2017

“Verification Team” forms

2024

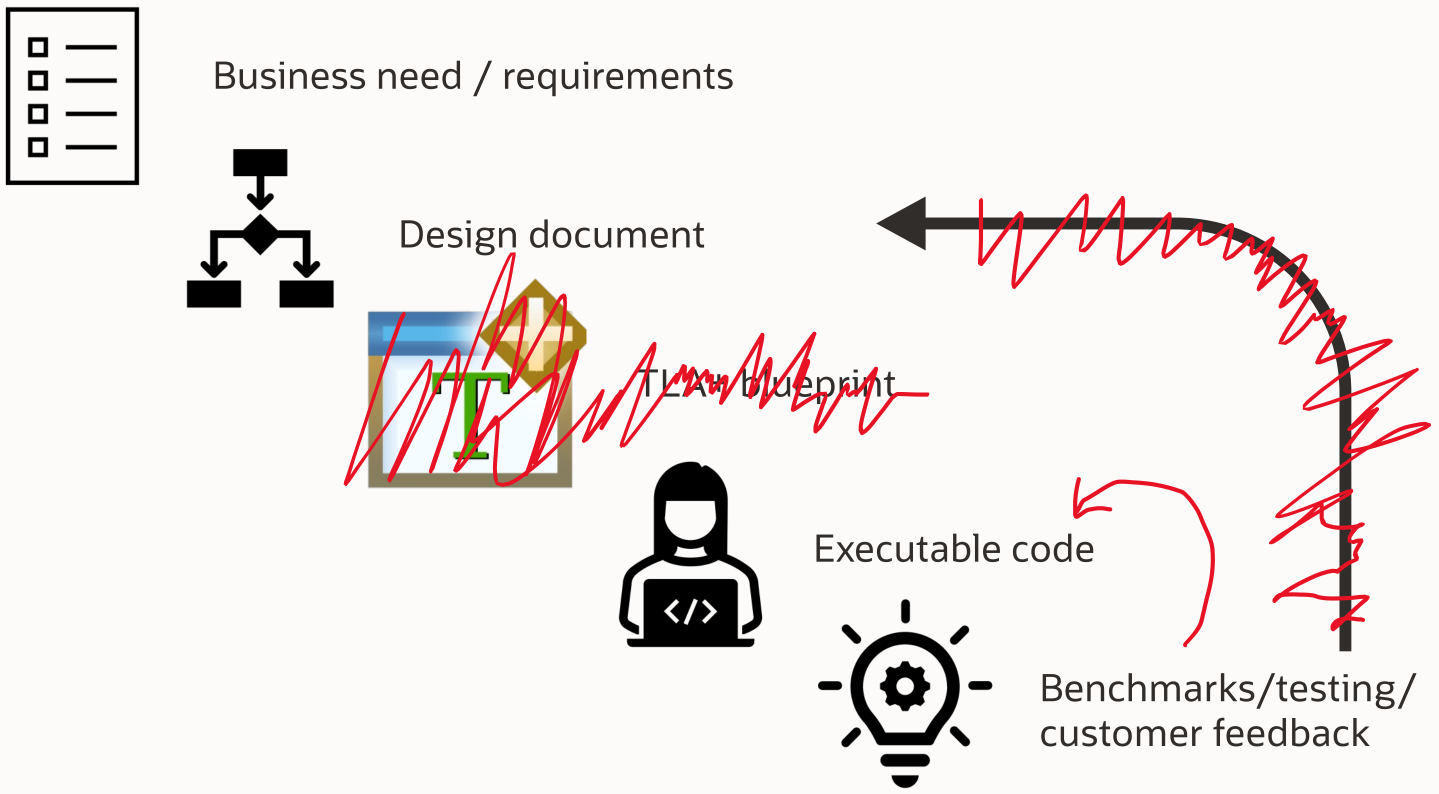
Many critical bugs discovered/fixed

Like an internal consultancy:
- Design reviews on steroids
- Careful analysis of service code
- Talks/workshops/education
- Pushing adoption of formal methods

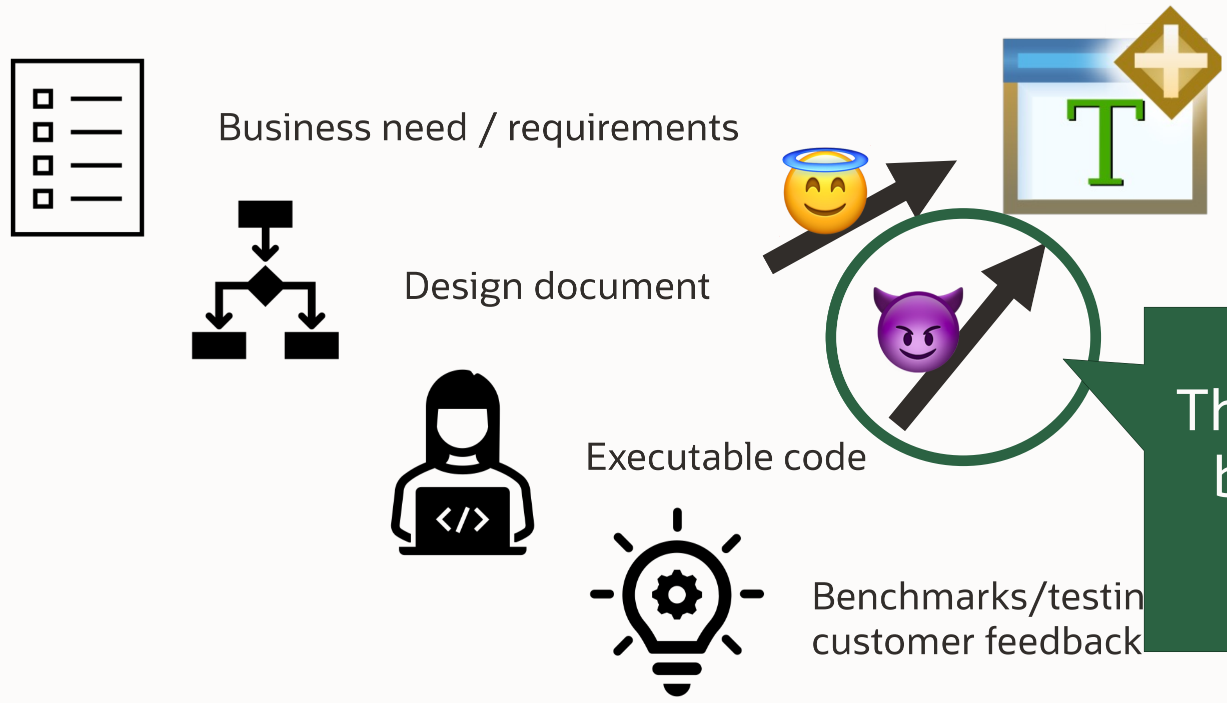
>6 years of TLA$^+$ in practice!

- Dozens of different services and teams
- Hundreds of specifications written
- Hundreds more subtle bugs discovered/fixed

How Software is Written



What do you do if the formalization step got skipped?



This workflow is totally backwards, ***but still effective!***

Why is this an effective way to improve software?

1. Quickly find incorrect assumptions

Order

Atomicity

External services

Environment

Correlated failures

3. Fills gaps not covered by testing

Power outages

Drive failures

Obscure interleavings

2. By construction, specs resemble source code

Easier to communicate findings

Easier to find practical fixes

Easier to update later

4. Yes, we still have to define correctness

The Scale of the Problem

$>10^6$ LoC

(For a single service!)

* We have a few secret weapons that can help!

Secret Weapon: we know what we’re looking for

```
if (condition) {
    log.info(“begin flush”);
    start = currentTime();
    writer.flush();
    duration = currentTime() - start;
    metrics.emitFlushDuration(duration);
}
```

Secret Weapon: we don’t have to model order

(*at least initially)

```
Flush ==
    /\ diskState’ = memState
    /\ UNCHANGED <<…>>
```

Implementation will make decisions about *when*; specification only has to capture *what*

Secret Weapon: we have access to the authors

”Hey @Developer,

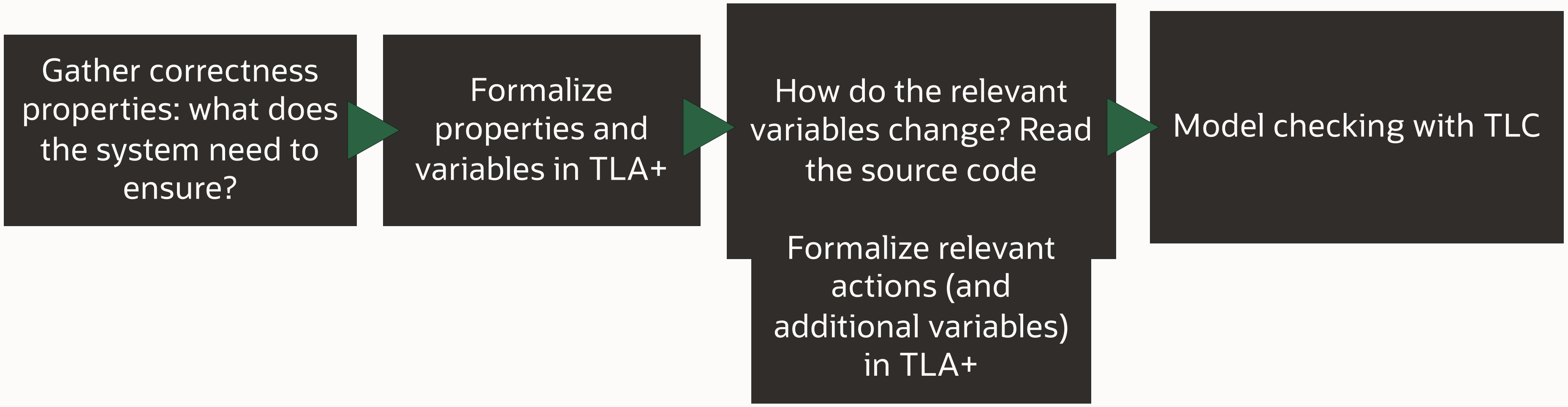
Can you walk me through what happens if the flush fails?”

```
FlushFails(pid) ==
    /\ diskState’ = Havoc
    /\ pc’ = [pc EXCEPT ![pid] = “recover”]
    /\ UNCHANGED <<…>>
```

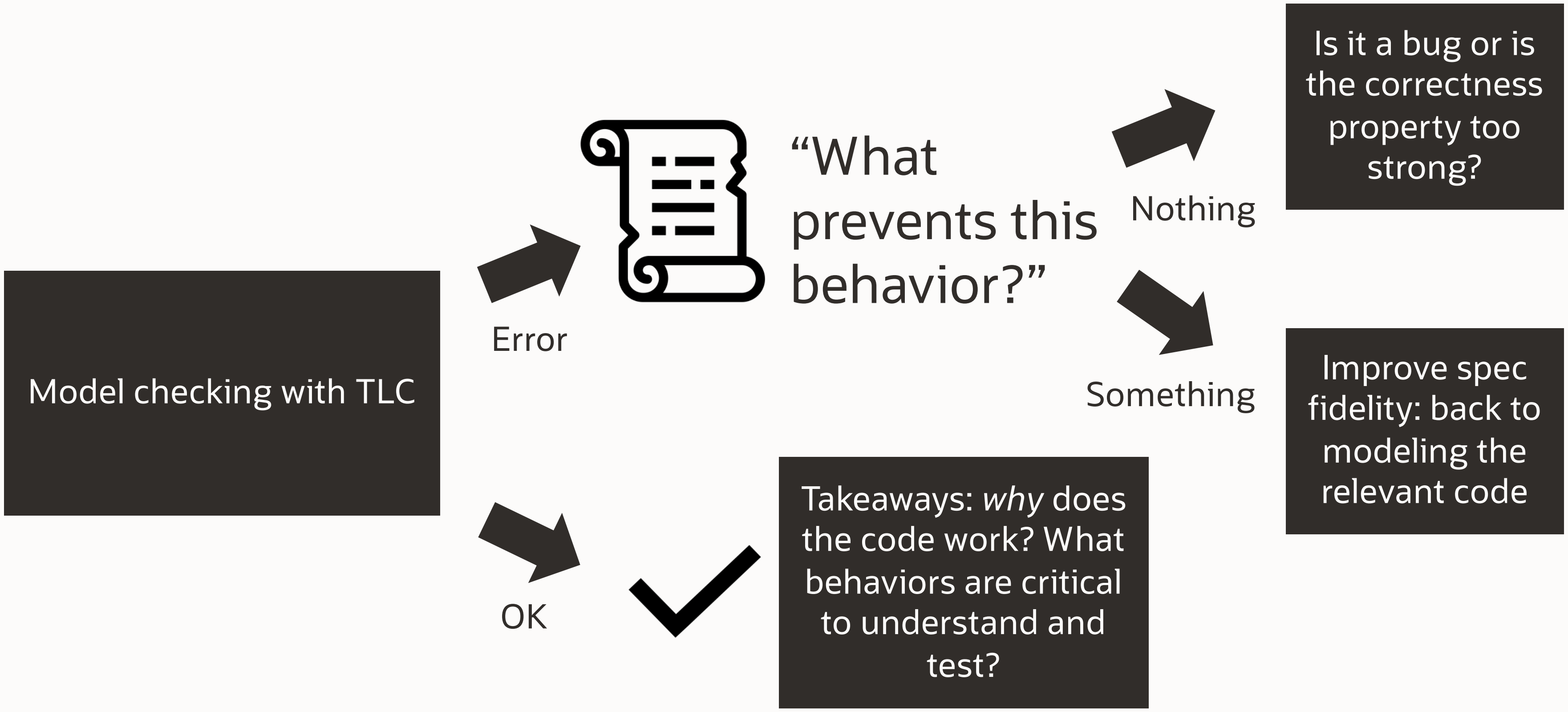
Environment

Correlated failures

The Basic Workflow


Gather correctness properties: what does the system need to ensure?
Formalize properties and variables in TLA+
How do the relevant variables change? Read the source code
Formalize relevant actions (and additional variables) in TLA+
Model checking with TLC

The Key Feedback Loop



Model checking with TLC

“What prevents this behavior?”

Is it a bug or is the correctness property too strong?

Improve spec fidelity: back to modeling the relevant code

Takeaways: *why* does the code work? What behaviors are critical to understand and test?

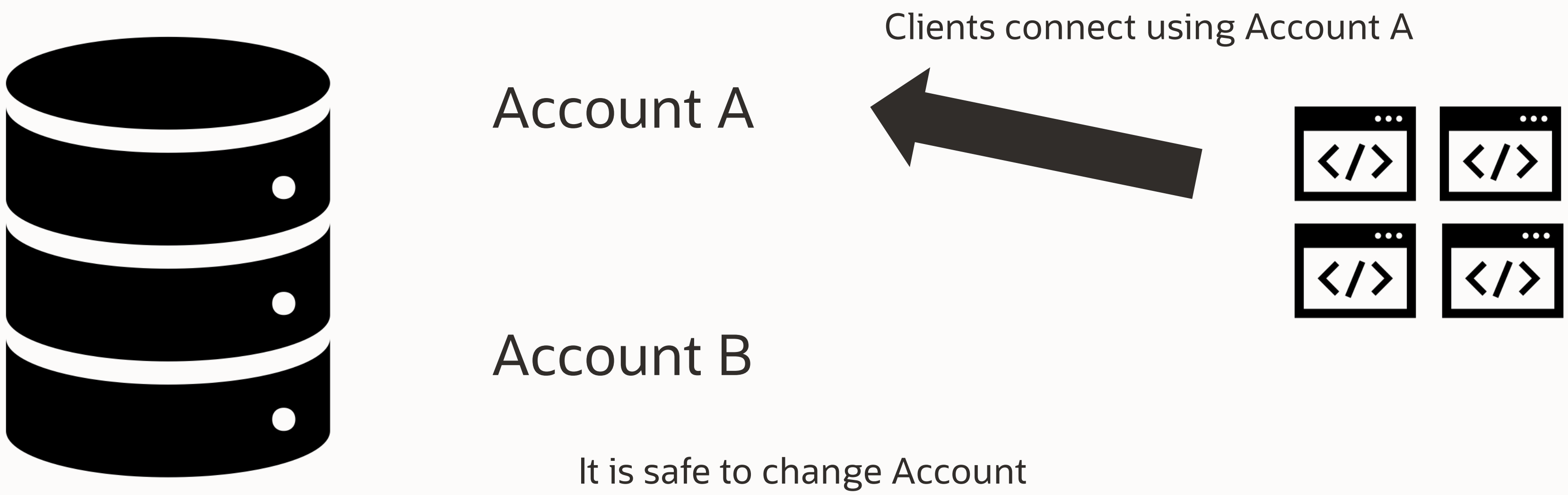
Recent Example: Automatic Password Rotation

August 2023:
- Initial design complete
- Short spec showing safety of a few core actions in steady state

Divergence large enough to justify reverse engineering

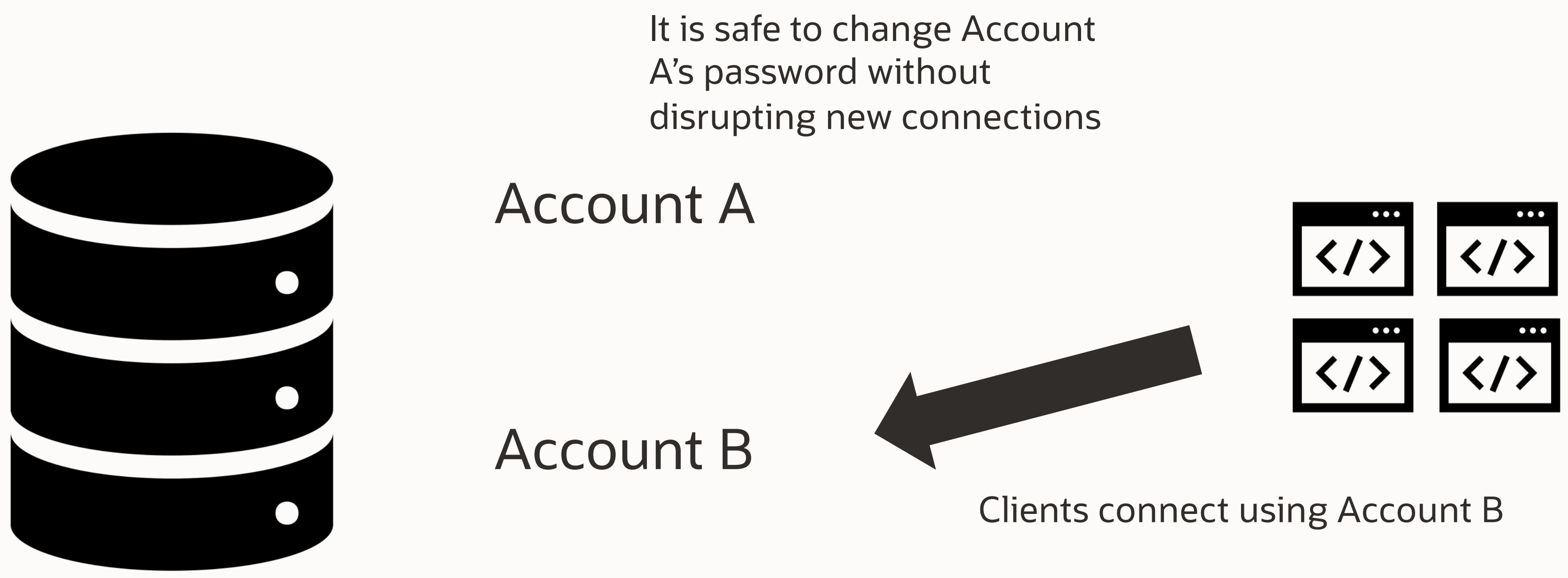
January 2024:
- Code complete
- Different from initial design!
    - New requirements (e.g. repair so-called “special-case” systems)
    - New features (e.g. in-memory cache for certain bits of remote state)

Ultra-High-Level Intuition



It is safe to change Account B’s password without disrupting new connections

Ultra-High-Level Intuition


It is safe to change Account A's password without disrupting new connections
Account A
Account B
Clients connect using Account B

A Roadmap to the Code

~300k LoC split across 4 repositories

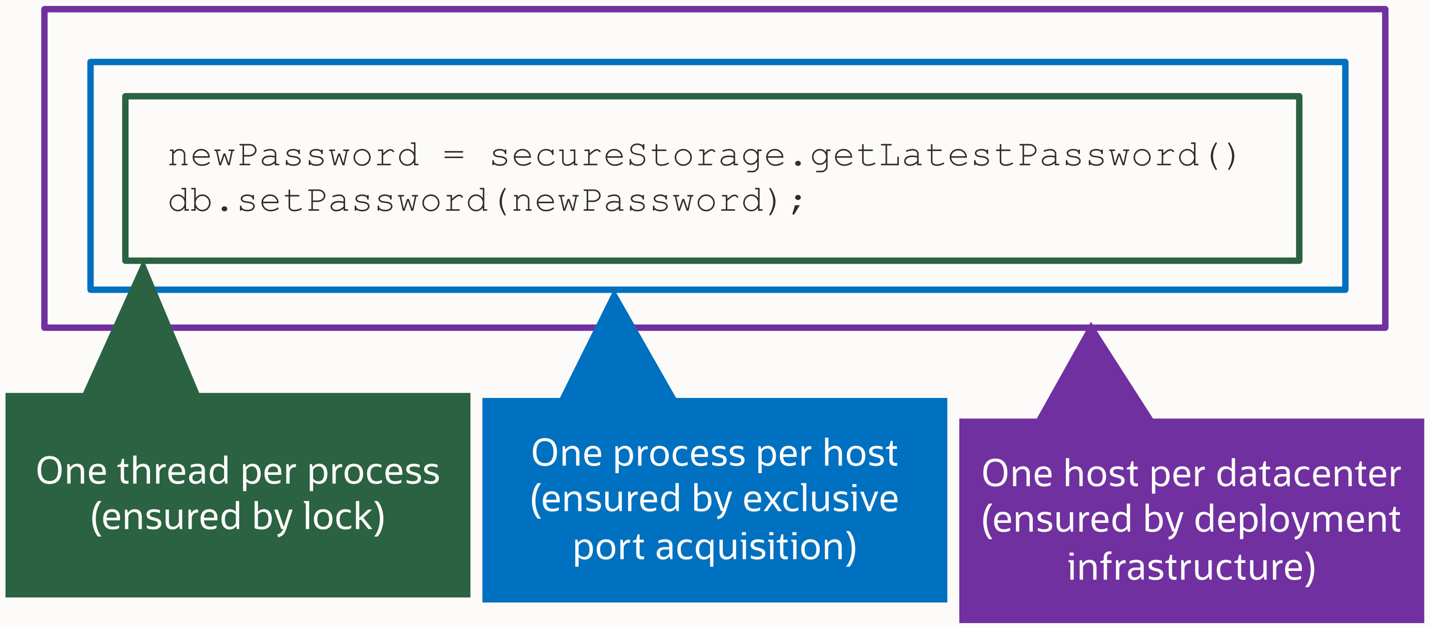
- Common utility library
- DB abstraction layer library
- "Control Plane" service
  - Password rotation algorithm lives here
- "Data Plane" service
  - Needs to respond to password changes

Only a tiny subset is relevant to password rotation!

Next: a few observations about the password rotation design

(These are common things *you* can look for if you ever find yourself reverse-engineering some source code!)

Common Pattern 1/3: Single-Threaded != Nonconcurrent



```
newPassword = secureStorage.getLatestPassword()
db.setPassword(newPassword);
```

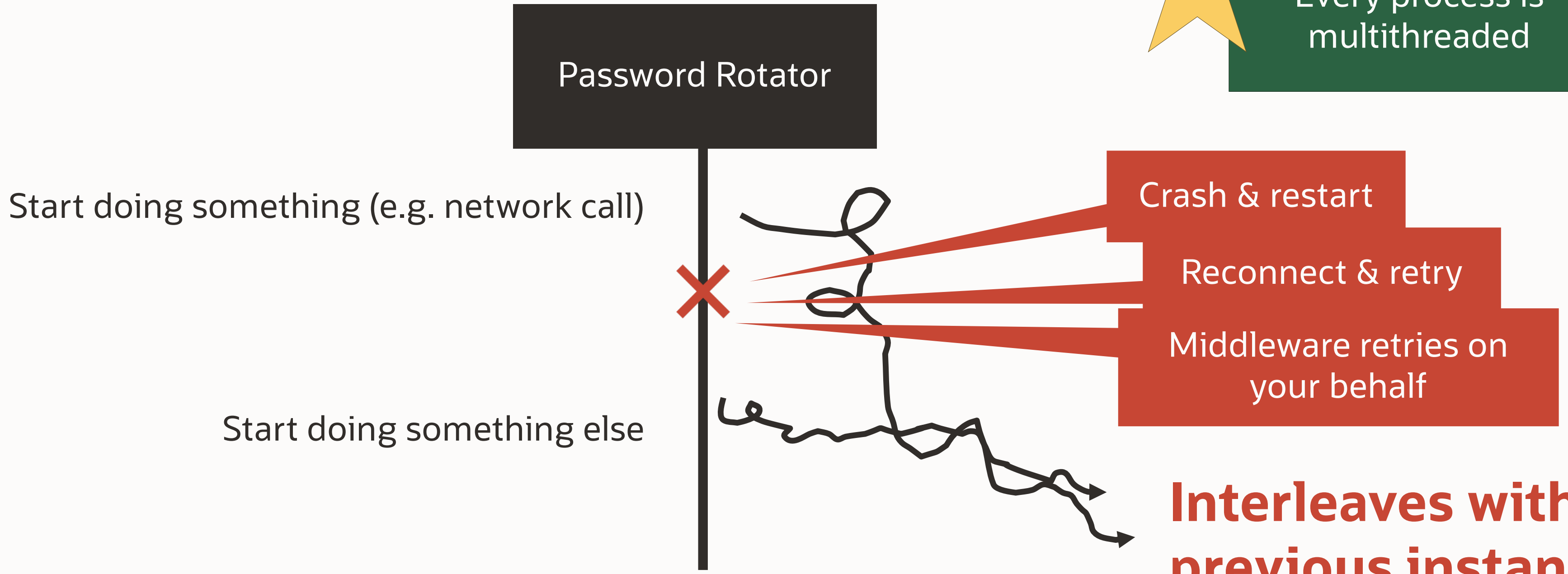
One thread per process (ensured by lock)
One process per host (ensured by exclusive port acquisition)
One host per datacenter (ensured by deployment infrastructure)

Tempting to treat this as a single-threaded process

--- but even with all these protections, concurrency is still possible!

Common Pattern 1/3: Single-Threaded != Nonconcurrent

Every process is multithreaded


Password Rotator
Start doing something (e.g. network call)
Start doing something else
Crash & restart
Reconnect & retry
Middleware retries on your behalf

Interleaves with a previous instance of itself!

Common Pattern 2/3: Unconditional Writes

```
newPassword = secureStorage.getLatestPassword()
db.setPassword(newPassword);
```

↓

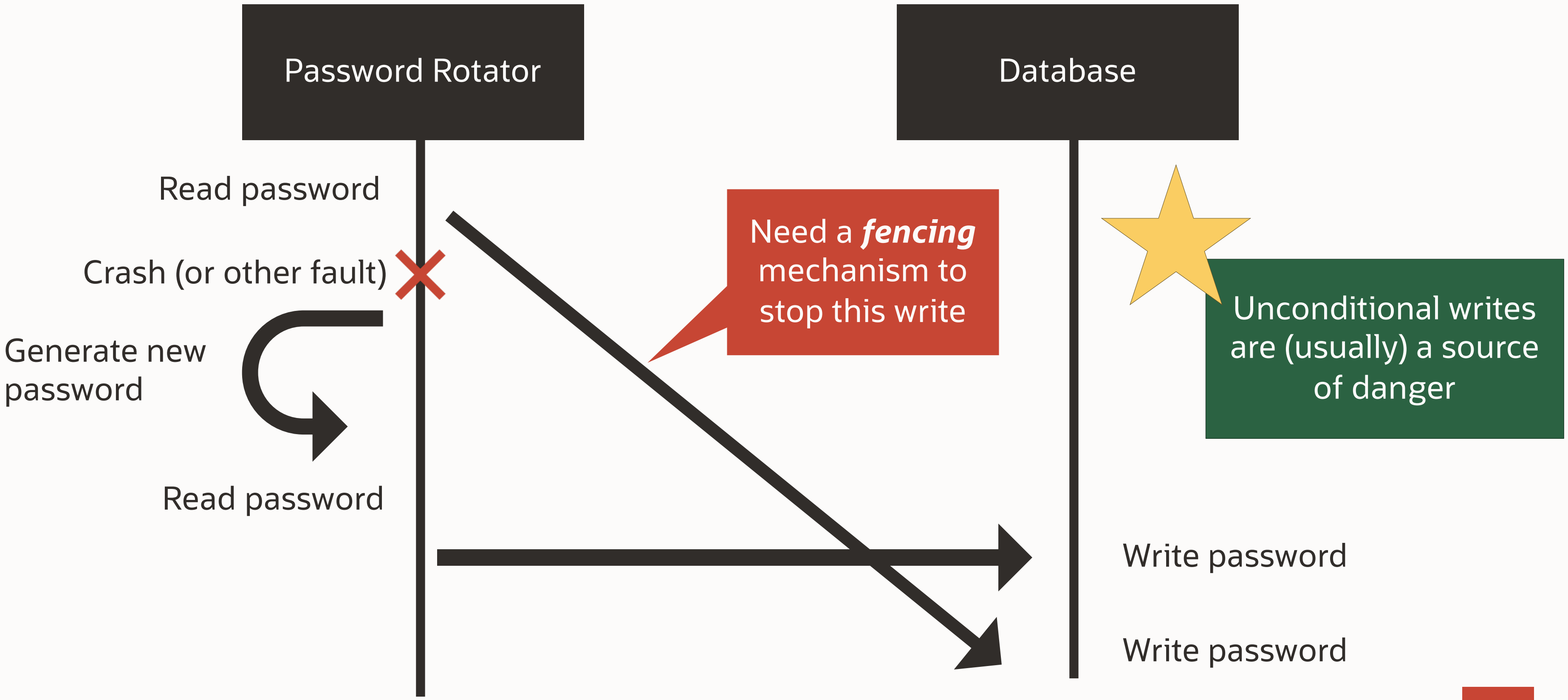
```
ReadPassword(pid) ==
       /\ observed_password' = …
       /\ …

WritePassword(pid) ==
       /\ db_password' = observed_password[pid]
       /\ …
```

Unconditional write is like a **bullet in the flight**, ready to overwrite your password at a later date

There is no way to “fence out” this action!

Common Pattern 2/3: Unconditional Writes



Common Pattern 2/3: Unconditional Writes

This is not a conditional write! The check and the network call are not atomic!

```
if (check) {
    db.setPassword(newPassword);
}
```

Common Pattern 3/3: Reliance on Timestamps

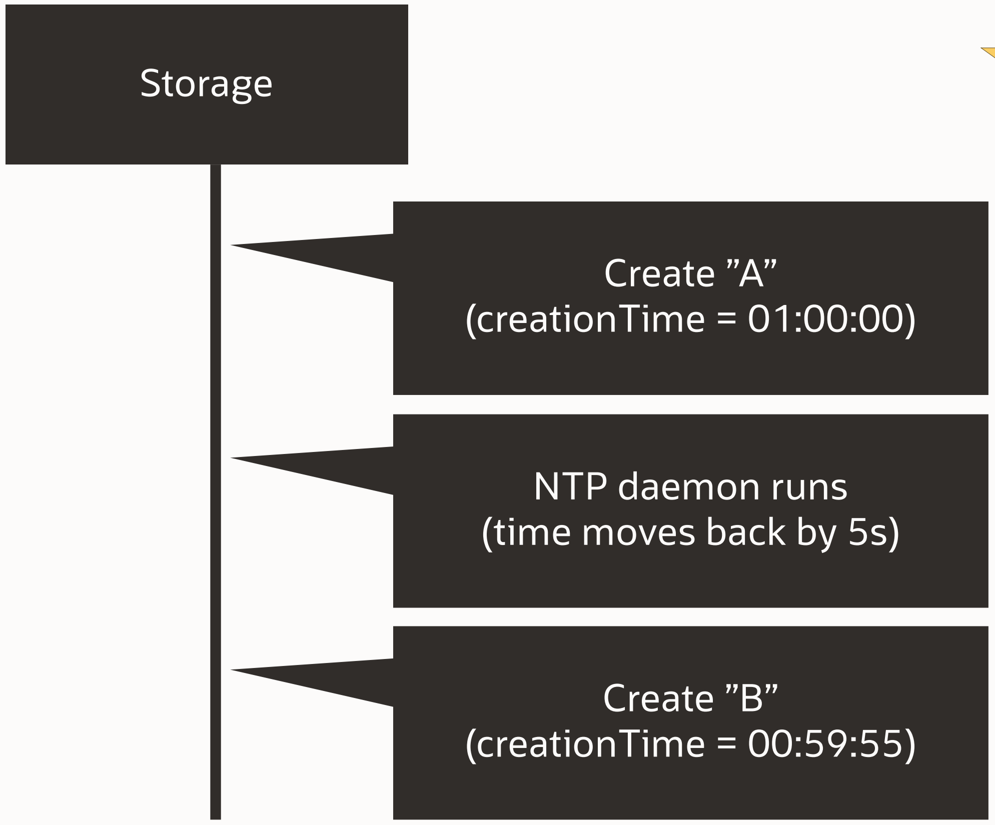
```
a = secureStorage.get("a")
b = secureStorage.get("b")

if (a.creationTime < b.creationTime) {
    …
}
```

This check is essentially a nondeterministic choice

Misconfigurations (rare, but possible!) can cause these to be off by seconds or *decades*

Common Pattern 3/3: Reliance on Timestamps


Storage
Create ”A” (creationTime = 01:00:00)
NTP daemon runs (time moves back by 5s)
Create ”B” (creationTime = 00:59:55)

Often there is no need to model real time; it (usually) won’t be part of the safety mechanism

Password Rotation: Findings and Outcomes

- ~1 week of reverse-engineering
  (spread across ~1 month)
- Timing assumptions revealed
- 1 new bug uncovered
- Safety property revised:
  ~~[]Safe~~
  <>[]Safe

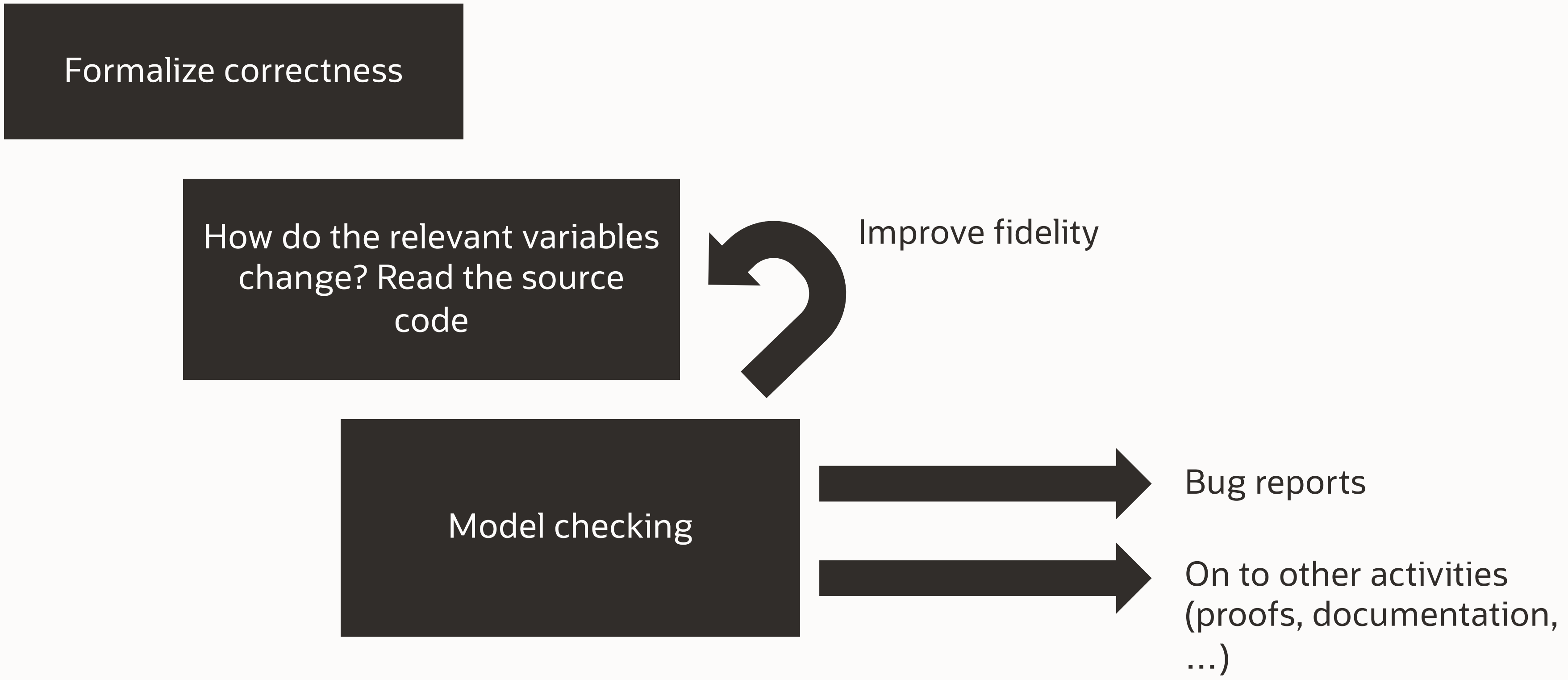
An unfortunate necessity: some underlying systems do not support proper conditional writes

Easy to understand: relates to a a specific check in the source code

Still a strong result!

Reverse-Engineering with TLA$^+$

Calvin Loncaric
<calvin.loncaric@oracle.com>


Formalize correctness
How do the relevant variables change? Read the source code
Improve fidelity
Model checking
Bug reports
On to other activities (proofs, documentation, …)