Mind Your Metrics: How (Not) to Evaluate a Fuzzer

Kyle Dewey, Mehmet Emre, Ben Hardekopf

Teaser

- We argue that fuzzers are best evaluated by a quantitative comparison of unique bugs found
- We define an automated technique to accurately identify unique bugs found, dramatically simplifying this comparison
- We show that commonly used alternative metrics often disagree with the metric of unique bugs found, making them ultimately useless
- 24 new bugs found in SMT solvers (including Z3); **correctness** bugs found in each solver tested

Outline

- Background
- Metrics used in the literature
- An automated approach
- How metrics compare
- Conclusion

Outline

- Background
- Metrics used in the literature
- An automated approach
- How metrics compare
- Conclusion

MathSAT 5

Z3

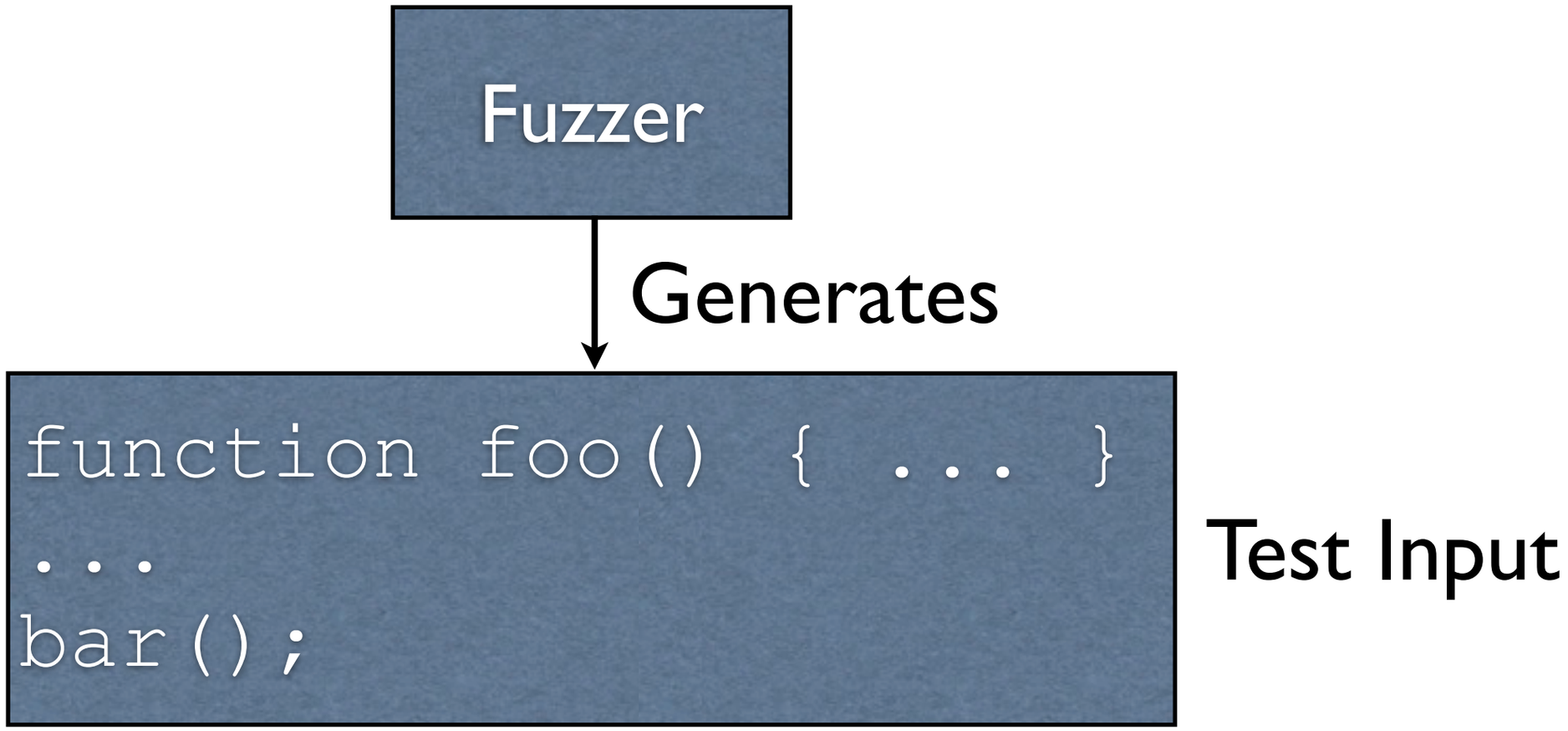
Automated Testing Motivation

- Writing correct software is hard
- Writing tests is time-consuming
- CPU cycles are cheap

Background: Fuzzing and Differential Testing

- Idea: generate an input via some process, known as a **fuzzer**
- Run input on different implementations
- If implementations disagree on result, bug has been found

Fuzzer


Fuzzer
Generates
function foo() { ... }
...
bar();
Test Input

Fuzzer

Generates

```
function foo() { ... }
...
bar();
```

Test Input

Executed on

Fuzzer

Generates

```
function foo() { ... }
...
bar();
```

Test Input

Executed on

Produce

42

42

53

Fuzzer

Generates

```
function foo() { ... }
...
bar();
```

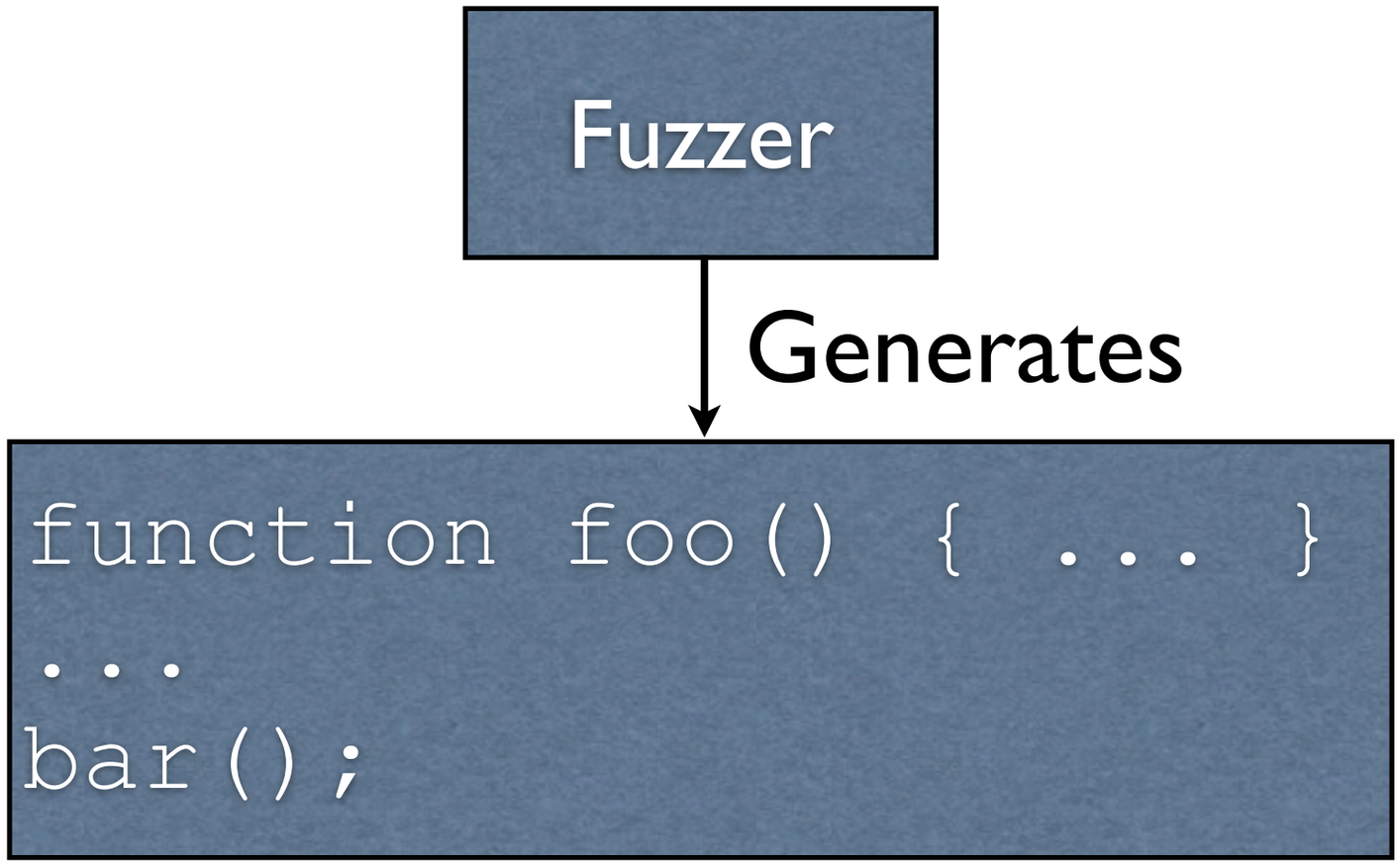
Test Input

Executed on

Produce

42

42 Mismatch: bug 53


Fuzzer
Generates
function foo() { ... }
...
bar();

Fuzzer 1

Generates

```
function foo() { ... }
...
bar();
```

Fuzzer 2

Generates

```
function baz() { ... }
...
blah();
```

Fuzzing is Broad

- Fuzzing is an active area of research
  - Many existing tools available
  - Many more fuzzing techniques available

Fuzzing is Broad

- Fuzzing is an active area of research
  - Many existing tools available
  - Many more fuzzing techniques available
- From a user perspective: which fuzzing technique should I apply for my domain?

Fuzzing is Broad

- Fuzzing is an active area of research
  - Many existing tools available
  - Many more fuzzing techniques available
- From a user perspective: which fuzzing technique should I apply for my domain?
- From a research perspective, why do we need a new fuzzing technique?

Fuzzing is Broad

- Fuzzing is an active area of research
  - Many existing tools available
  - Many more fuzzing techniques available

- From a user perspective: which fuzzing technique should I apply for my domain?
- From a research perspective, why do we need a new fuzzing technique?

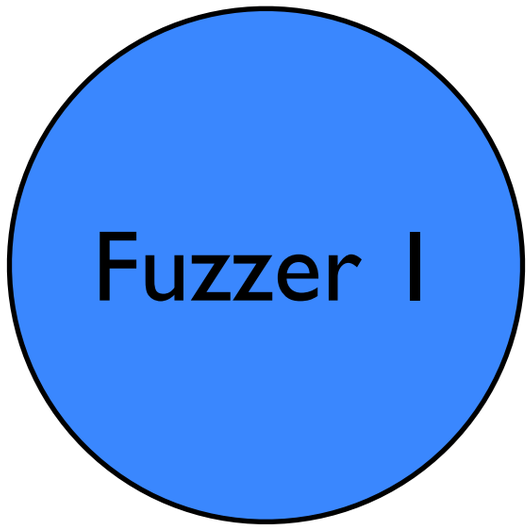
Answering these questions requires both a metric to compare fuzzers and a way of gathering this metric.

Outline

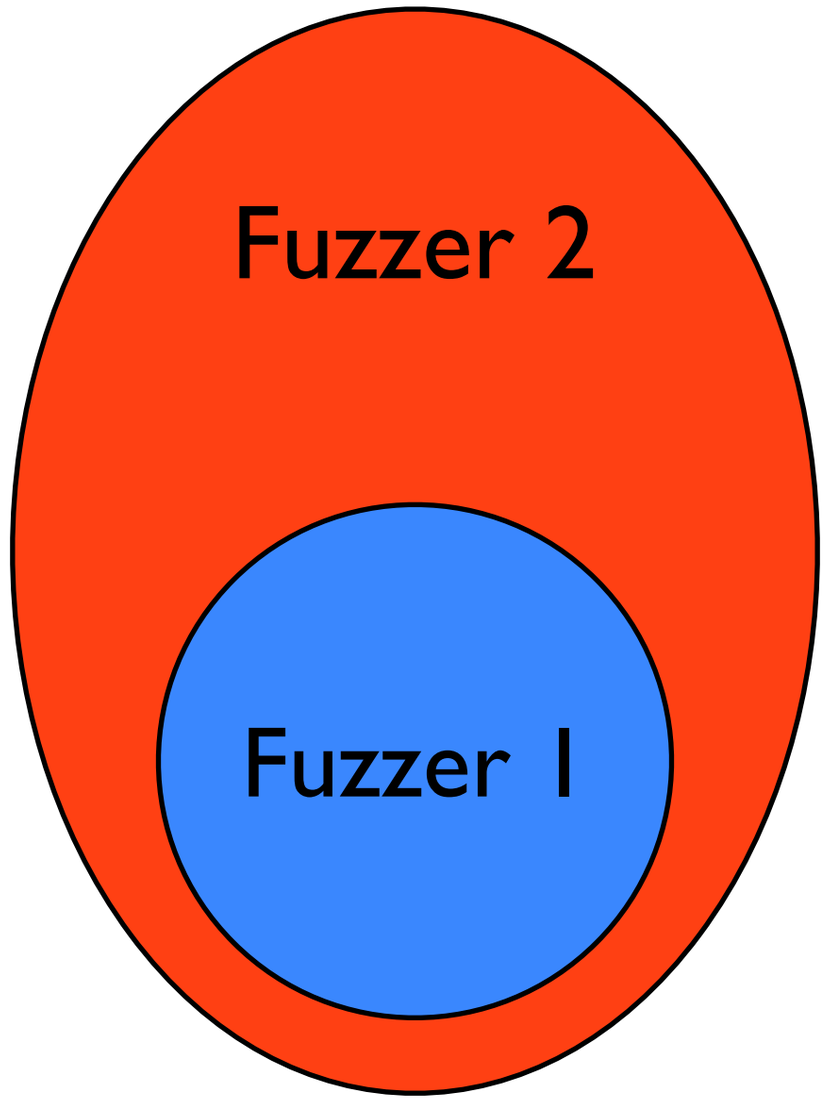
- Background
- Metrics used in the literature
- An automated approach
- How metrics compare
- Conclusion

“Obvious” metric: Unique Bugs Found per Unit Time

“Obvious” metric: Unique Bugs Found per Unit Time

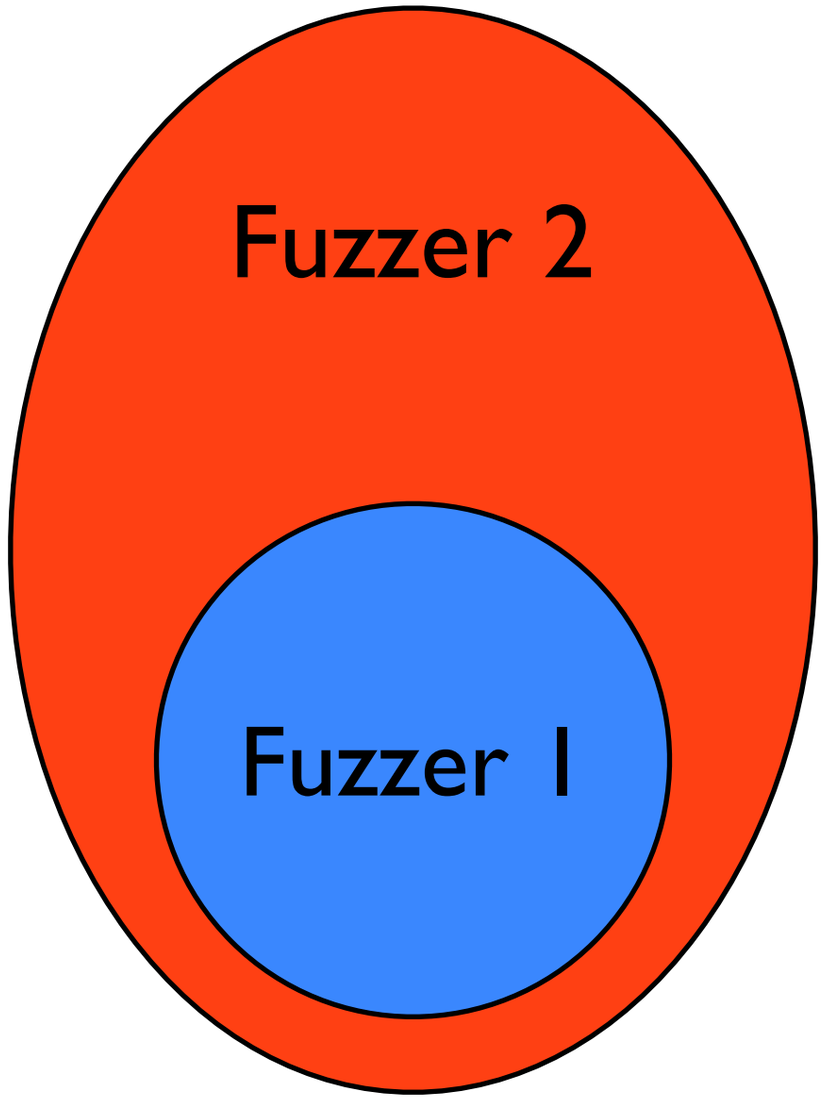


“Obvious” metric: Unique Bugs Found per Unit Time

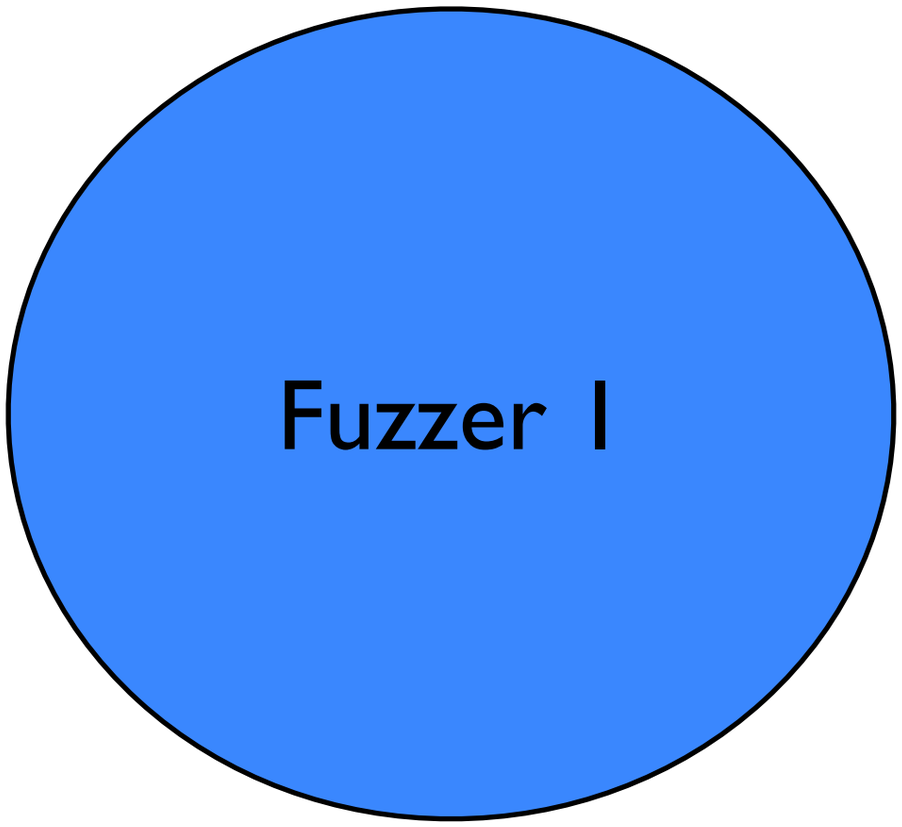


“Obvious” metric: Unique Bugs Found per Unit Time

Clear winner:
Fuzzer 2

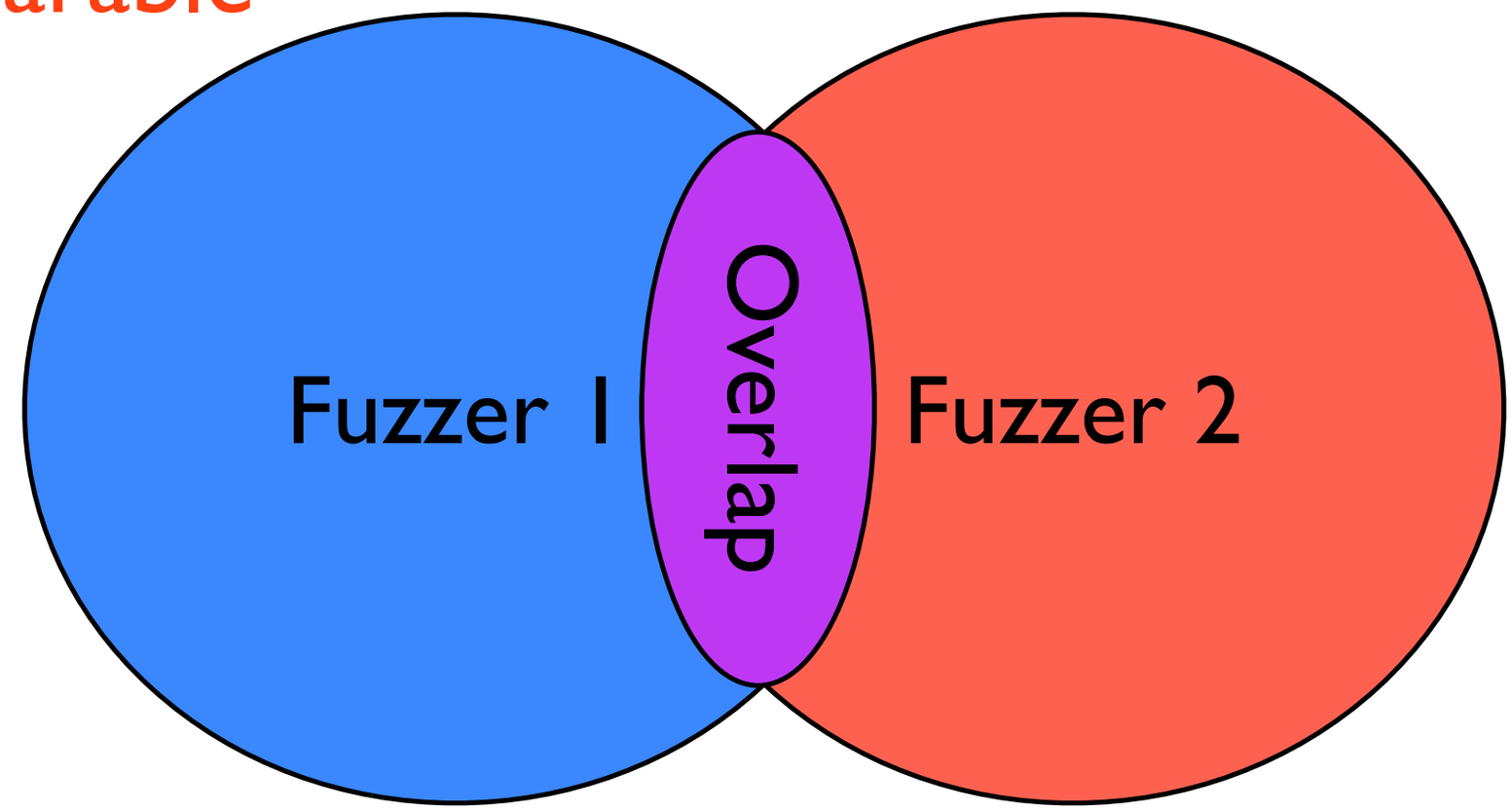


“Obvious” metric: Unique Bugs Found per Unit Time



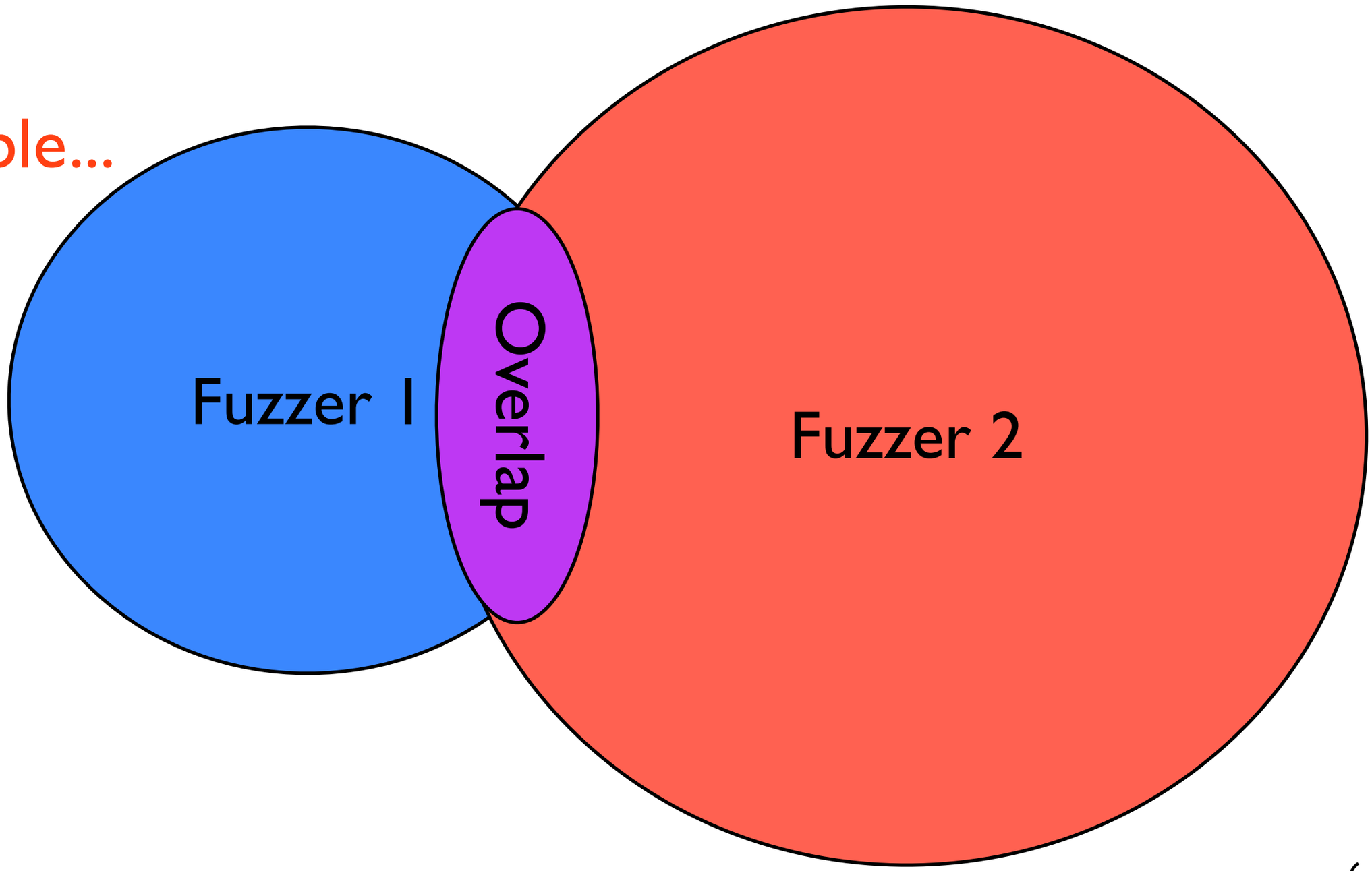
“Obvious” metric: Unique Bugs Found per Unit Time

Non-comparable



“Obvious” metric: Unique Bugs Found per Unit Time

Non-comparable...



“Obvious”

- This metric (unique bugs found) is **seldom** used
- Underlying reason why: it is **exceedingly** difficult to collect

Fuzzer

Generates

```
function foo() { ... }
...
bar();
```

Test Input

Executed on

Produce

42

42 Mismatch: bug 53

Fuzzer

Generates

```
function baz() { ... }
...
boo();
```

Test Input

Executed on

Produce

8

4 Mismatch: bug 8

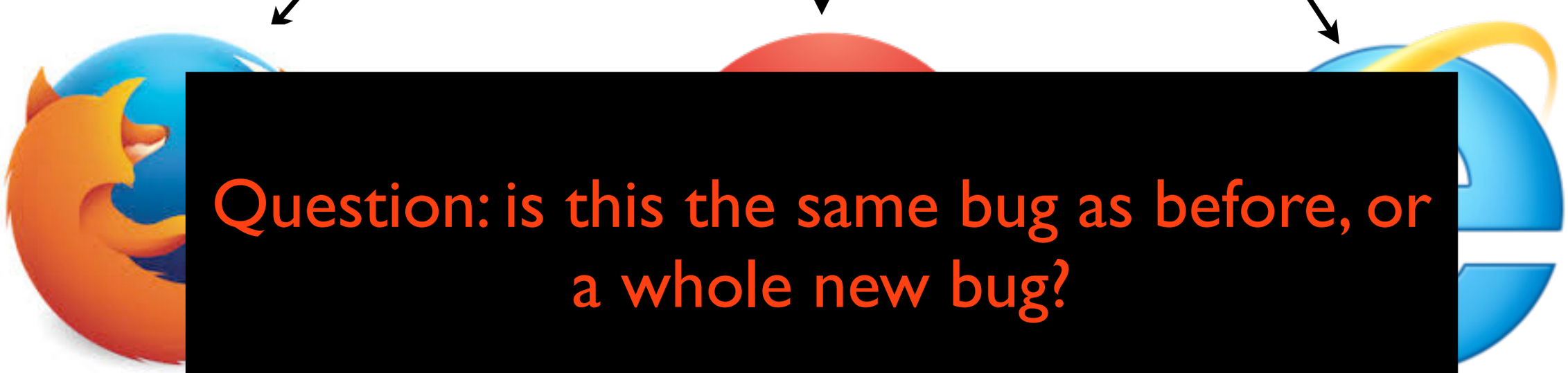
Fuzzer

Generates

```
function baz() { ... }
...
boo();
```

Test Input

Executed on



Question: is this the same bug as before, or a whole new bug?

Produce

8

4 Mismatch: bug 8

Unique Bugs?

- Ultimately, only the developers can answer this question
- Existing approaches require developer feedback and lots of manual effort
- End result: only **one** quantitative comparison of different fuzzing techniques in the literature uses this metric (that we know of), and it **was done incorrectly**

Workaround

- A number of "surrogate" metrics have been used, which are more easily collected
- The fundamental validity of these metrics has never before been evaluated

Surrogate #1: Crashes with Stack Traces

Surrogate #1: Crashes with Stack Traces

Input: `217`

Output:

```
ASSERTION VIOLATION: line 32 in main.c
```

Surrogate #1: Crashes with Stack Traces

Input: `217`

Output:

```
ASSERTION VIOLATION: line 32 in main.c
```

Input: `438`

Output:

```
ASSERTION VIOLATION: line 32 in main.c
```

Surrogate #1: Crashes with Stack Traces

Input: `217`

Output:

```
ASSERTION VIOLATION: line 32 in main.c
```

Input: `438`

Output:

```
ASSERTION VIOLATION: line 32 in main.c
```

Input: `60`

Output:

```
ASSERTION VIOLATION: line 91 in foo.c
```

Surrogate #1: Crashes with Stack Traces

Input: `217`

Bug 1

Output:

```
ASSERTION VIOLATION: line 32 in main.c
```

Input: `438`

Output:

```
ASSERTION VIOLATION: line 32 in main.c
```

Input: `60`

Output:

```
ASSERTION VIOLATION: line 91 in foo.c
```

Surrogate #1: Crashes with Stack Traces

Input: `217`

Bug 1

Output:

```
ASSERTION VIOLATION: line 32 in main.c
```

Input: `438`

Output:

```
ASSERTION VIOLATION: line 32 in main.c
```

Input: `60`

Bug 2

Output:

```
ASSERTION VIOLATION: line 91 in foo.c
```

Surrogate #1: Crashes with Stack Traces

- Problem: ignores correctness bugs entirely
    - In so doing, assumes that crash bugs behave similarly as correctness bugs
    - Multiple fuzzing techniques exist which are specialized for finding correctness bugs
    - Ergo, we know this assumption is invalid in general

Surrogate #2: Count Bug-Inducing Inputs

Surrogate #2: Count Bug-Inducing Inputs

```
function foo() { ... }
...
bar();
```

Surrogate #2: Count Bug-Inducing Inputs

```
function foo() { ... }
...
bar();
```

```
function baz() { ... }
...
boo();
```

Surrogate #2: Count Bug-Inducing Inputs

```
function foo() { ... }
...
bar();
```

Two Bug-Inducing Inputs Found

```
function baz() { ... }
...
boo();
```

Surrogate #2: Count Bug-Inducing Inputs

```
function foo() { ... }
...
bar();
```

Surrogate #2: Count Bug-Inducing Inputs

```
function foo() { ... }
...
bar();
```

Same Input

```
function foo() { ... }
...
bar();
```

Surrogate #2: Count Bug-Inducing Inputs

```
function foo() { ... }
...
bar();
```

Two Bug-Inducing Inputs Found

Same Input

```
function foo() { ... }
...
bar();
```

Surrogate #2: Count Bug-Inducing Inputs

```
function foo() { ... }
...
bar();
```

Two Bug-Inducing Inputs Found

Same Input

```
function foo() { ... }
...
bar();
```

Techniques exist which very nearly do this.

Surrogate #2: Count Bug-Inducing Inputs

- Assumes that there is a one-to-one correspondence between bug-inducing inputs and bugs
  - No evidence exists to confirm or refute this assumption in general
  - Can very clearly break under certain circumstances
- This metric is easy to measure and **wildly popular**

Outline

- Background
- Metrics used in the literature
- An automated approach
- How metrics compare
- Conclusion

Goal for An Automated Approach

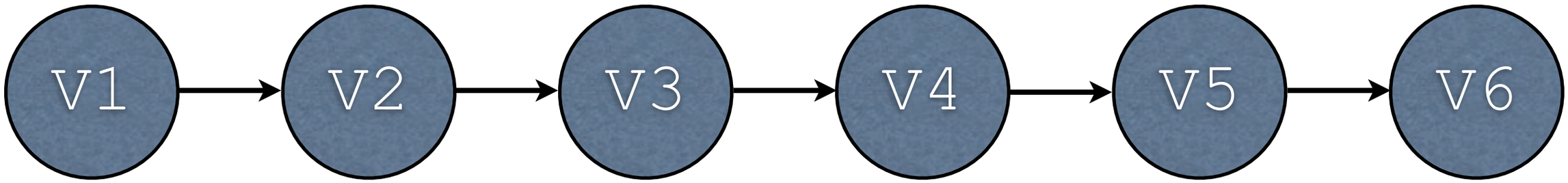
- Goal: be able to **automatically derive** the number of unique bugs found by a given fuzzer
  - For both crash and correctness bugs
  - Minimal manual effort

Assumptions

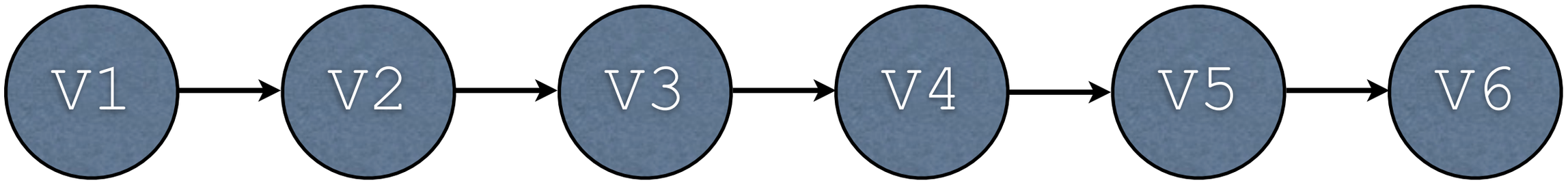
- The system under test is under version control, and each commit fixes at most one bug
- Past bugs behave similarly to new bugs
  - We will actually test an **older version** of the software

Approach

Approach

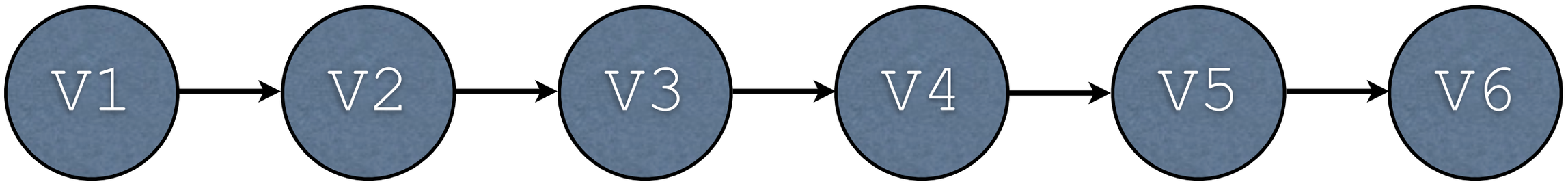


Approach



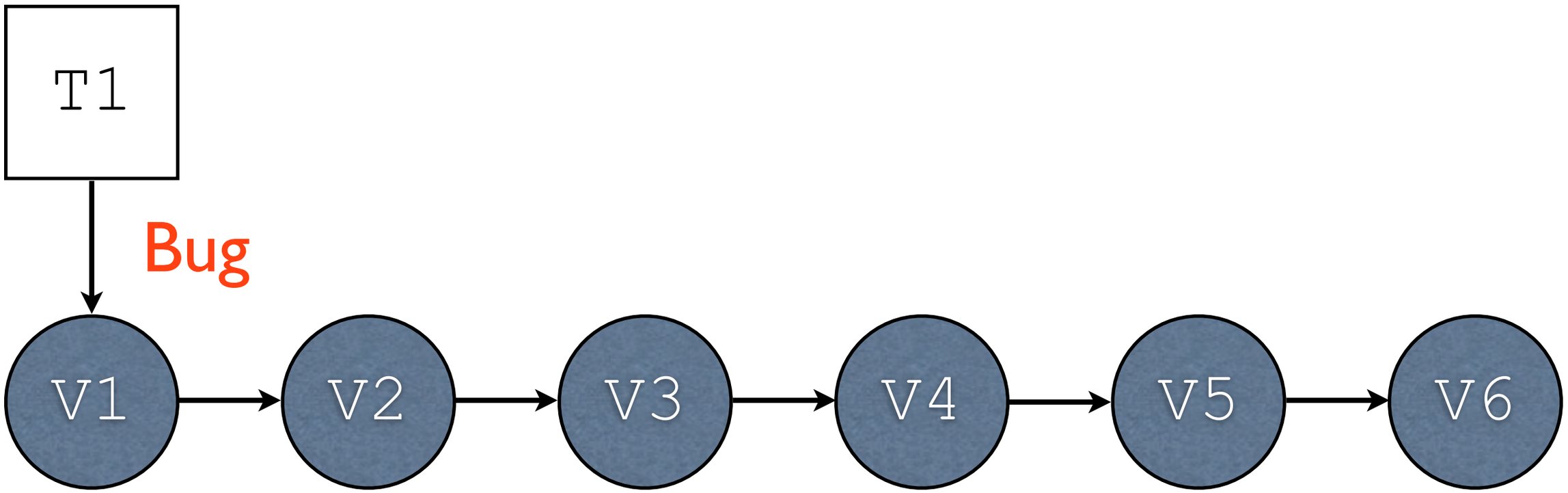
Three tests (`T1`, `T2`, `T3`) expose some number of bugs ≤ 3 on version `V1`

Approach



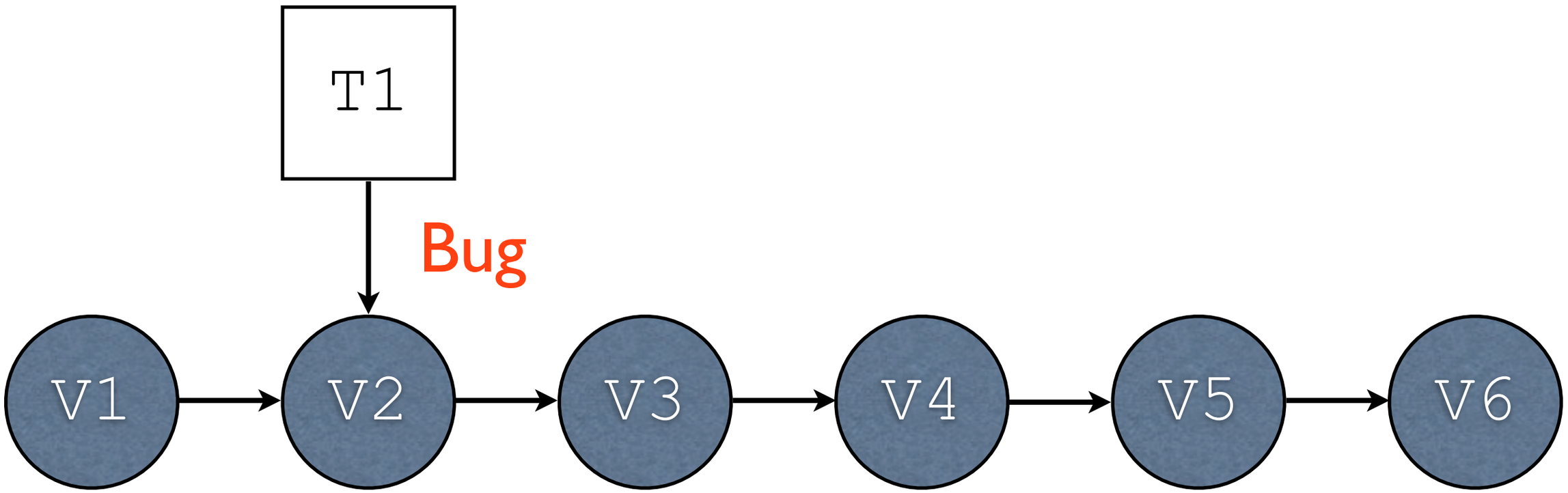
Idea: walk the versions until we hit one where a given test no longer triggers a bug.

Approach



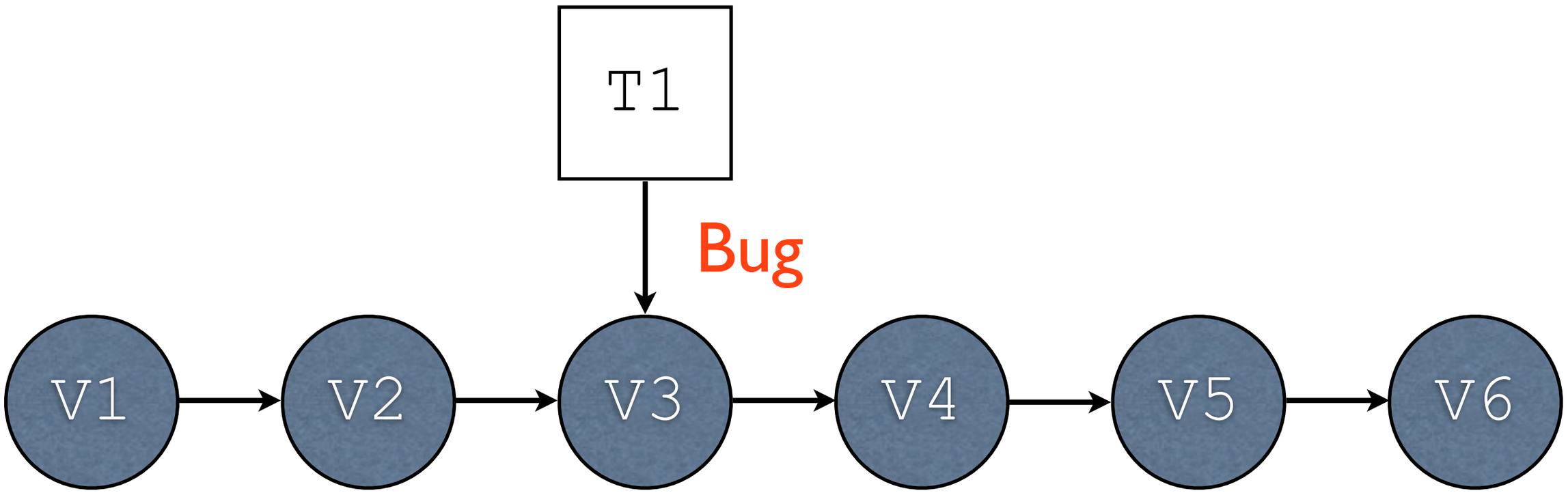
Idea: walk the versions until we hit one where a given test no longer triggers a bug.

Approach



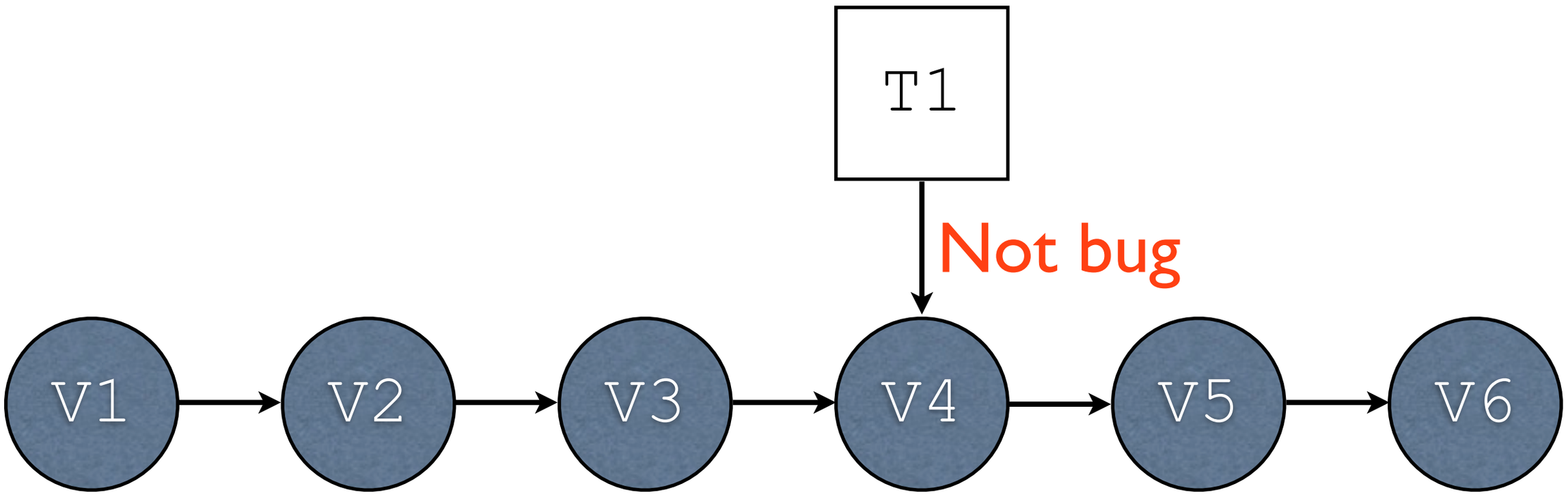
Idea: walk the versions until we hit one where a given test no longer triggers a bug.

Approach



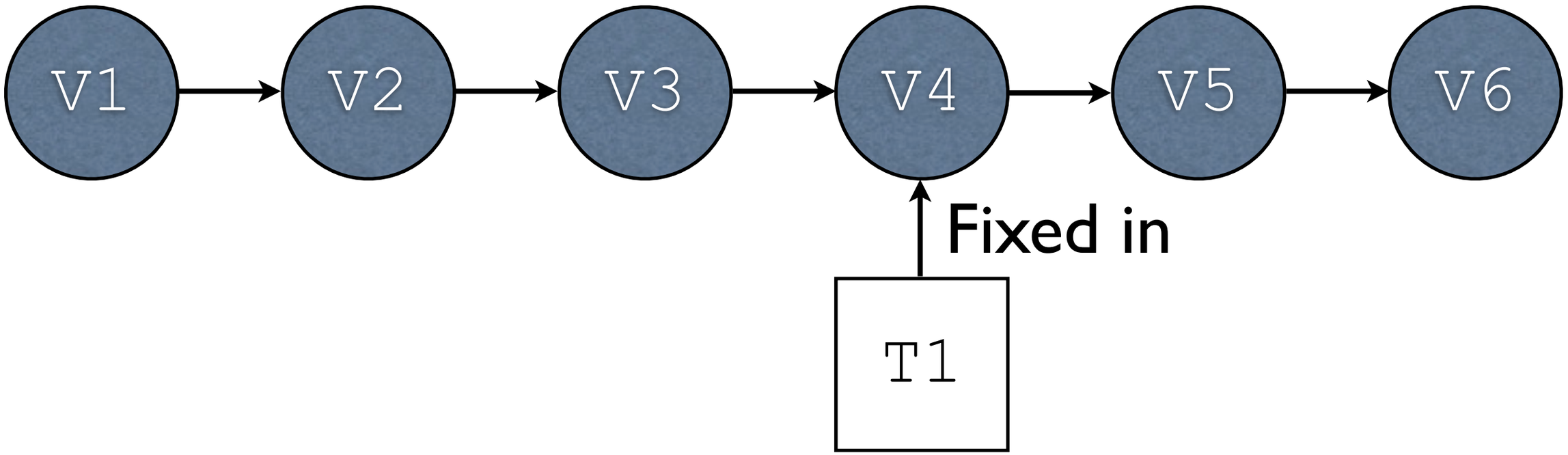
Idea: walk the versions until we hit one where a given test no longer triggers a bug.

Approach



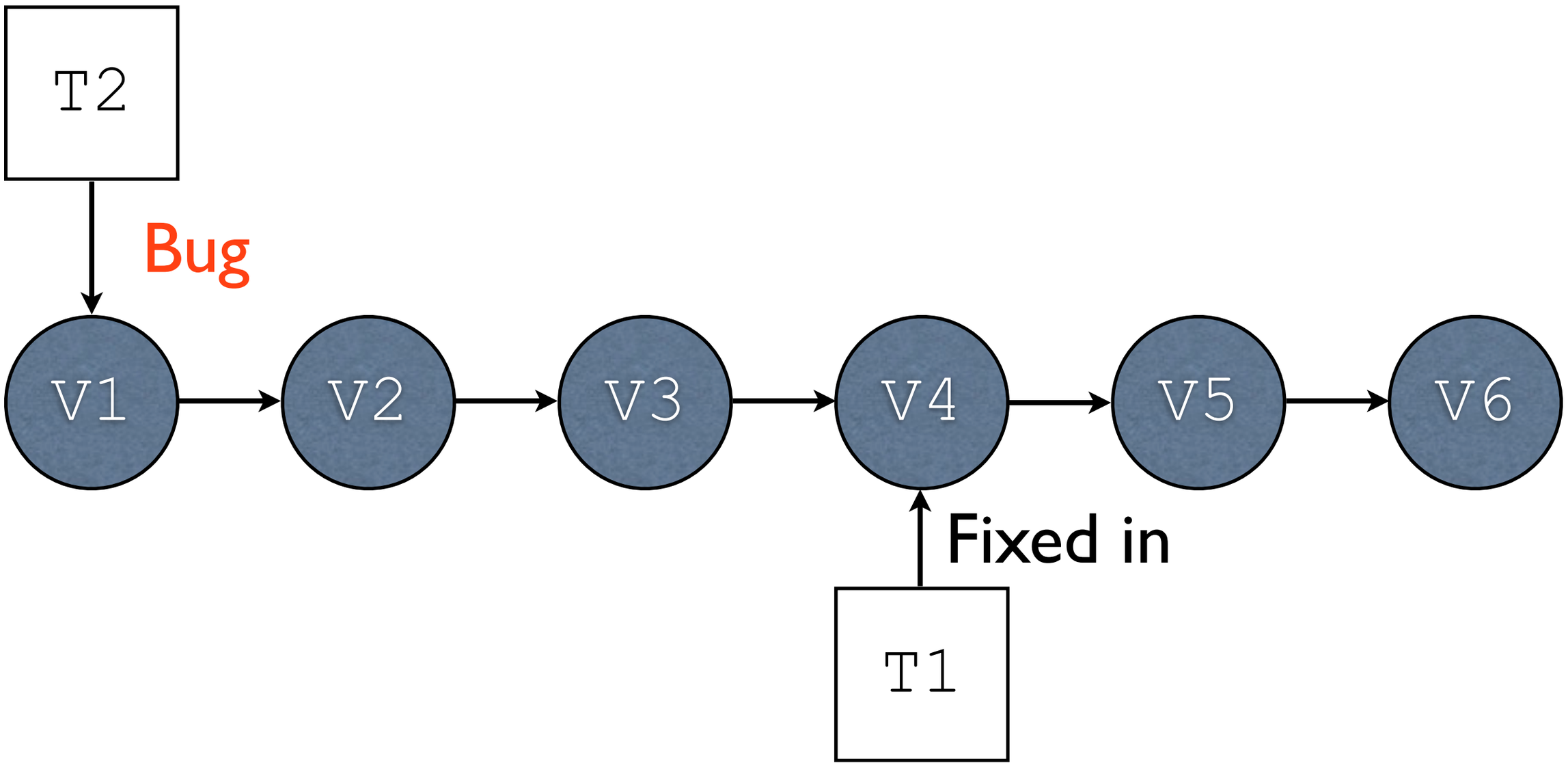
Idea: walk the versions until we hit one where a given test no longer triggers a bug.

Approach



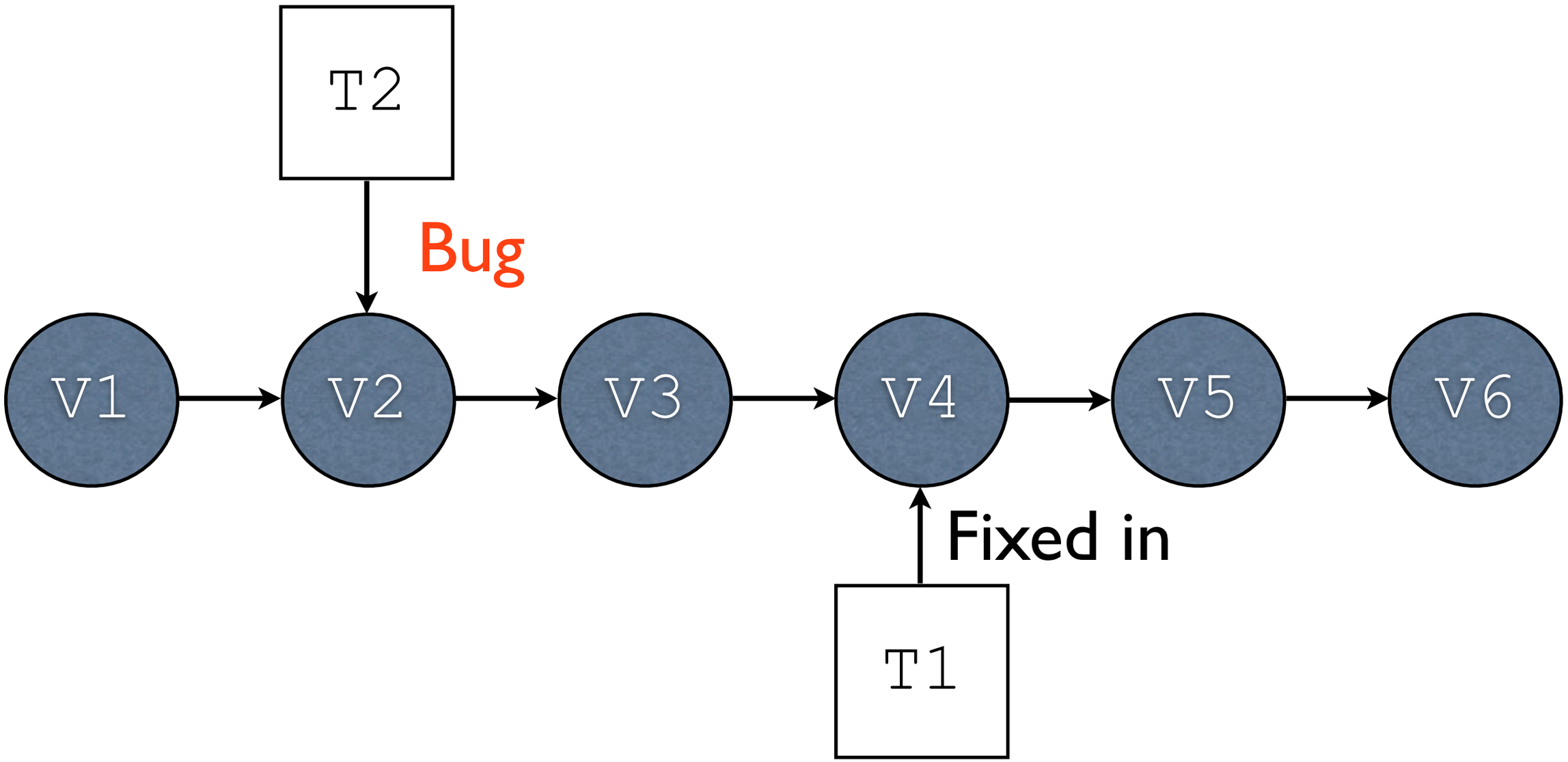
Idea: walk the versions until we hit one where a given test no longer triggers a bug.

Approach



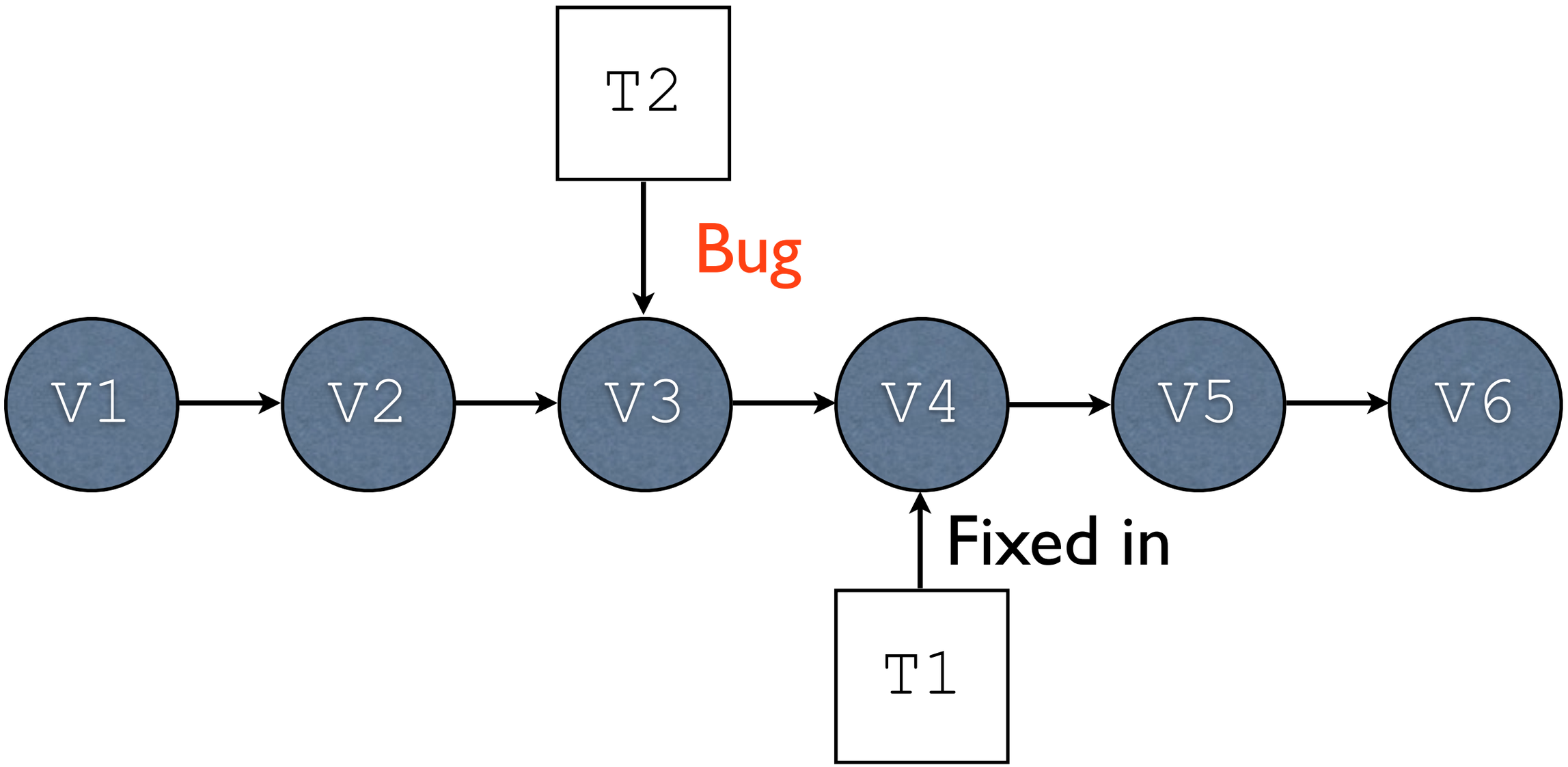
Idea: walk the versions until we hit one where a given test no longer triggers a bug.

Approach



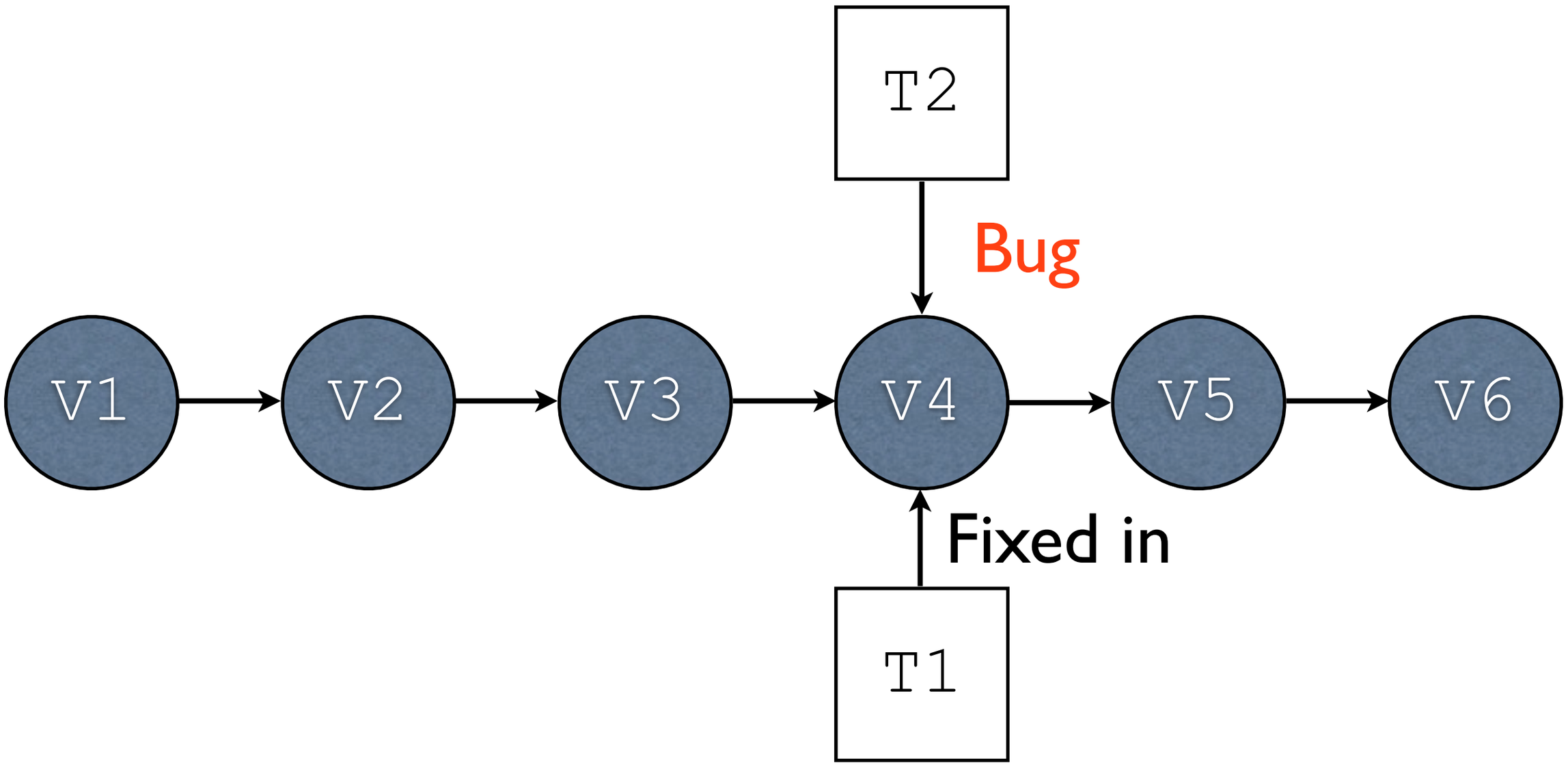
Idea: walk the versions until we hit one where a given test no longer triggers a bug.

Approach



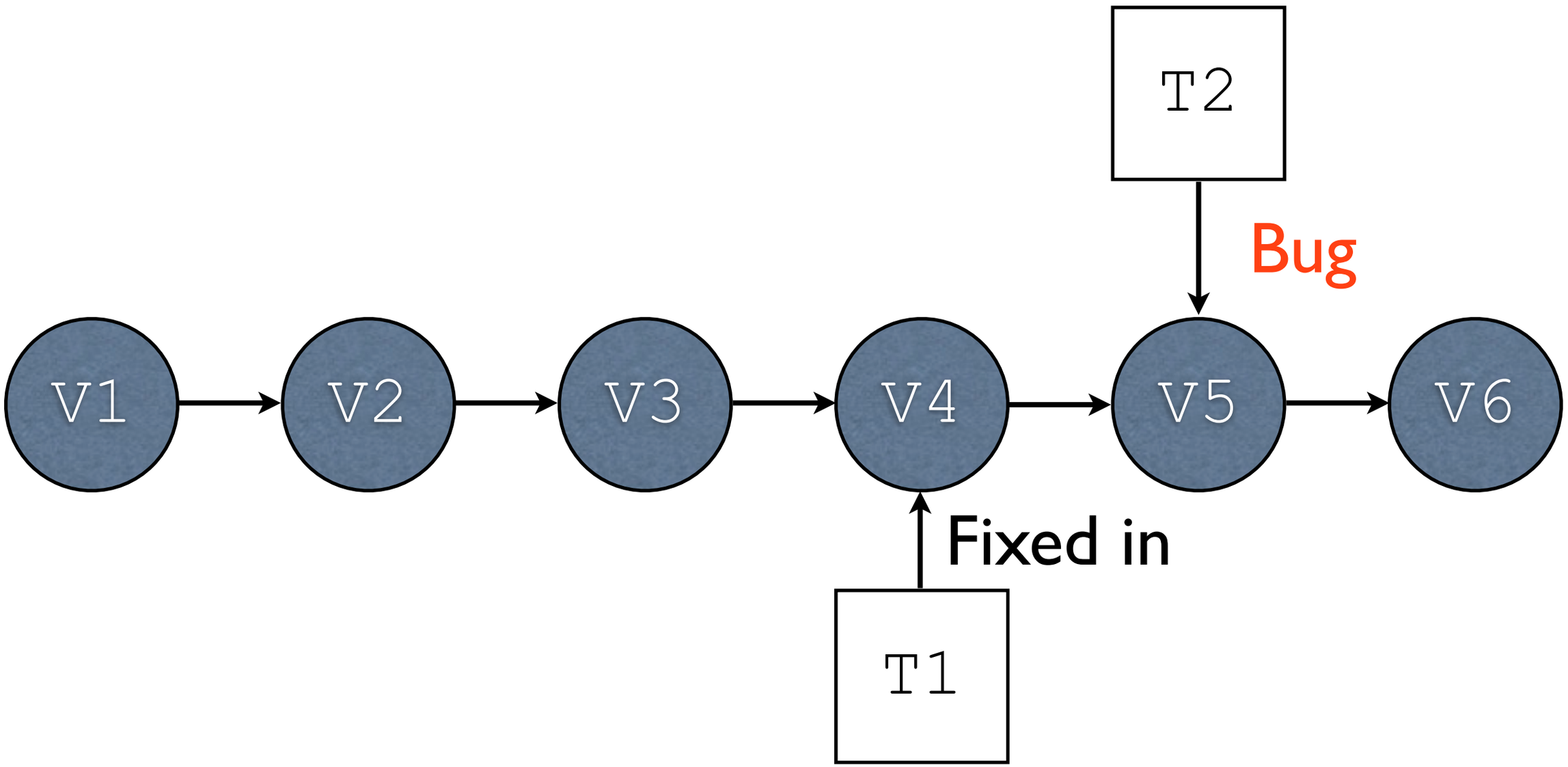
Idea: walk the versions until we hit one where a given test no longer triggers a bug.

Approach



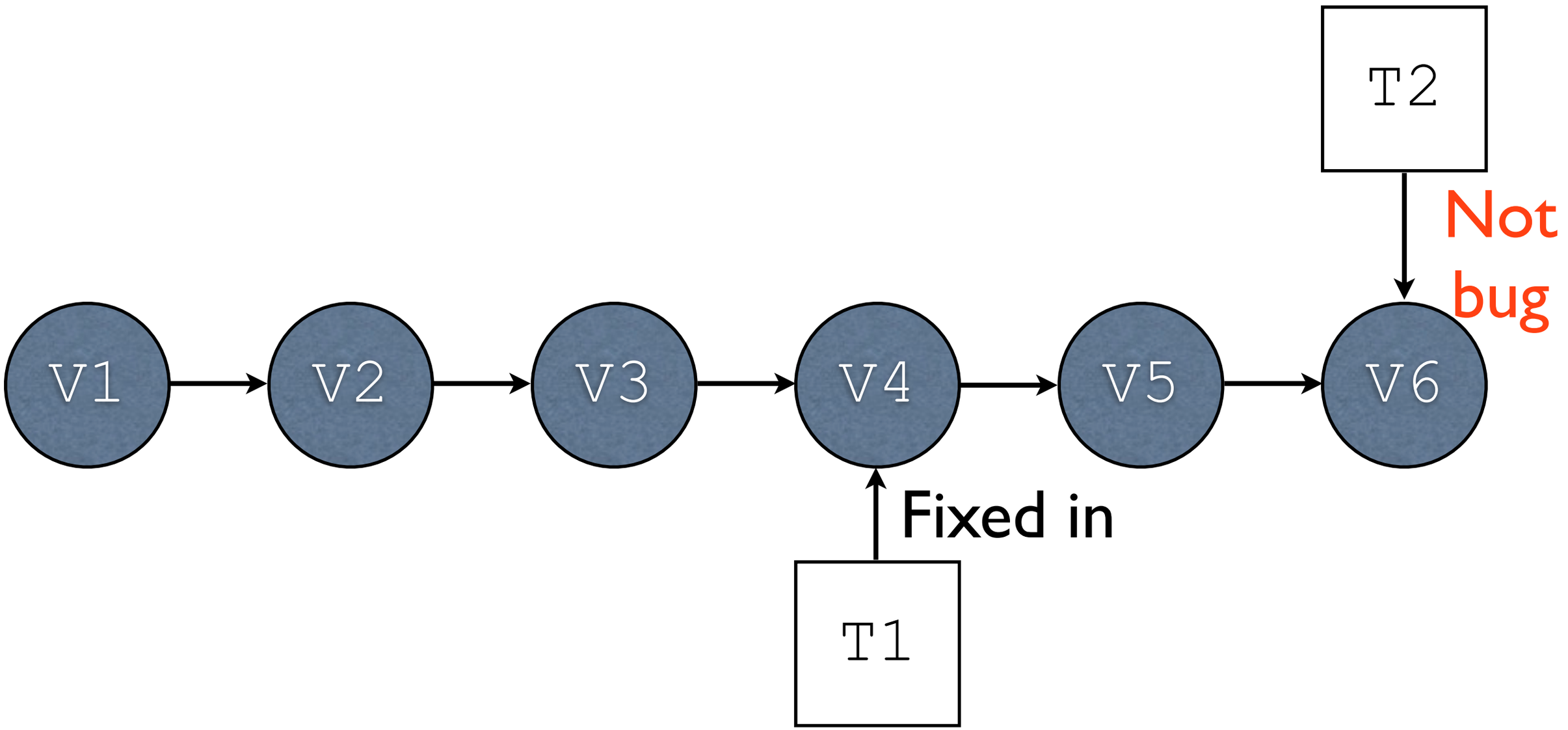
Idea: walk the versions until we hit one where a given test no longer triggers a bug.

Approach



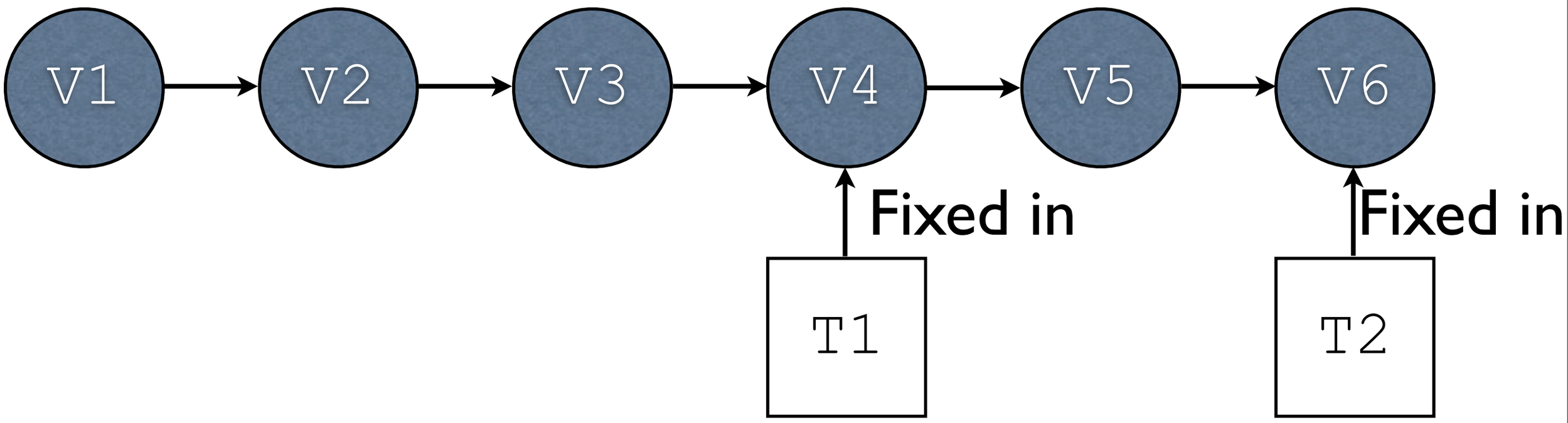
Idea: walk the versions until we hit one where a given test no longer triggers a bug.

Approach



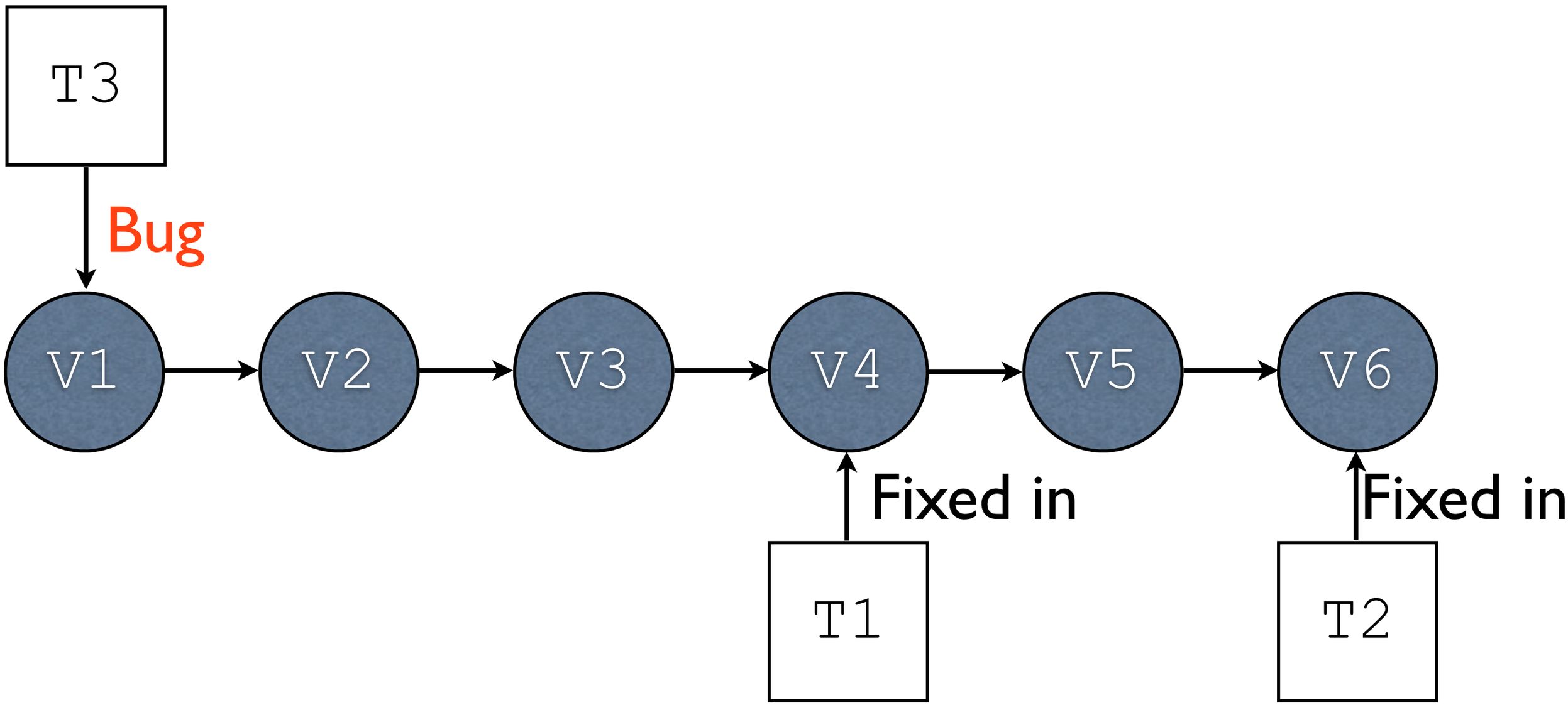
Idea: walk the versions until we hit one where a given test no longer triggers a bug.

Approach



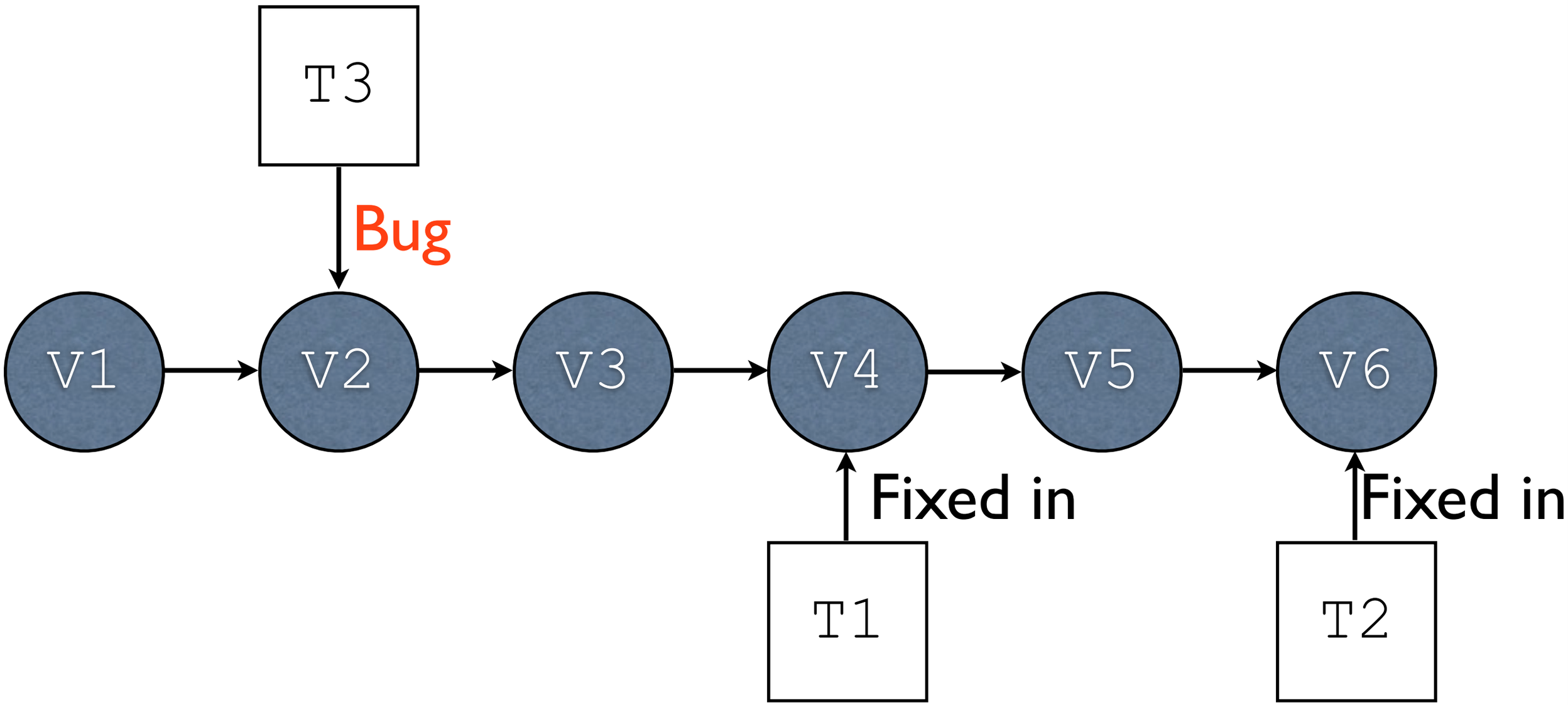
Idea: walk the versions until we hit one where a given test no longer triggers a bug.

Approach



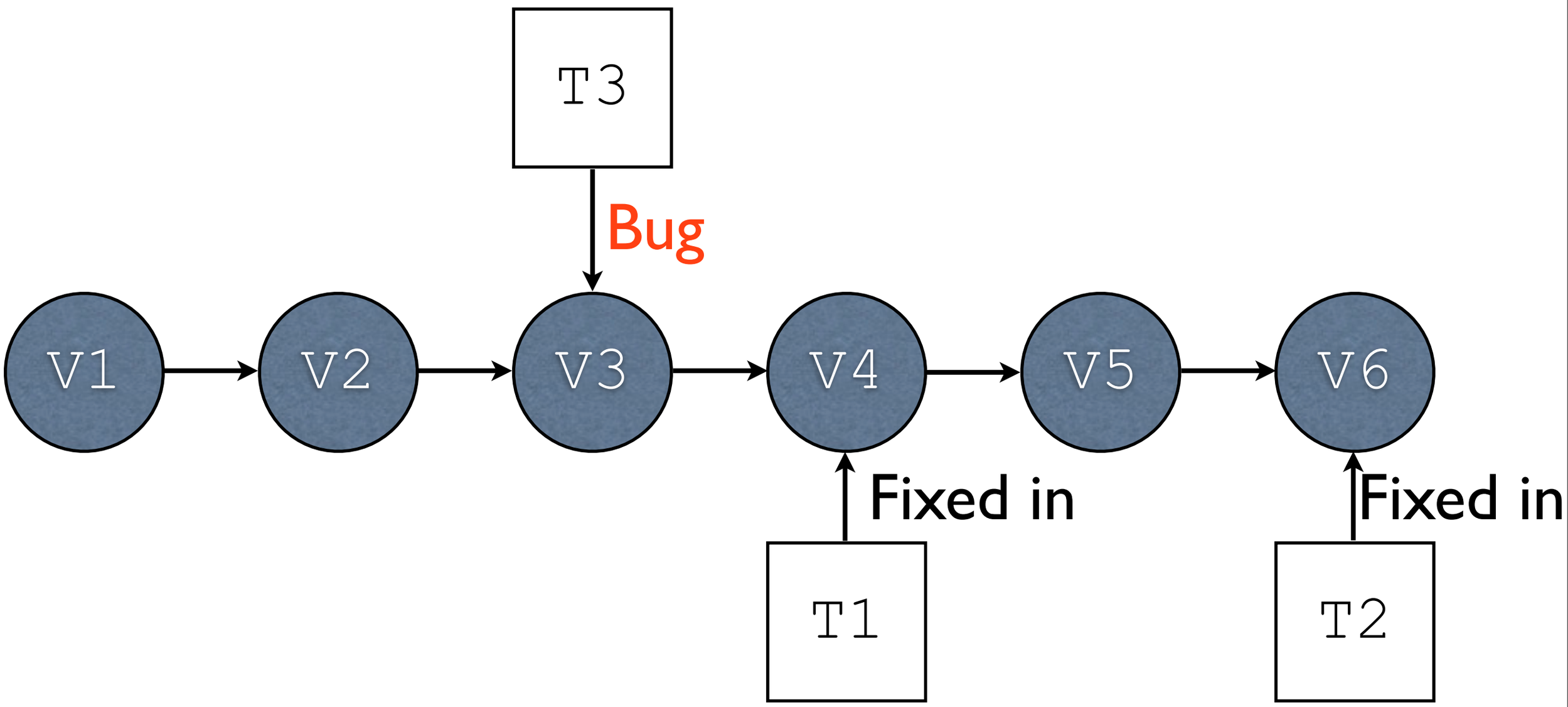
Idea: walk the versions until we hit one where a given test no longer triggers a bug.

Approach



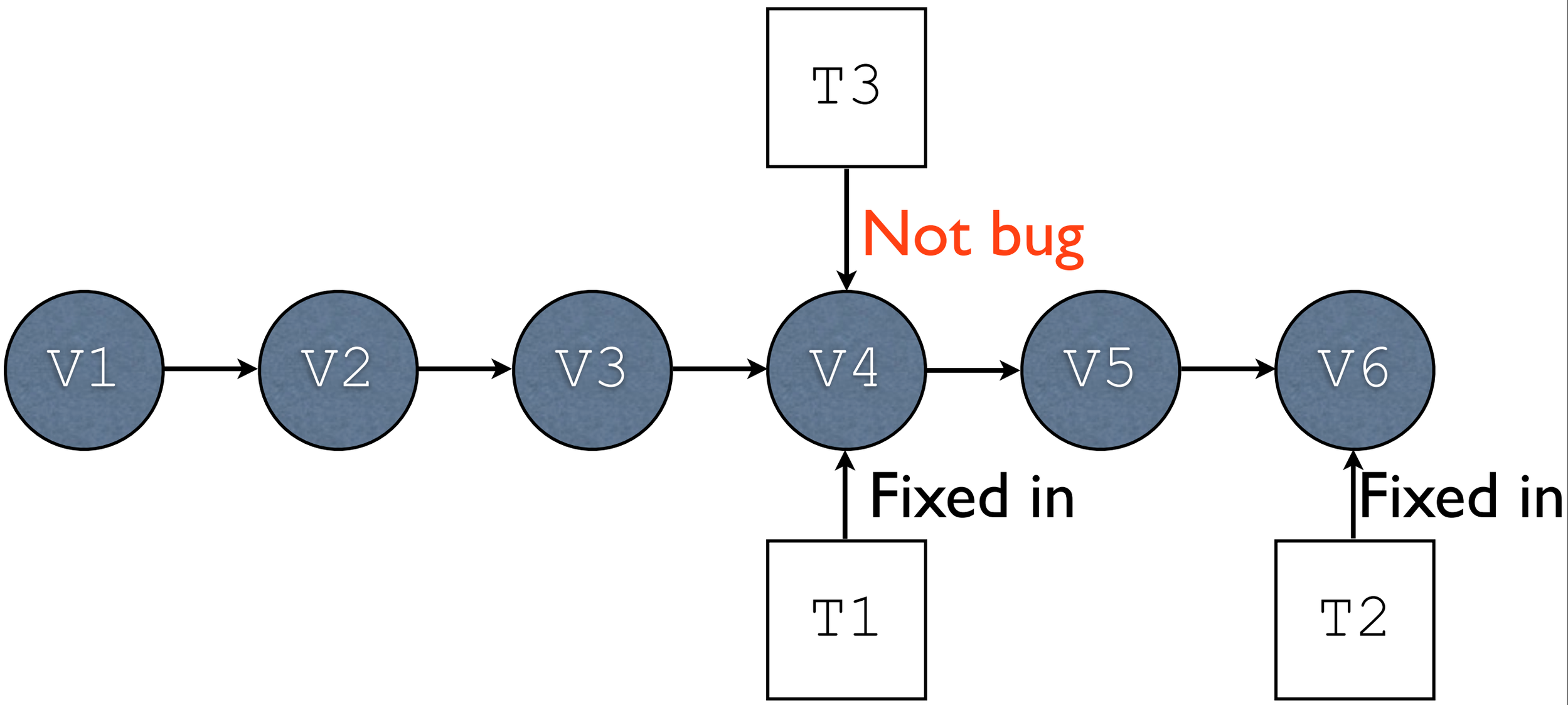
Idea: walk the versions until we hit one where a given test no longer triggers a bug.

Approach



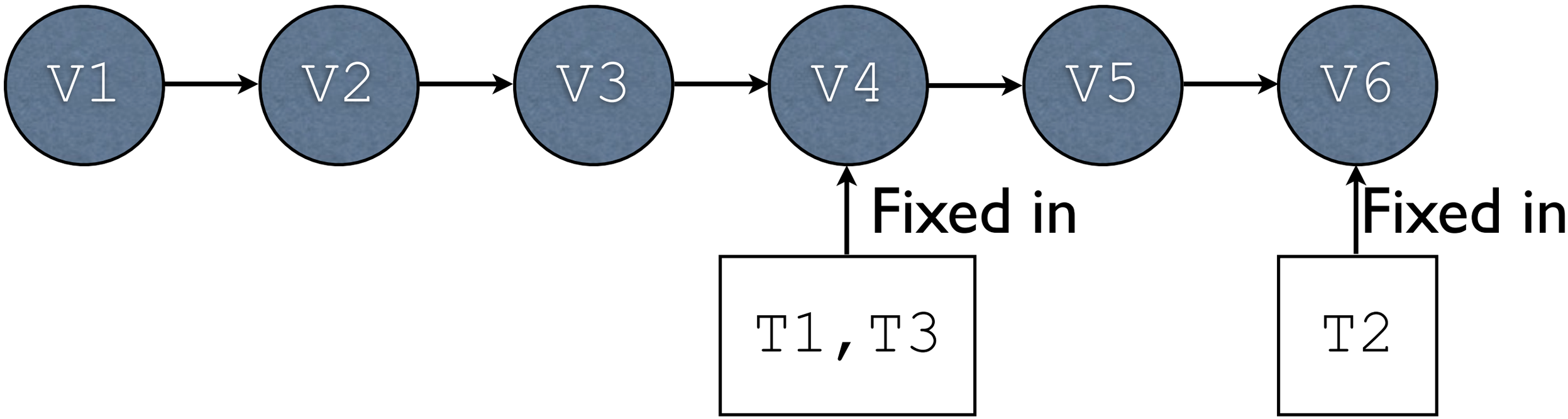
Idea: walk the versions until we hit one where a given test no longer triggers a bug.

Approach



Idea: walk the versions until we hit one where a given test no longer triggers a bug.

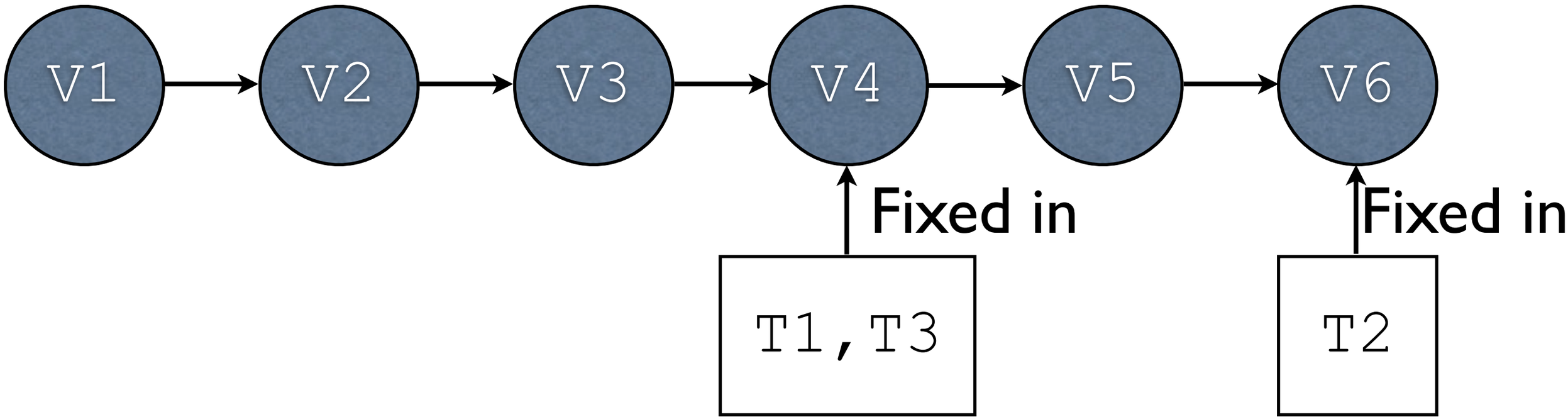
Approach



By the assumption that each version fixes at most one bug, `T1` and `T3` both trigger one bug (fixed in `V4`), and `T2` triggers another bug (fixed in `V6`).

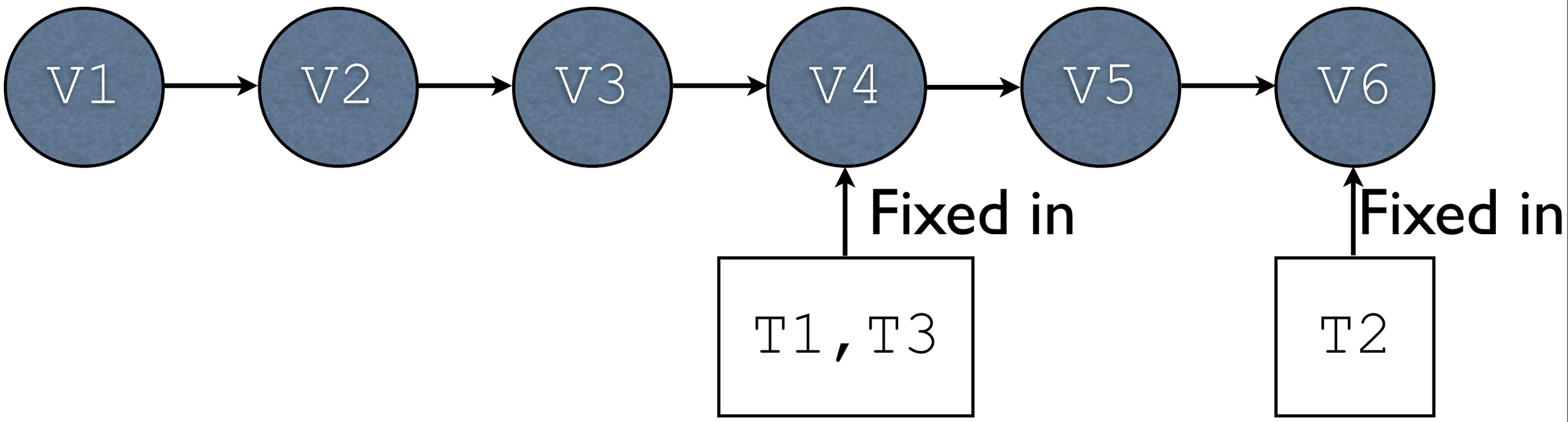
Approach

End result: two unique bugs found



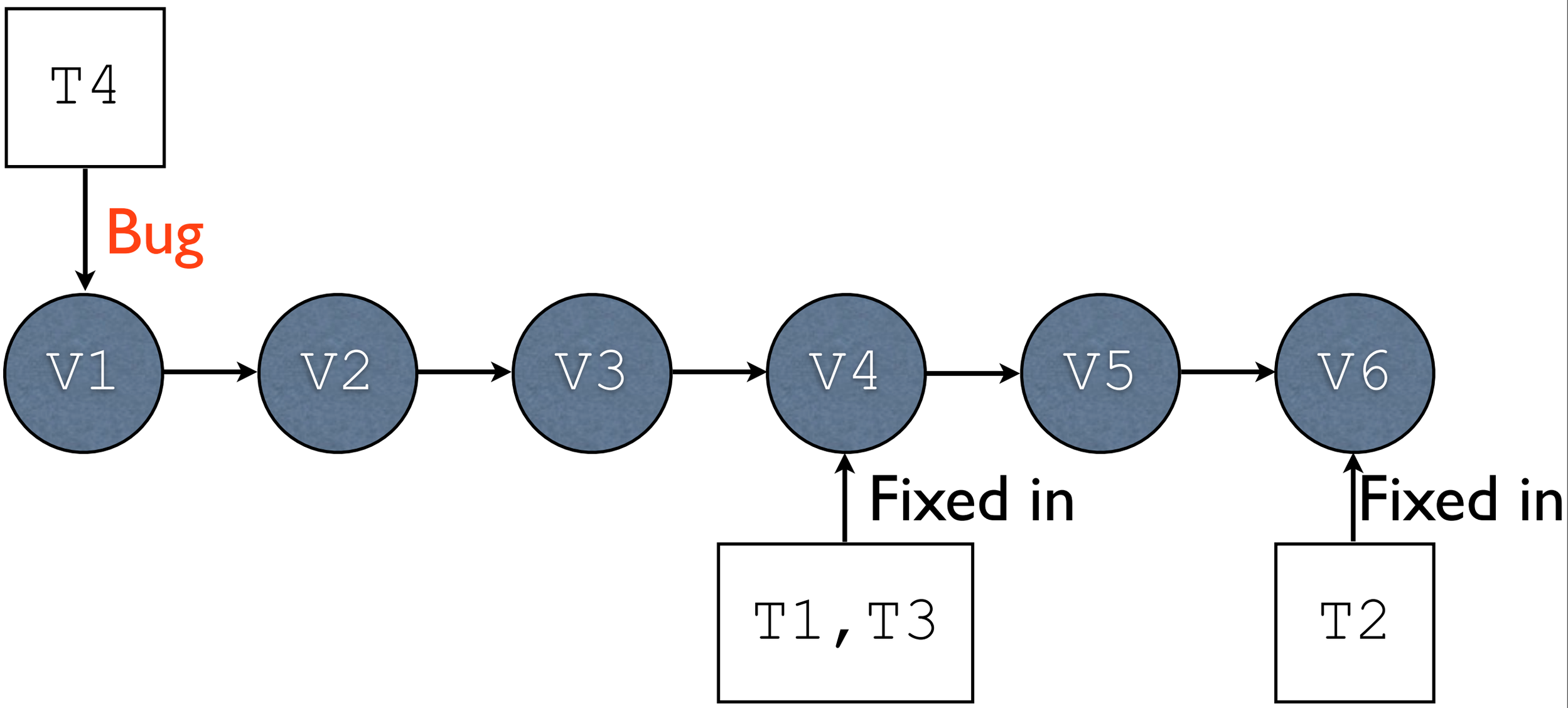
By the assumption that each version fixes at most one bug, `T1` and `T3` both trigger one bug (fixed in `V4`), and `T2` triggers another bug (fixed in `V6`).

Approach



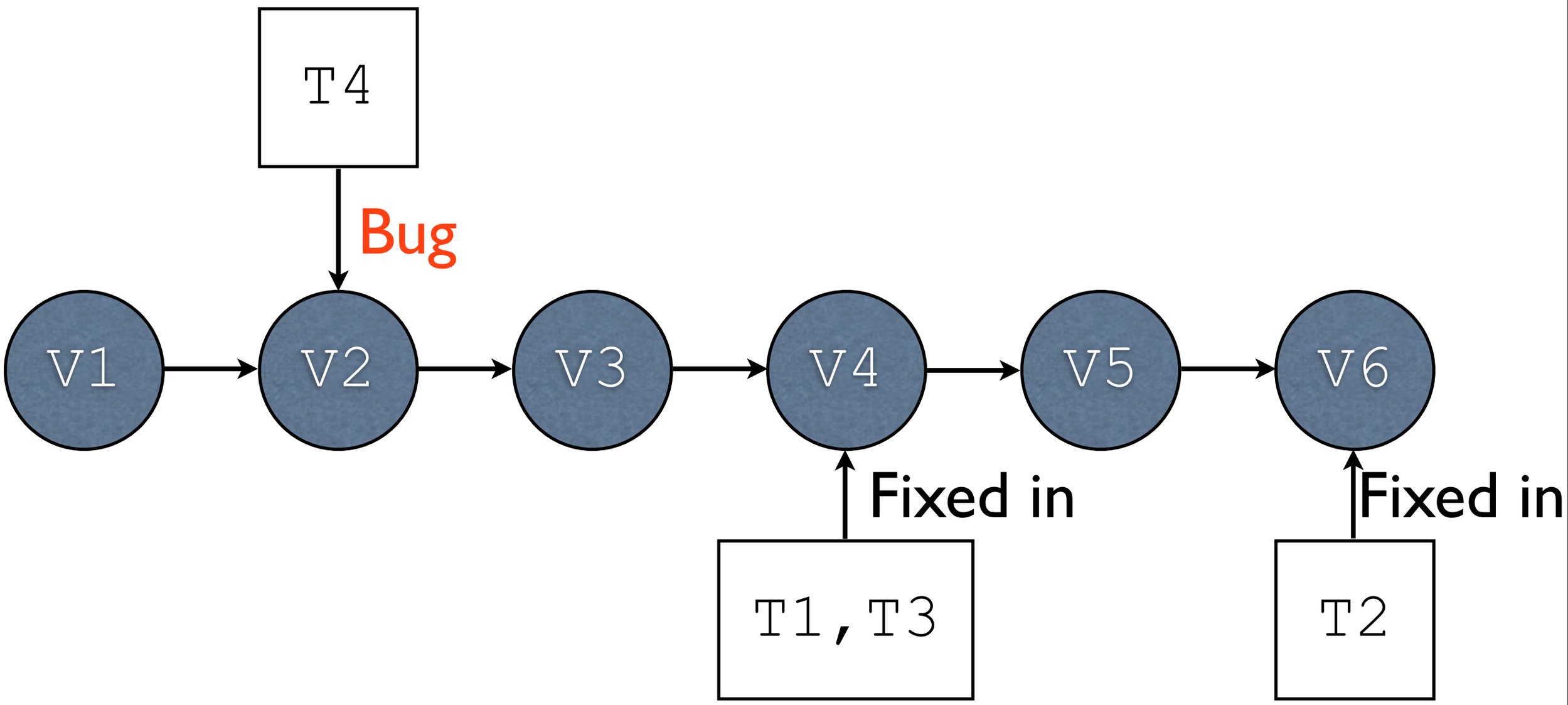
We could run out of versions on some inputs...

Approach



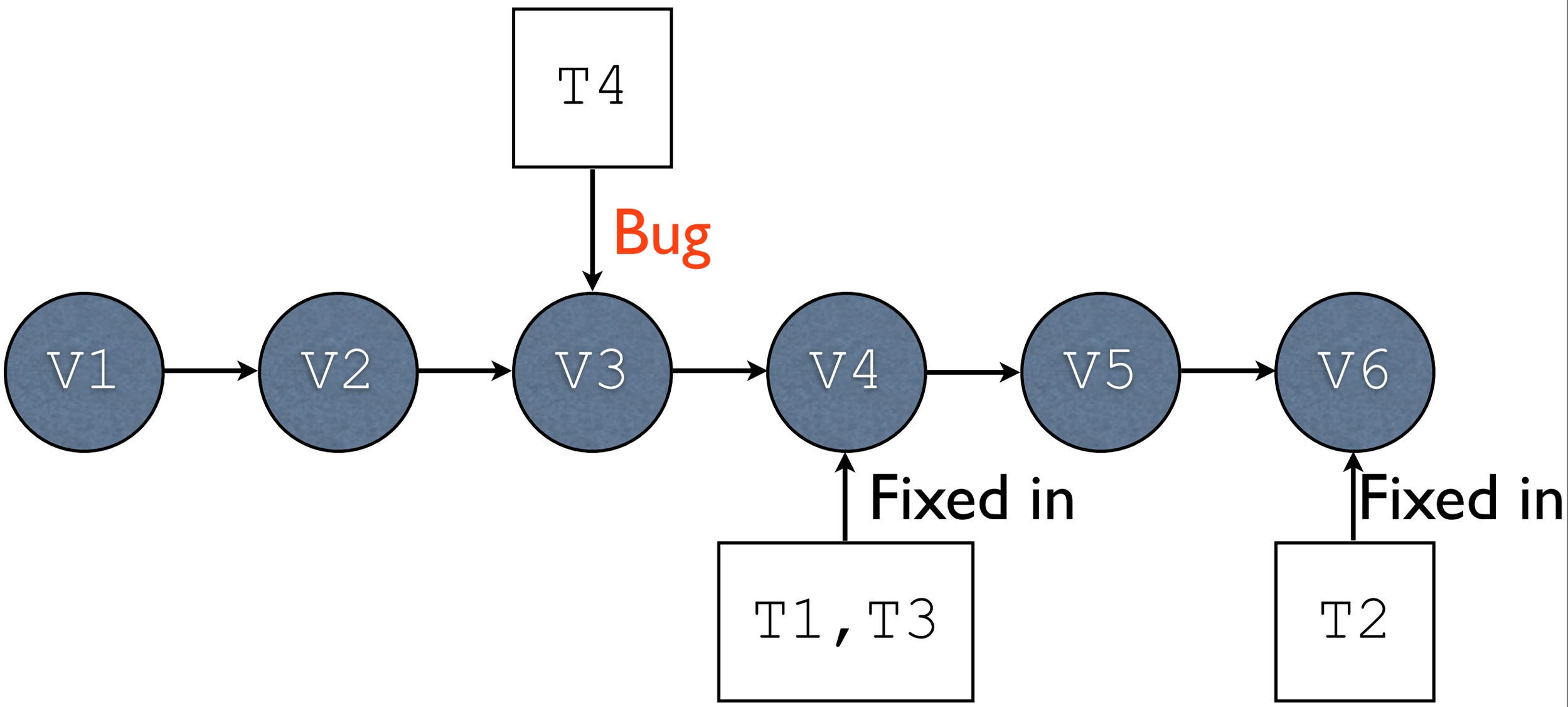
We could run out of versions on some inputs...

Approach



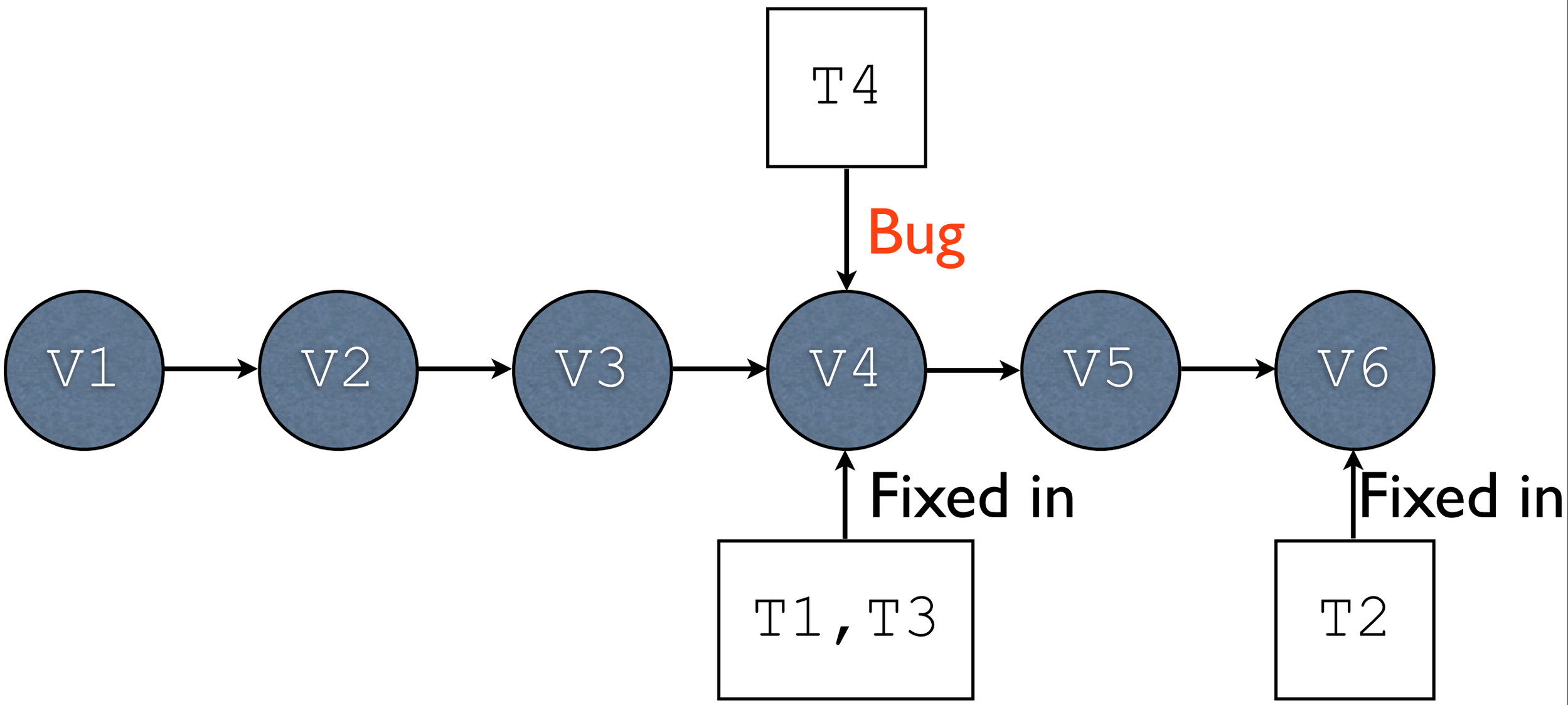
We could run out of versions on some inputs...

Approach



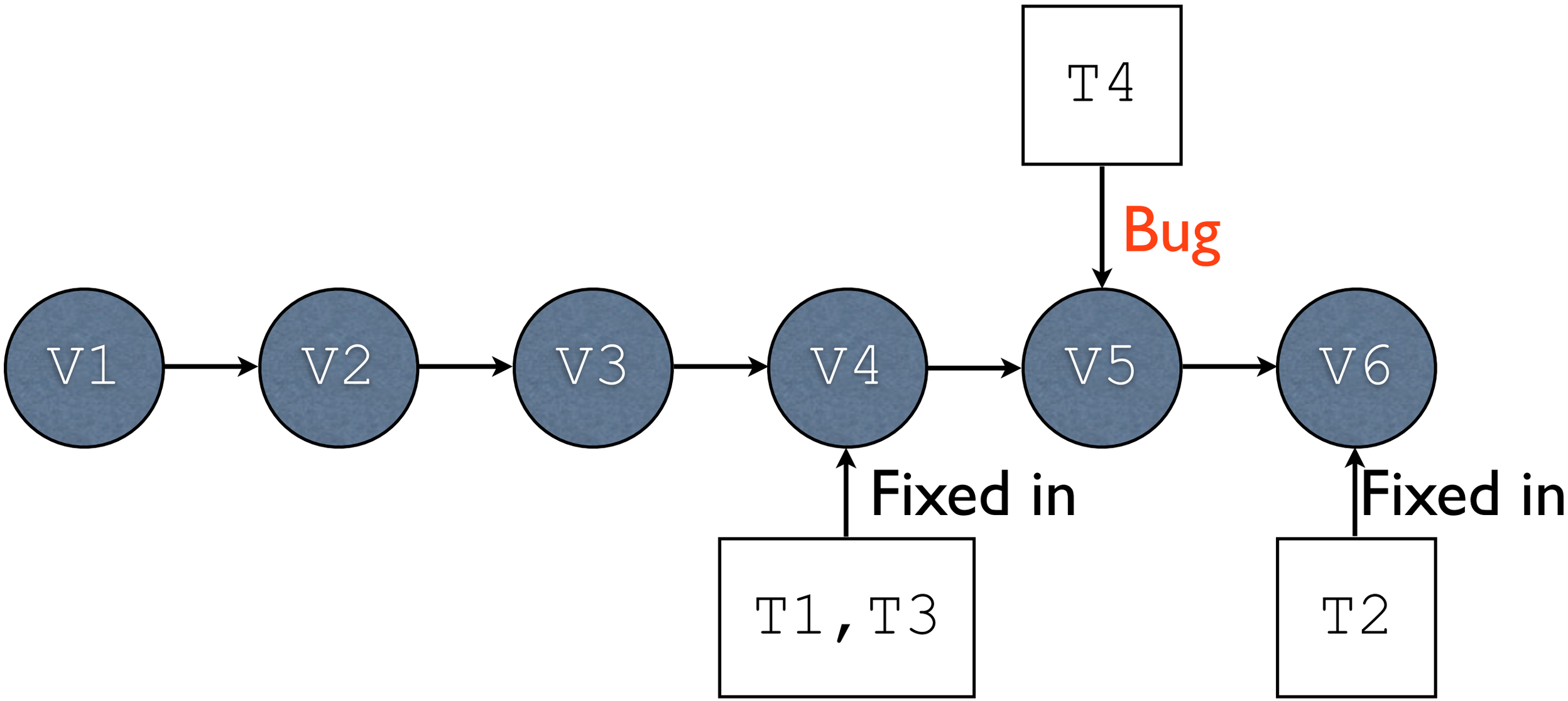
We could run out of versions on some inputs...

Approach



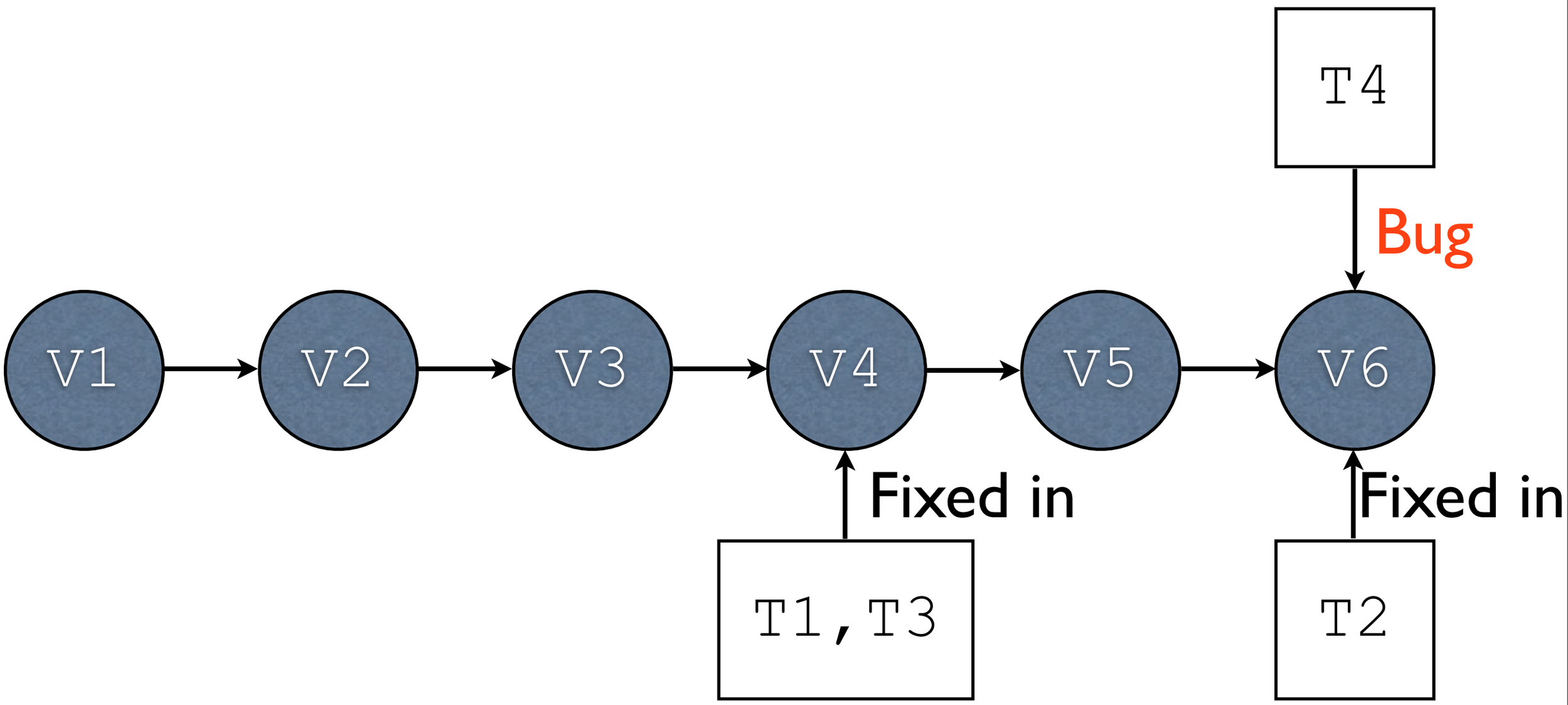
We could run out of versions on some inputs...

Approach



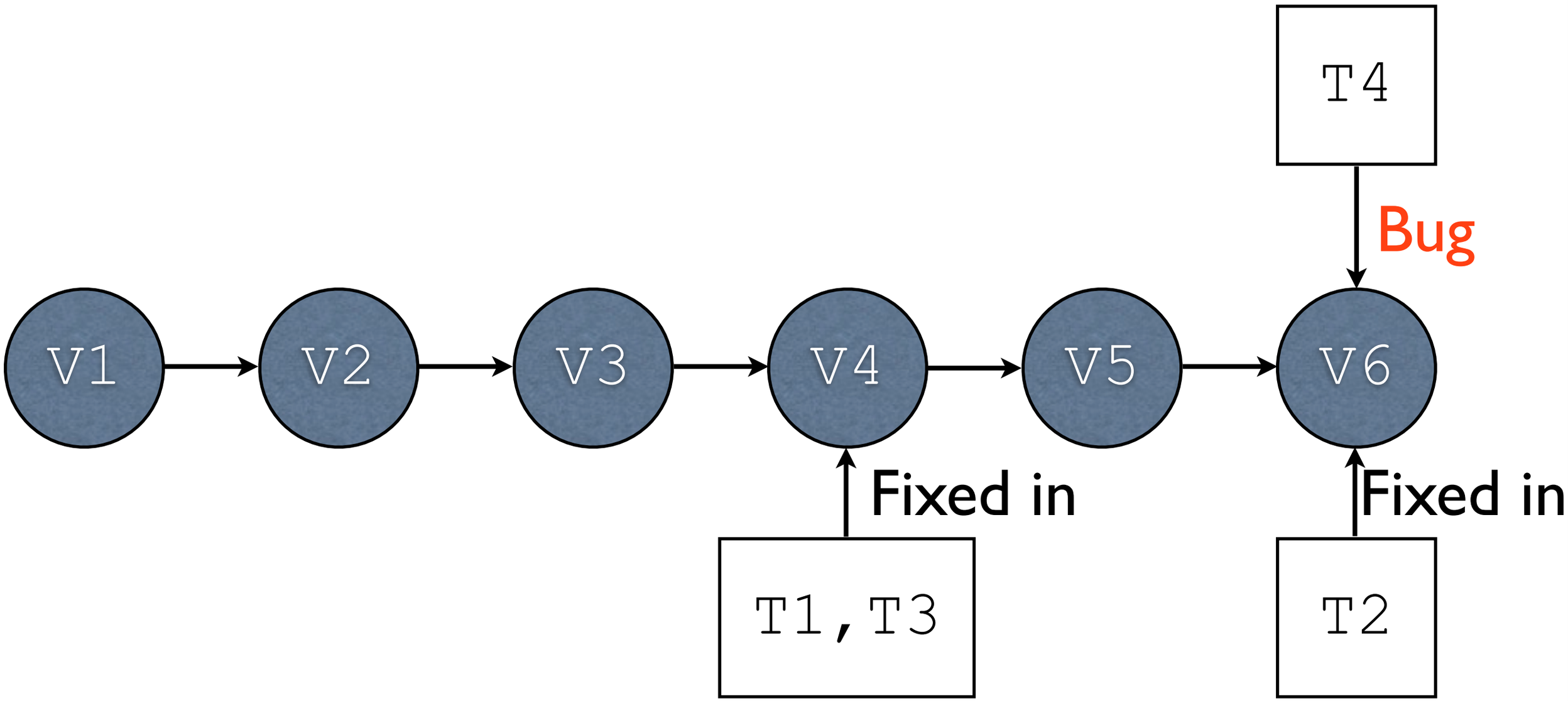
We could run out of versions on some inputs...

Approach



We could run out of versions on some inputs...

Approach



By the assumption that past bugs behave like new bugs, this is not significant.

New Bugs

- Can still report these bugs using existing approaches
    - Prompts developers to create new versions which fix the bugs
- Requires manual effort, but **only** if the evaluator wants to drop the assumption that new bugs behave like old ones, and then only for select bugs
- Existing approach is pure manual effort

Outline

- Background
- Metrics used in the literature
- An automated approach
- How metrics compare
- Conclusion

Experimental Goal

- We have a technique for easily measuring unique bugs
- Want to see how unique bugs compares to the aforementioned surrogate metrics
  - First time any such comparison has been performed
  - Can answer whether or not these surrogate metrics are worth collecting

Systems Under Test

- Interested in testing SMT solvers
  - Answer queries like $x \leq 7 \wedge y \geq 8$
    - Vitally important in software verification, among many others
- Solvers can be buggy, and buggy solvers mean invalid proofs
- Looked at Z3, CVC4, MathSAT5, and Boolector

Experimental Setup

- Developed four pairs of fuzzers for SMT-LIB
  - Each of the two fuzzers in a pair was radically different, but both attempted to test the same part of SMT-LIB
  - Whole pairs of fuzzers are not directly comparable to each other (each pair tests a distinct subset of SMT-LIB)

Experimental Setup

- Developed four pairs of fuzzers for SMT-LIB
  - Each of the two fuzzers in a pair was radically different, but both attempted to test the same part of SMT-LIB
  - Whole pairs of fuzzers are not directly comparable to each other (each pair tests a distinct subset of SMT-LIB)

Pair 1	Pair 2	Pair 3	Pair 4
F1 F2	F3 F4	F5 F6	F7 F8
(e.g., integers)	(e.g., BV)	(e.g., FP)	(e.g., SAT)

Measurement

- Fuzzers within a pair are directly comparable to each other
- For each metric, see which fuzzer in the pair is better
- Question: do surrogate metrics agree with the metric of unique bugs found?

Pair 1	Pair 2	Pair 3	Pair 4
F1 | F2	F3 | F4	F5 | F6	F7 | F8
(e.g., integers)	(e.g., BV)	(e.g., FP)	(e.g., SAT)

	Unique Bugs	Crash Bugs	Bug-Inducing Inputs
F1	2	0	231,254
F2	2	1	17,744
F3	6	2	260,085
F4	1	0	2,800
F5	9	3	20,796
F6	6	2	73,382
F7	12	2	11,917
F8	3	0	374

	Unique Bugs	Crash Bugs	Bug-Inducing Inputs
F1	2	0	231,254
F2	2	1	17,744
F3	6	2	260,085
F4	1	0	2,800
F5	9	3	20,796
F6	6	2	73,382
F7	12	2	11,917
F8	3	0	374

	Unique Bugs	Crash Bugs	Bug-Inducing Inputs
F1	2	0	231,254
F2	2	1	17,744
F3	6	2	260,085
F4	1	0	2,800
F5	9	3	20,796
F6	6	2	73,382
F7	12	2	11,917
F8	3	0	374

	Unique Bugs	Crash Bugs	Bug-Inducing Inputs
F1	2	0	231,254
F2	2	1	17,744
F3	6	2	260,085
F4	1	0	2,800
F5	9	3	20,796
F6	6	2	73,382
F7	12	2	11,917
F8	3	0	374

Only in 2/4 cases did all three metrics agree

	Unique Bugs	Crash Bugs	Bug-Inducing Inputs
F1	2	0	231,254
F2	2	1	17,744
F3	6	2	260,085
F4	1	0	2,800
F5	9	3	20,796
F6	6	2	73,382
F7	12	2	11,917
F8	3	0	374

Crash bugs usually agreed with unique bugs...

	Unique Bugs	Crash Bugs	Bug-Inducing Inputs
F1	2	0	231,254
F2	2	1	17,744
F3	6	2	260,085
F4	1	0	2,800
F5	9	3	20,796
F6	6	2	73,382
F7	12	2	11,917
F8	3	0	374

...though not entirely (and we know they don't in general)

	Unique Bugs	Crash Bugs	Bug-Inducing Inputs
F1	2	0	231,254
F2	2	1	17,744
F3	6	2	260,085
F4	1	0	2,800
F5	9	3	20,796
F6	6	2	73,382
F7	12	2	11,917
F8	3	0	374

Bug-inducing inputs does not correlate to bugs found.

Data Elided

- We looked at five other metrics, based on either bugs found or bug-triggering inputs
- Data for these metrics is very similar to that previously shown

Take-Home Points

- Take-home point #1: metrics based on actual bugs found tend to be consistent with each other (read: useful for comparison)
- Take-home point #2: metrics based on bug-triggering inputs look completely random (read: useless for comparison)

Take-Home Points

- Take-home point #1: metrics based on actual bugs found tend to be consistent with each other (read: useful for comparison)
- Take-home point #2: metrics based on bug-triggering inputs look completely random (read: useless for comparison)

Take-home point #3: SMT solvers are broken (24 bugs found; each solver had correctness bugs, including Z3).

Outline

- Background
- Metrics used in the literature
- An automated approach
- How metrics compare
- Conclusion

Conclusion

- Bug-based metrics are useful for comparison, and our automated evaluation technique makes these much easier to gather
- Our evaluation shows that metrics based on bug-inducing inputs are meaningless
    - Concerning, considering that **most existing work** uses these metrics
- SMT solvers tested are now a little less buggy (22 bugs fixed across all solvers tested)