Great Works in Invariant/Specification Mining

Presented by Ayaka Yorihiro
Mentored by Eric Campbell
GreatPL 04/27/2021

Some reasons we want formal specs...

- Understanding programs
- Documentation
- Generating Tests
- Formal/Runtime Verification

Why would we want to autogenerate them?

Some reasons for autogeneration...

- People don't write specs!
  - Time & Effort
  - Need to maintain both code and specs
- But Formal/Runtime Verification needs specs to work
- Unexpected helpful invariants

Quickly Detecting Relevant Program Invariants

Ernst, Czeisler, Griswold, Notkin

University of Washington Technical Report 1999

Problem Space and Contributions

Goal: Find program invariants to help programmers understand code!

Challenges:

- Reported too many invariants
  - Many of these were not useful!
- Performance Issues

Problem Space and Contributions

Goal: Find program invariants to help programmers understand code!

Contributions:

- Four approaches to increase relevance of mined invariants
  (1) Implication
  (2) Comparability
  (3) Polymorphism Elimination
  (4) Repeated Values

Daikon (Dynamic Invariant Inference)

```
// Return the sum of the elements of
//      array b, which has length n.
long array_sum(int * b, long n) {
    long s = 0;
    for (int i=0; i<n; i++)
        s = s + b[i];
    return s;
}
```

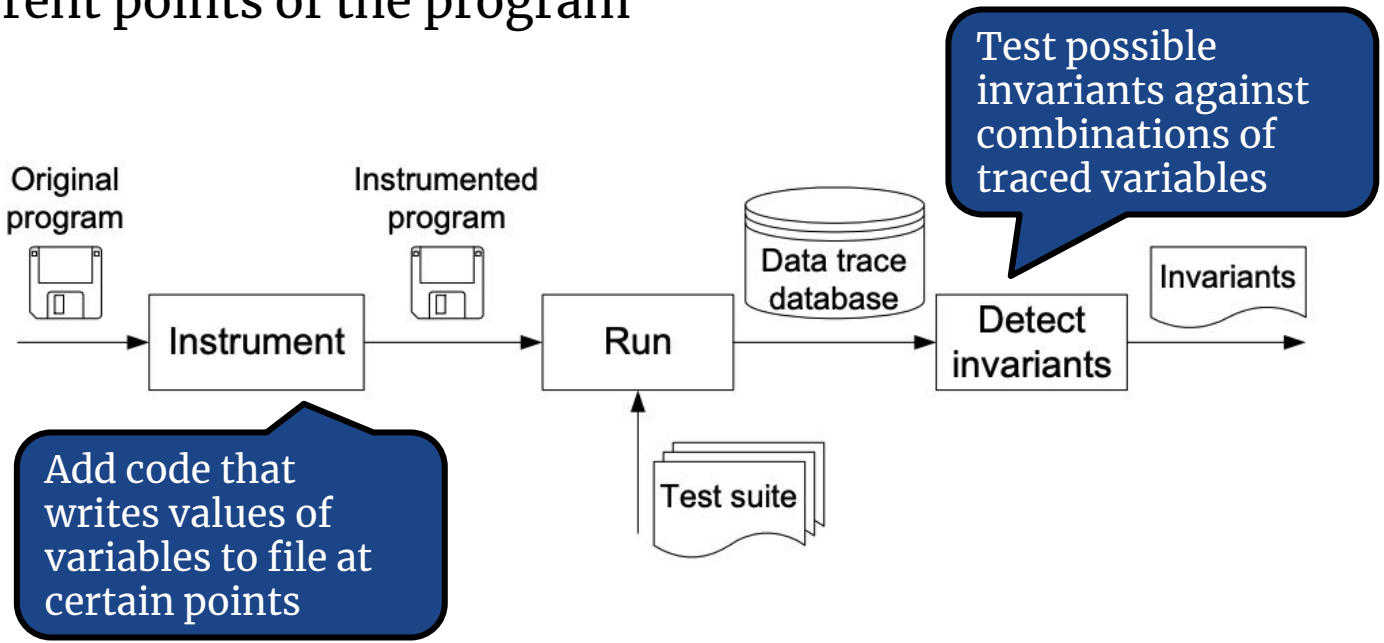
A sample program returning sum of elements of array b

```
15.1.1:::ENTER  100 samples // Invariants at beginning of function
  N = size(B)                              (24 values)
  N >= 0                                   (24 values)

15.1.1:::EXIT    100 samples // Invariants at end of function
  B = B_orig                               (96 values)
  N = I = N_orig = size(B)                 (24 values)
  S = sum(B)                               (95 values)
  N >= 0                                   (24 values)

15.1.1:::LOOP   986 samples // Invariants at start of loop
  N = size(B)                              (24 values)
  S = sum(B[0..I-1])                       (95 values)
  N >= 0                                   (24 values)
  I >= 0                                   (36 values)
  I <= N                                   (363 values)
```

Invariants mined from ←

Daikon (Dynamic Invariant Inference)

- Test set of possible invariants against the values from variables at different points of the program



Invariant Detection

- Templates of equations from First Order Logic
  - Single Variables
    - Any Variables (ex. constant)
    - Single Numeric Variables (ex. range, never zero)
  - Multiple Numeric Variables (ex. linear relationship, ordering comparison)
  - Sequence Variables (ex. ordering, min/max values)

- Reported invariants: those that were tested a sufficient degree without falsification

Implication

- Invariants logically implied by other invariants are redundant!

- ex) “x in [7..13]” ⇒ “x ≠ 0”
  - The second one is redundant!

- BIG improvement:
  - Disabling Implication optimizations makes system run out of memory!

Comparability

- Compare only variables that can be sensibly compared!

Methods of Comparability

```
// Return the sum of the elements of
//      array b, which has length n.
long array_sum(int * b, long n) {
    long s = 0;
    for (int i=0; i<n; i++)
        s = s + b[i];
    return s;
}
```

- Two vars are comparable...

Unconstrained:
...by default

Source Program Types:
...iff declared to have the same type

Coerced Program Types:
...if their types are coercible to each other

Lackwit Types (Polymorphic Type Inference):
...if they participate in an expression

Polymorphism Elimination

- Daikon wants to infer invariants over polymorphic variables! But implementation doesn't allow for this...
  - Data trace format (involving declared types) are statically determined during instrumentation
  - Can't directly find invariants over polymorphic variables where exact type and data fields are not known until runtime

```
class ListNode { Object element;
                 ListNode next; ... }
```

Can't infer invariants on the runtime type of element!

Two-pass technique for Polymorphism

- Original:
```
class ListNode { Object element;
                 ListNode next; ... }
```

- Pass 1: Daikon finds invariant over the Polymorphic class and learns what runtime types it can have

- (If always one type) User annotates declaration with comments
```
class ListNode { /*refined_type: MyInteger*/ Object element;
                 ListNode next; ... }
```
- Pass 2: Daikon reads comments and treats variables as having specified types

Repeated Values

- Daikon does a statistical confidence test to avoid properties that could easily have occurred by chance
  - We can avoid overweighting of variable values in statistical tests!

Strategies for avoiding overweighting of variable values

Always:
Every sample contributes to confidence

+: Trivial to implement
-: LOTS of undue confidence

Changed Value:
Contributes only when value is different from last program point

-: Doesn't detect recomputing to same value

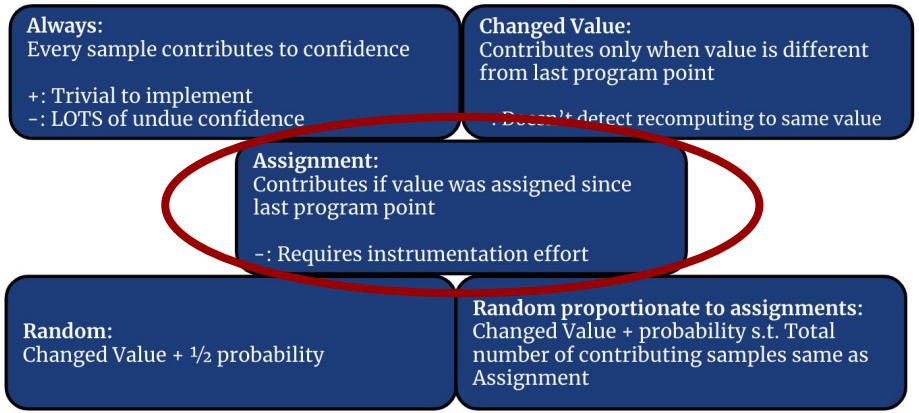
Assignment:
Contributes if value was assigned since last program point

-: Requires instrumentation effort

Random:
Changed Value + ½ probability

Random proportionate to assignments:
Changed Value + probability s.t. Total number of contributing samples same as Assignment

Strategies for avoiding overweighting of variable values



	All	Value	Random	Random $\propto$
Added	63	32	122	31
relevant	6	6	7	6
irrelevant	57	26	115	25
Removed	0	67	71	19
relevant	0	5	1	1
irrelevant	0	26	70	18

Comparison with Assignment on 300 test cases

	All	Value	Random	Random $\propto$
Added	33	23	36	26
relevant	0	4	0	0
irrelevant	33	19	36	26
Removed	10	9	14	14
relevant	6	1	6	6
irrelevant	4	8	8	8

Comparison with Assignment on 1000 test cases

Conclusion on Daikon

- Detect invariants by trying templated First Order Logic equations

- Four approaches to increase relevance of mined invariants
  (1) Implication
  (2) Comparability
  (3) Polymorphism Elimination
  (4) Repeated Values

Leveraging Test Generation and Specification Mining for Automated Bug Detection with False Positives

Pradel & Gross

ICSE 2012

Problem Space & Contributions

Goal: Use Spec Mining for Bug Detection!

Challenges:

- Reliance on program input
  - Dynamic approaches depend on execution of program
  - ...which rely on input
- False Positives
  - LOTS of spurious warnings
  - Makes tools unreliable!

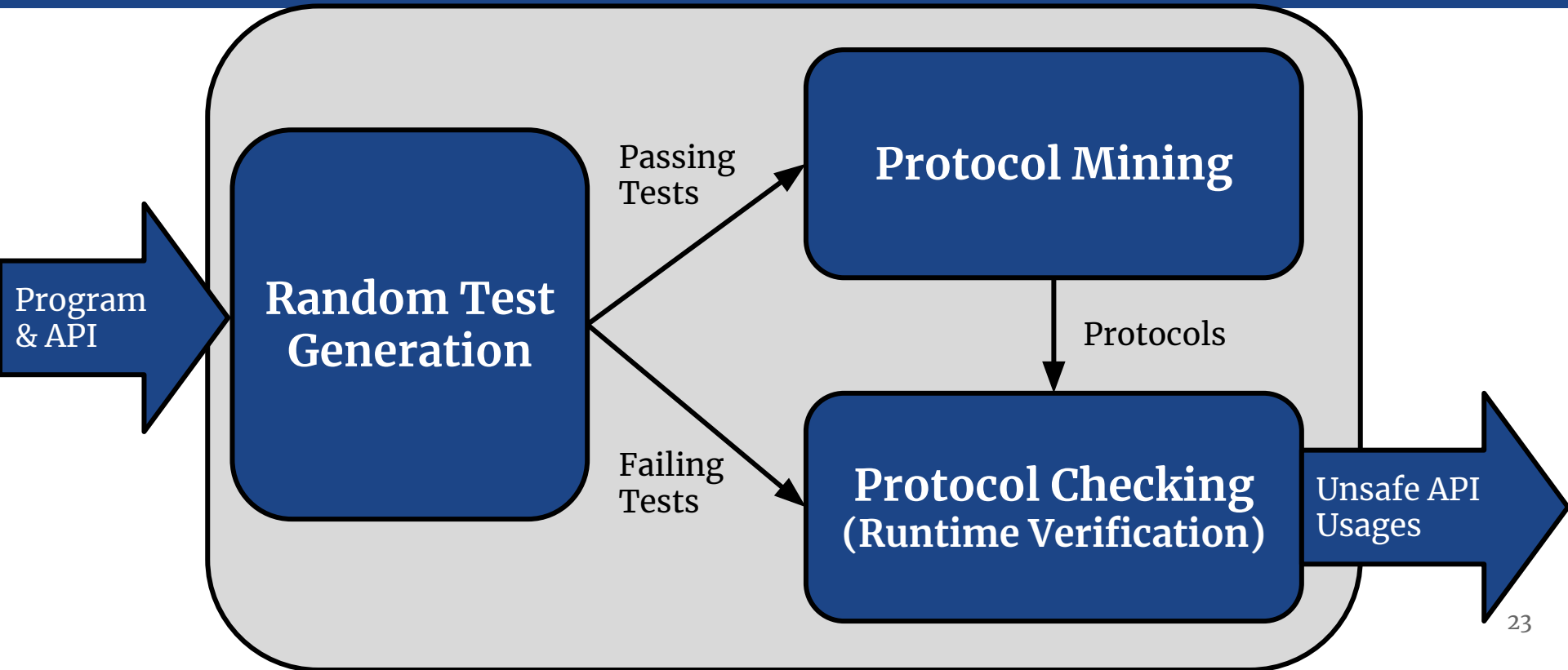
Problem Space & Contributions

Goal: Use Spec Mining for Bug Detection!

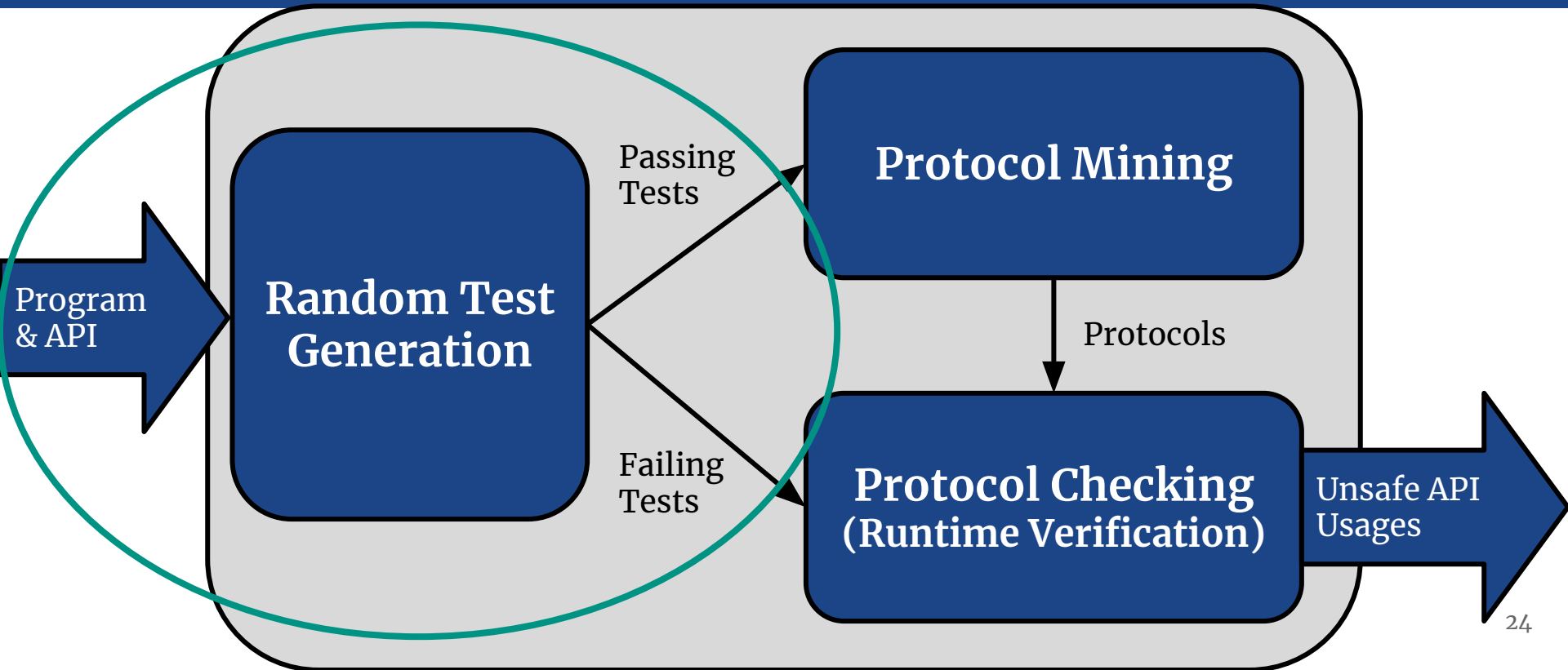
Contributions:

- Use autogenerated tests to drive mining and checking
- Eliminate false positives
- (Guided random test generation)

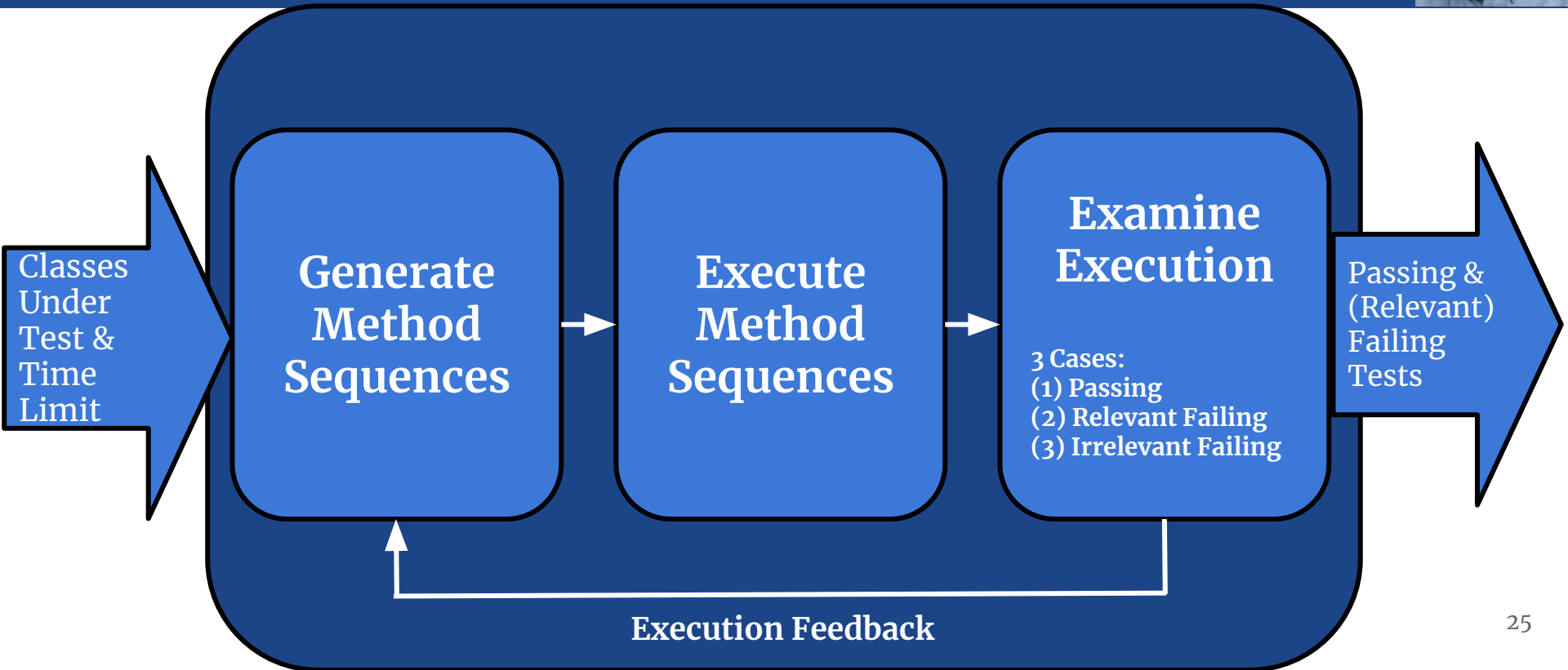
Pipeline


Program & API
Random Test Generation
Passing Tests
Failing Tests
Protocol Mining
Protocols
Protocol Checking (Runtime Verification)
Unsafe API Usages

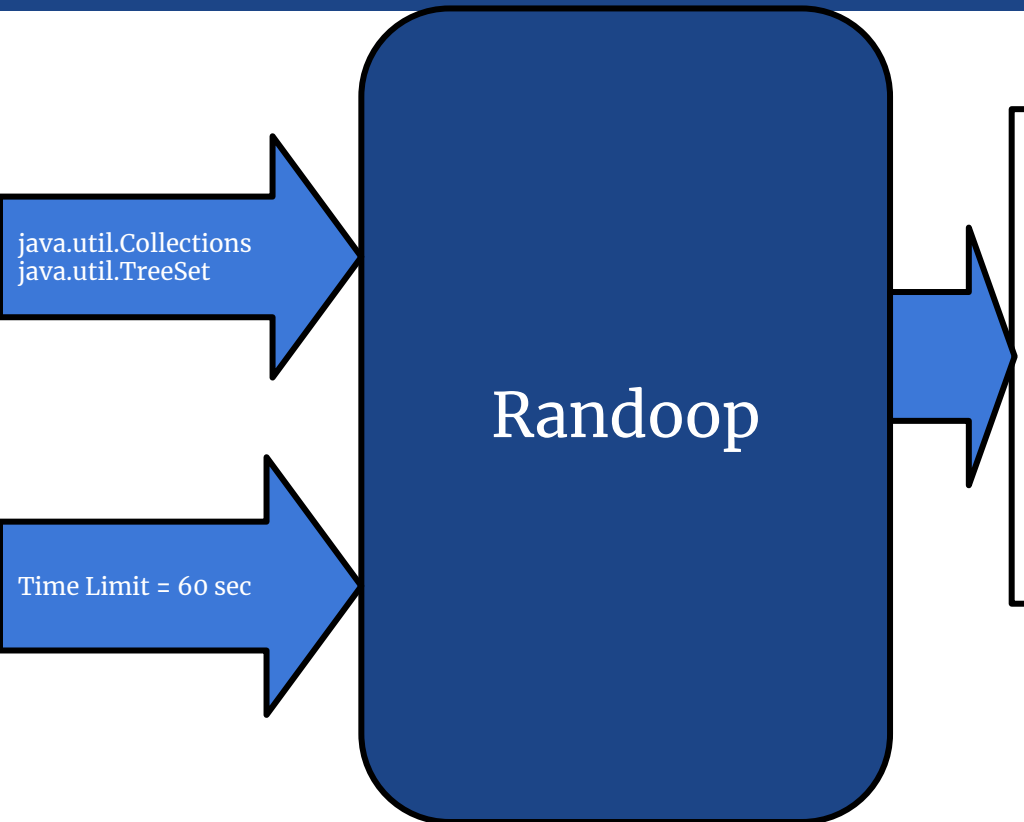
Pipeline


Program
& API
Random Test
Generation
Passing
Tests
Failing
Tests
Protocol Mining
Protocols
Protocol Checking
(Runtime Verification)
Unsafe API
Usages

Randoop: Random Test Generation

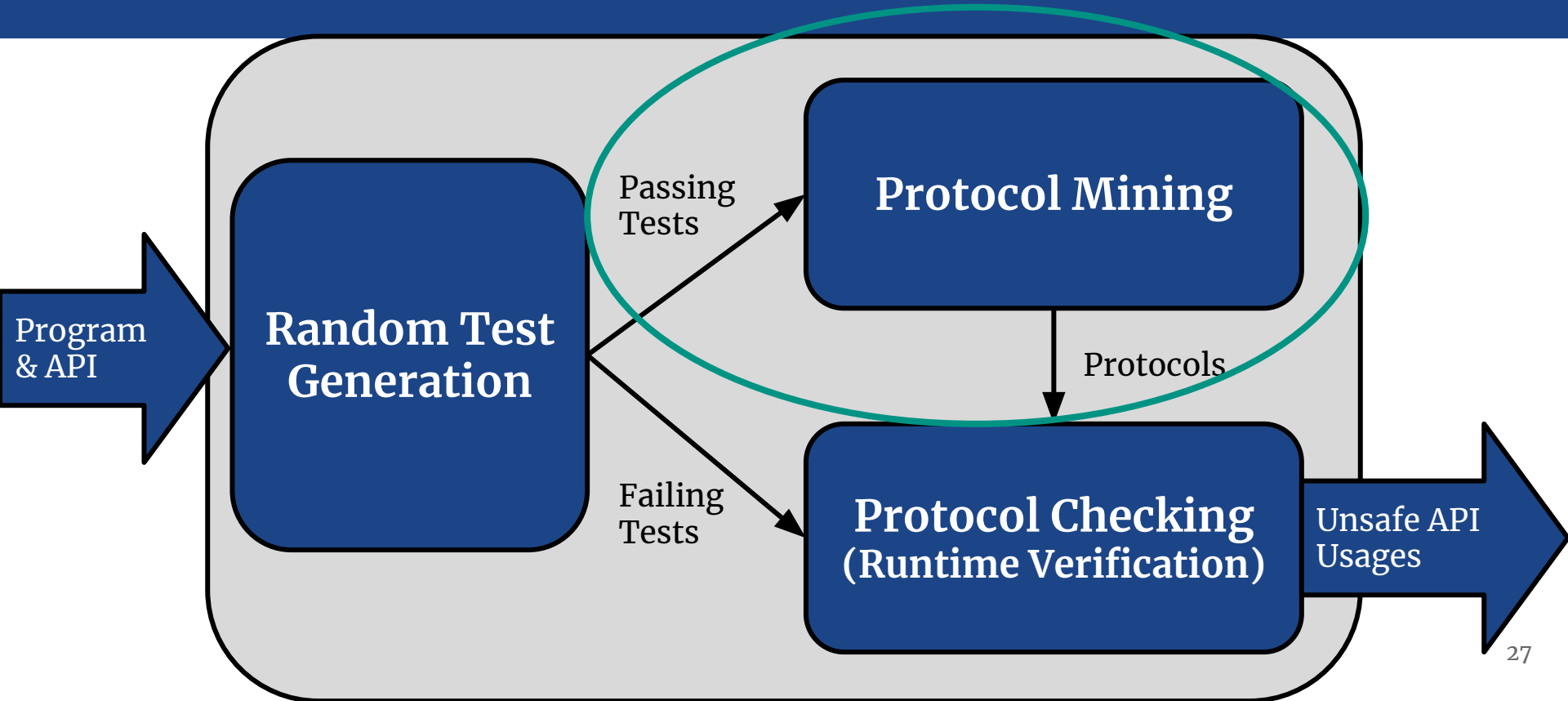


Example Failing Test Generated by Randoop

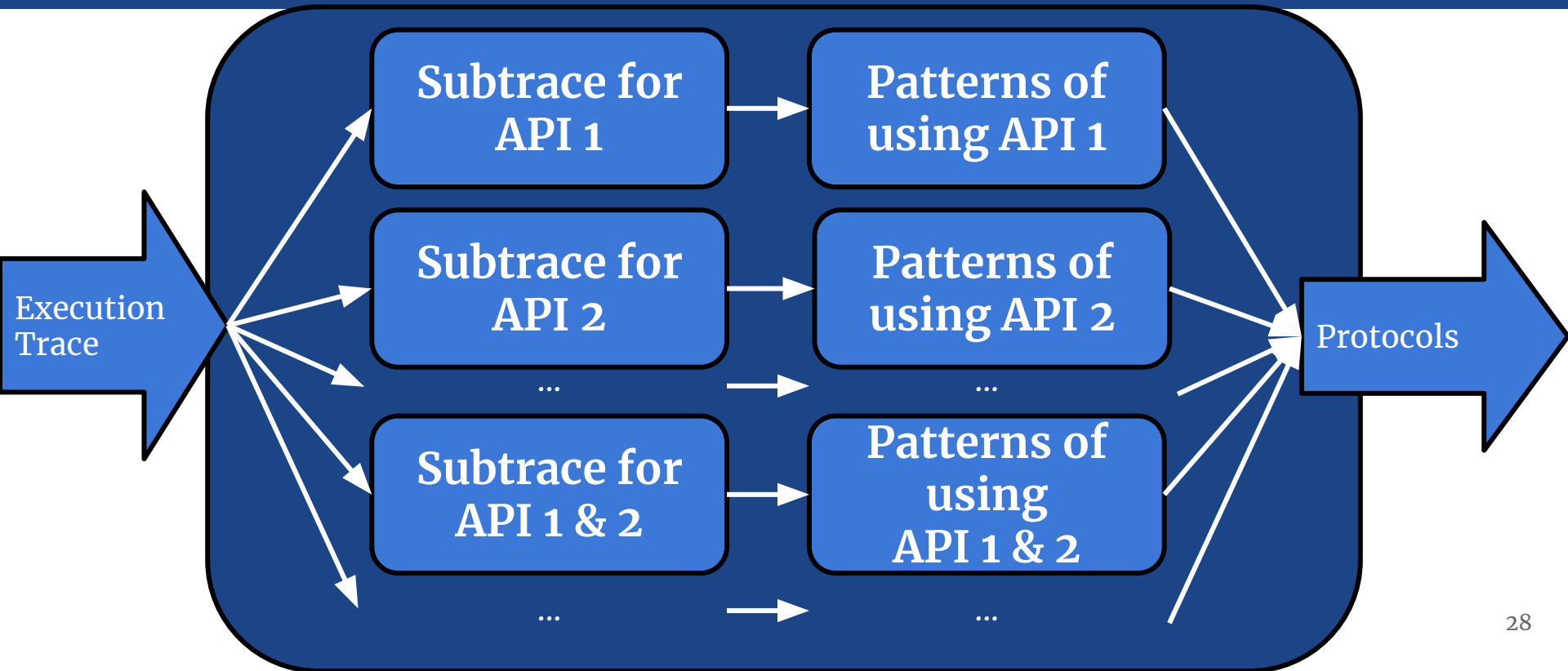



```
// This test shows that the JDK collection classes
// can create an object that is not equal to itself.
@Test
public static void test1() {

  LinkedList list = new LinkedList();
  Object o1 = new Object();
  list.addFirst(o1);

  // A TreeSet is an ordered collection. According to the API
  // documentation, this constructor call should throw a
  // ClassCastException because the list element is not Comparable. But
  // the constructor silently (and problematically) accepts the list.
  TreeSet t1 = new TreeSet(list);

  Set s1 = Collections.synchronizedSet(t1);

  // At this point, we have successfully created a set (s1)
  // that violations reflexivity of equality: it is not equal
  // to itself! This assertion fails at run time on OpenJDK.
  org.junit.Assert.assertEquals(s1, s1);
}
```

Pipeline


Program & API
Random Test Generation
Passing Tests
Failing Tests
Protocol Mining
Protocols
Protocol Checking (Runtime Verification)
Unsafe API Usages

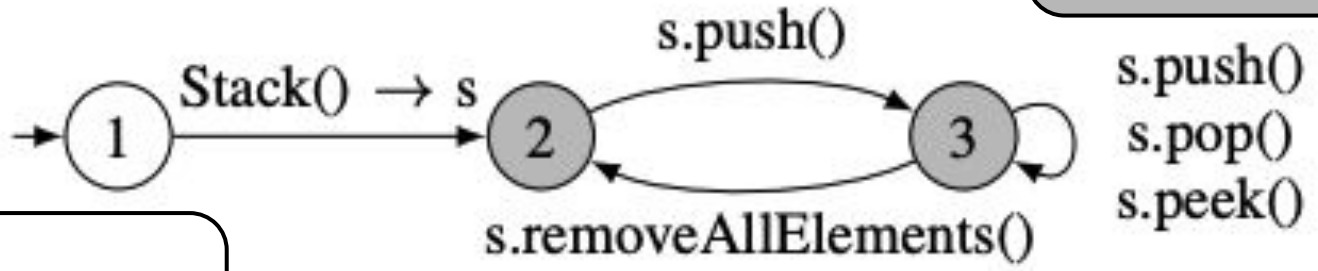
Mining Protocols from Passing Tests



Protocols

- FSMs describing ordering of API methods

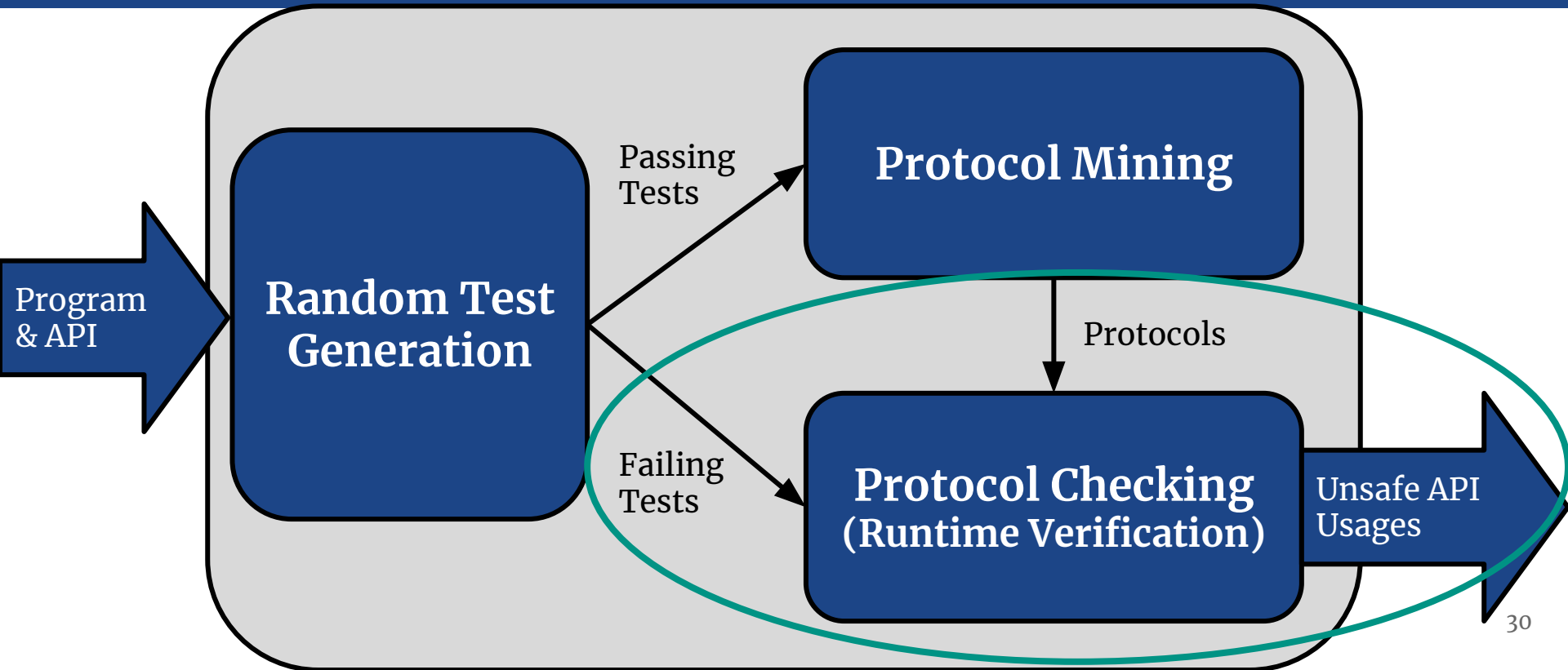
Liable States:
Describe constraints to respect



Mined Protocol for Stack
(Regular Approx. of push() pop() balance)

Setup State:
Bind objects to protocol parameters

Pipeline


Program & API
Random Test Generation
Passing Tests
Failing Tests
Protocol Mining
Protocols
Protocol Checking (Runtime Verification)
Unsafe API Usages

Checking Protocols on Failing Tests

- Bug = Test Fail & Protocol Violation
  - How do we figure out if a Protocol was violated?

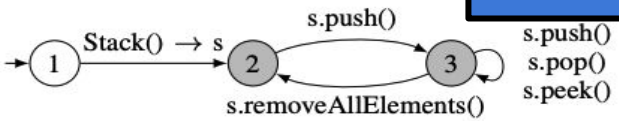
- Runtime Verification (RV)
  - Check relevant program method calls against specifications

Runtime Verification Example

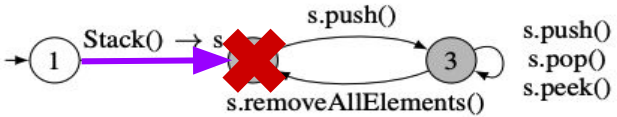
```
class Main {
    Stack s = new Stack();
    s.peek();
}
```

Failing Test

Protocol




```
class Main {
    Stack s = new Stack();
    s.peek();
}
```



Reported Protocol Violation

Eliminating False Positives

- Report only violations that **certainly** lead to exception thrown by API

- Check 1: Want to focus on exceptions that were thrown by the API in question
  - → Check that Type of Exception is declared in API

- Check 2: Want to make sure that the protocol violation was involved in the exception
  - → Check that Violating API call occurs in stack trace

Eliminating False Positives

- Case 3:

```
class BrowserHistory {
    private Stack history = new Stack();
    public String peekPrevious() throws EmptyStackException {
        return get().toString();
    }
    private Object get() throws EmptyStackException {
        history.peek();
    }
    public void fill() {
        history.push(...);
    }
}
```

Not unsafe...

Class that checks most recent site visited on browser

peek

```
public E peek()
```

Looks at the object at the top of this stack without removing it from the stack.

Returns:

the object at the top of this stack (the last item of the `Vector` object).

Throws:

`EmptyStackException` - if this stack is empty.

peek() declaration in Stack API

```
new Stack() -> history
# EmptyStackException
# at Stack.peek()
# at BrowserHistory.get()
# at BrowserHistory.peekPrevious()
```

Execution Trace

We need to check that the type of Exception is not declared to be thrown in the program itself!

Eliminating False Positives

- Filter - a violation is only reported when:
  1. Type of Exception declared in API
  2. Violating API call occurs in stack trace
  3. Type of Exception is not declared to be thrown in program

Example of Violation…

```
class BrowserHistory {
    private Stack history = new Stack();
    public String peekPrevious() {
        return get().toString();
    }
    private Object get() {
        history.peek();
    }
    public void fill() {
        history.push(...);
    }
}
```

Filter - a violation is only reported when:

1. Type of Exception declared in API
2. Violating API call occurs in stack trace
3. Type of Exception is not declared to be thrown in program

peek

`public E peek()`

Looks at the object at the top of this stack without removing it from the stack.

Returns:

the object at the top of this stack (the last item of the `Vector` object).

Throws:

`EmptyStackException` if this stack is empty.

peek() declaration in Stack API

```
new Stack() -> history
history.peek()
# EmptyStackException
# at Stack.peek()
# at BrowserHistory.get()
# at BrowserHistory.report()
```

Execution Trace

Recap and Conclusion

- Daikon (1999)
  - Established methodology for mining invariants
  - Propose four methods of increasing relevance of reported invariants

- ICSE 2012
  - Describes bug finding as an application of protocol mining
  - Use autogenerated tests to drive mining and checking
  - Propose filter to eliminate false positives

Thank you!!