CSE467: Computer Security

23. Static Analysis

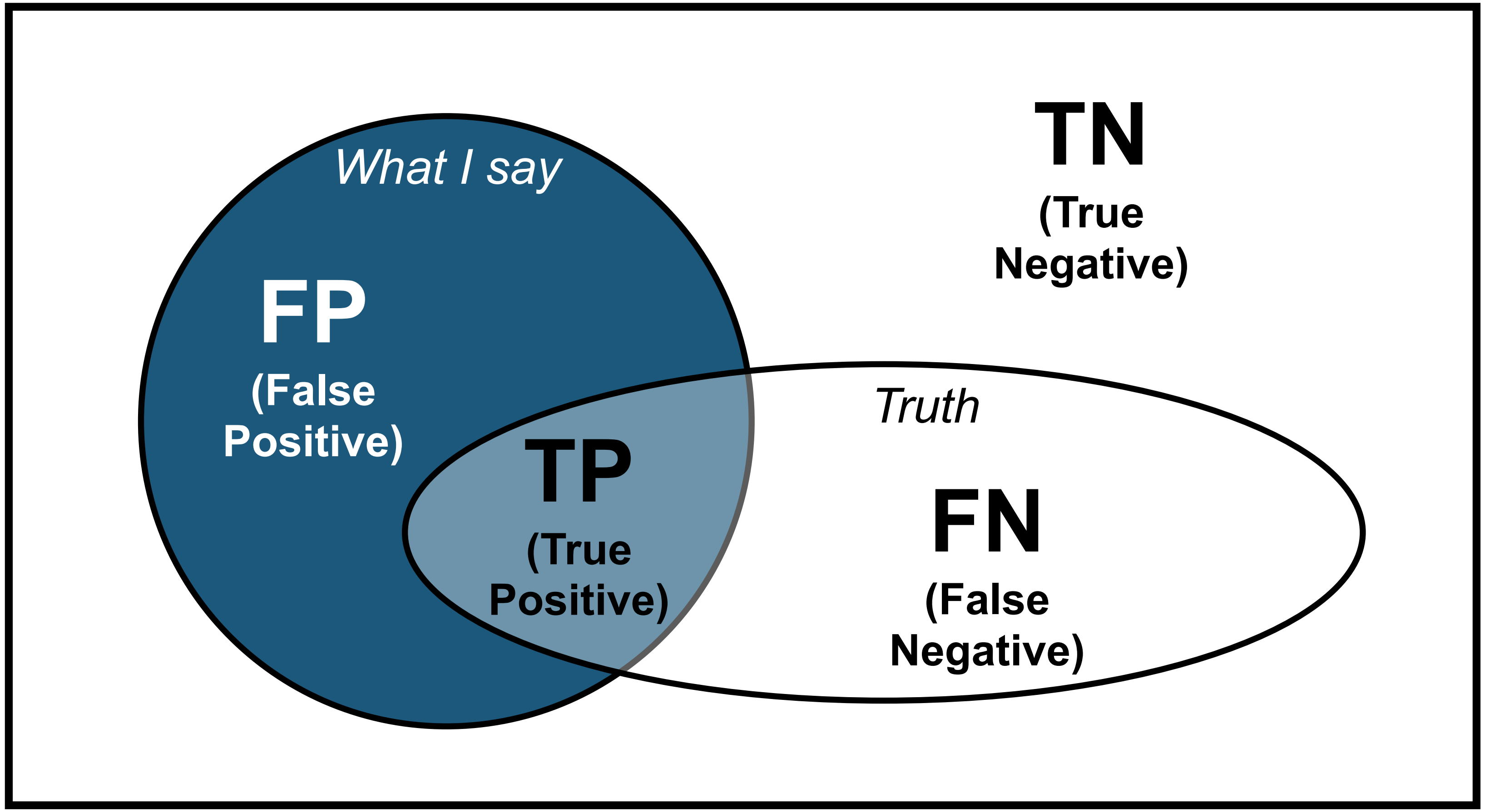
Seongil Wi

Department of Computer Science and Engineering

Class Overview

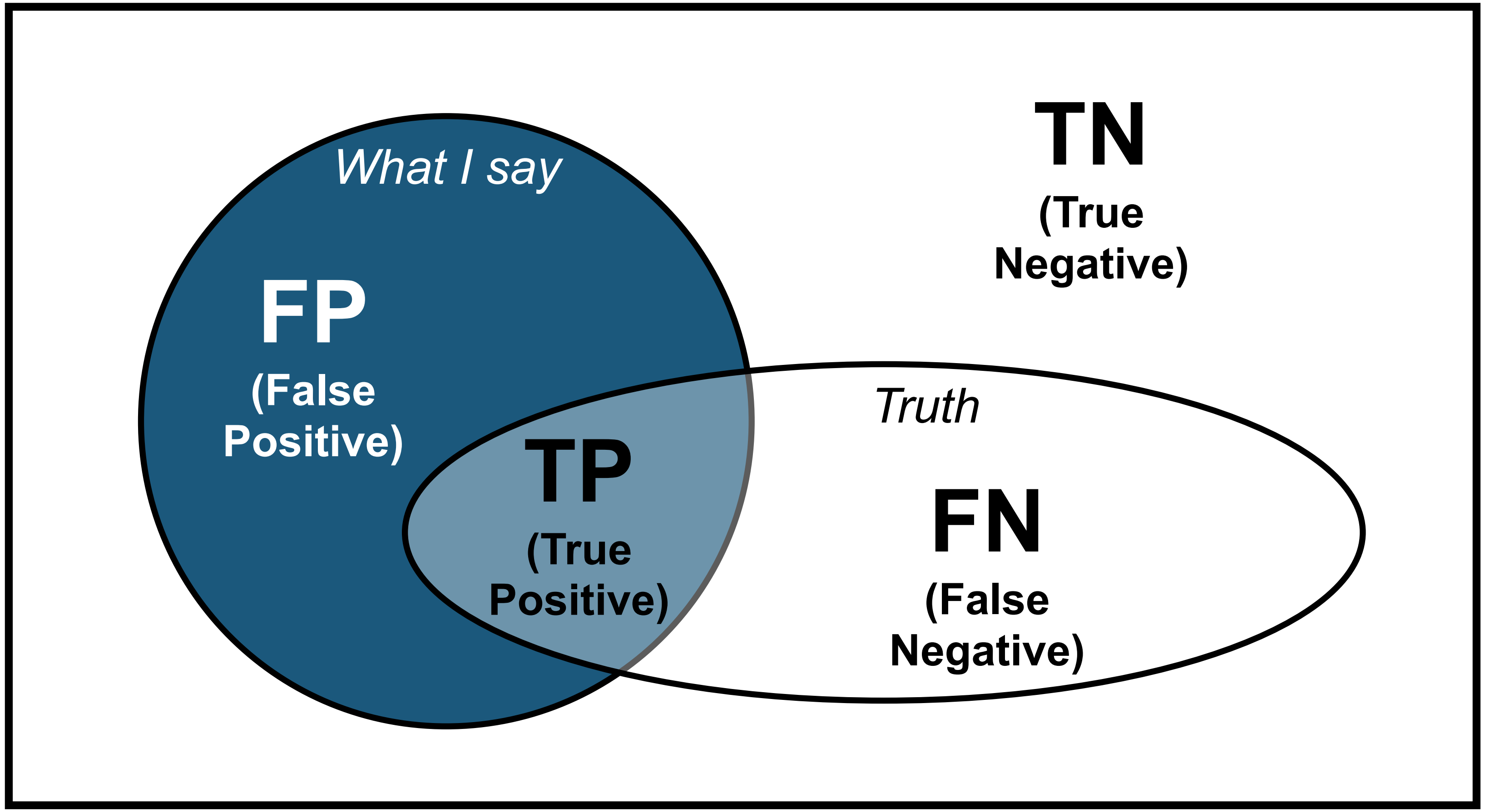
- 30~40 mins: Lecture
- 25 mins: Quiz
- 10 mins: Sharing solution

Recap: Precision, Recall, and Accuracy



- Precision
  = TP / (TP + FP)
- Recall
  = TP / (FN + TP)
- Accuracy
  = (TP+TN)/
  (TP + FP + FN + TN)

Recal: False Positive Rate vs. False Negative Rate



- FP Rate
  = FP / (TP + FP)

- FN Rate
  = FN / (FN + TN)

Recap: Three Forms of Testing

- **Manual testing**
  - A human test the code
- **Static analysis**
  - Analyze the program without executing it
- **Dynamic analysis**
  - Analyze the program during an execution

Today’s Topic!

- **Manual testing**
  - A human test the code
- **Static analysis**
  - Analyze the program without executing it
- **Dynamic analysis**
  - Analyze the program during an execution

Static Analysis

- Analyze the program ***without executing it*** to detect potential security bugs
- *Abstract (over-approximate)* across ***all possible executions***

- Keywords: (static) taint analysis, (static) symbolic execution, abstract interpretation, abstract syntax tree, control flow graph, data flow graph

Symbolic Execution

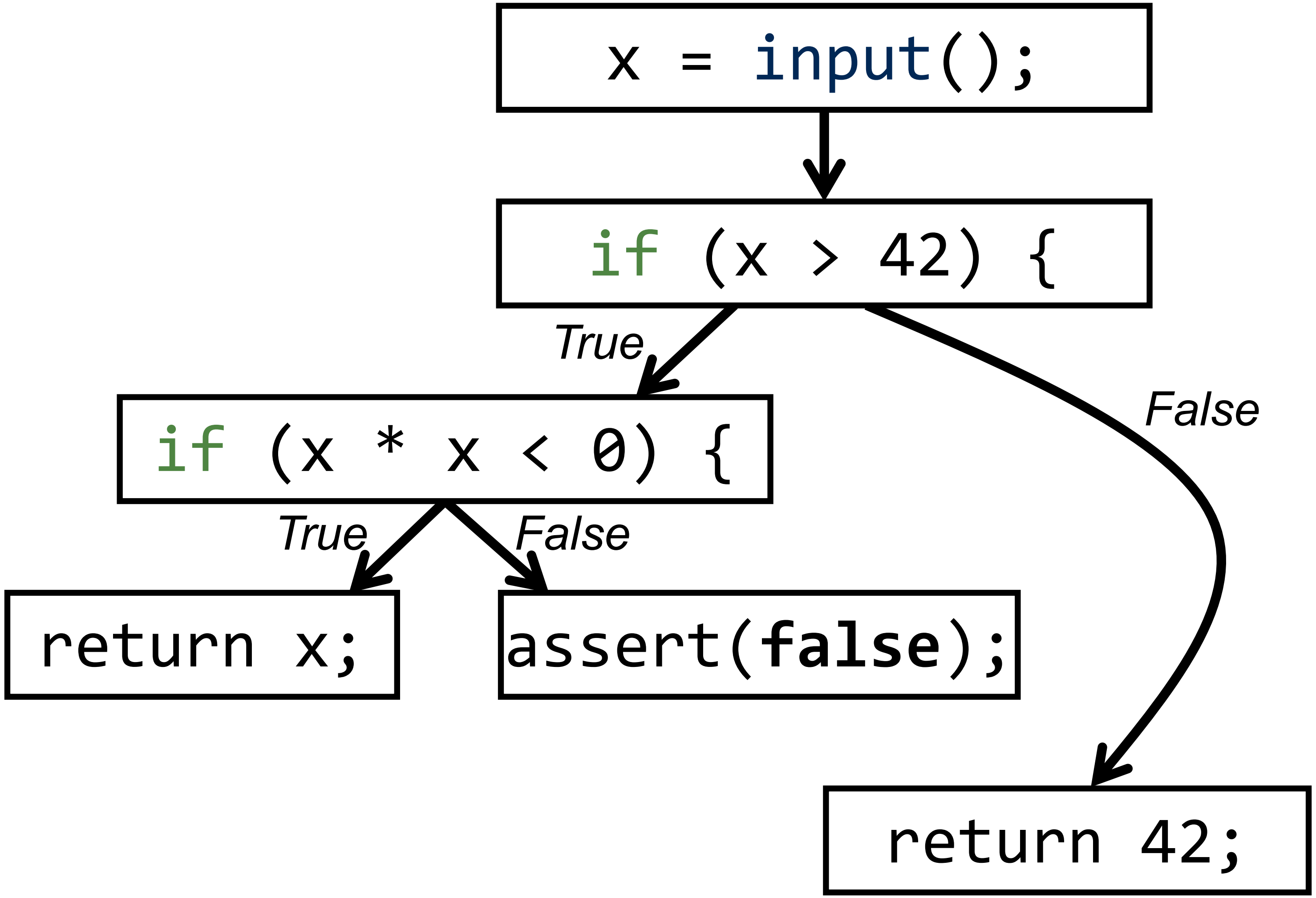
Symbolic Execution

- A program analysis technique that executes a program with symbolic – rather than concrete – input values.

- For each execution path, construct a ***path formula*** that describes the input constraint to follow the path

Example Scenario

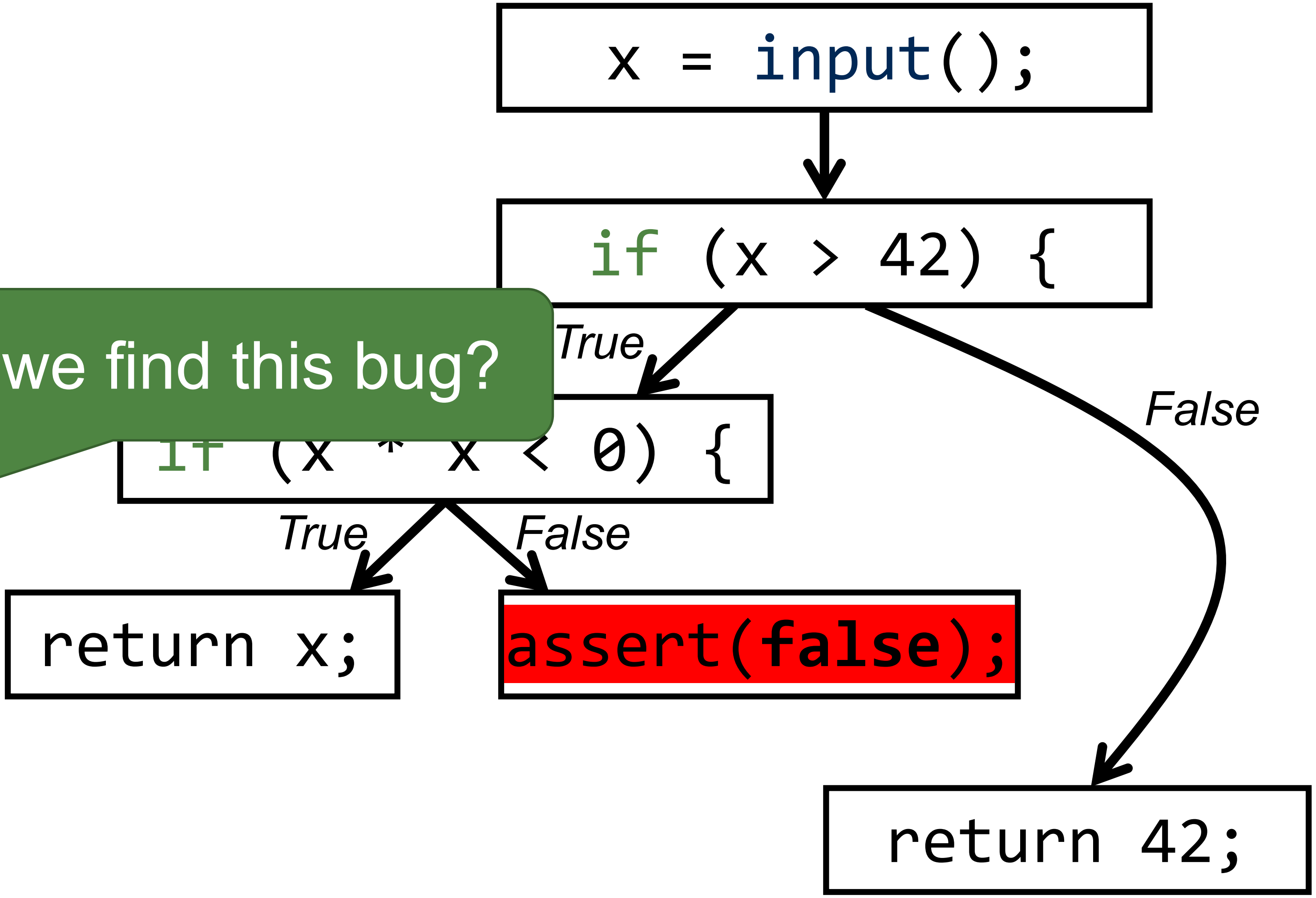
```
x = input();
if (x > 42) {
    if (x * x < 0)
        return x;
    else {
        assert(false);
    }
} else {
    return 42;
}
```



Example Scenario

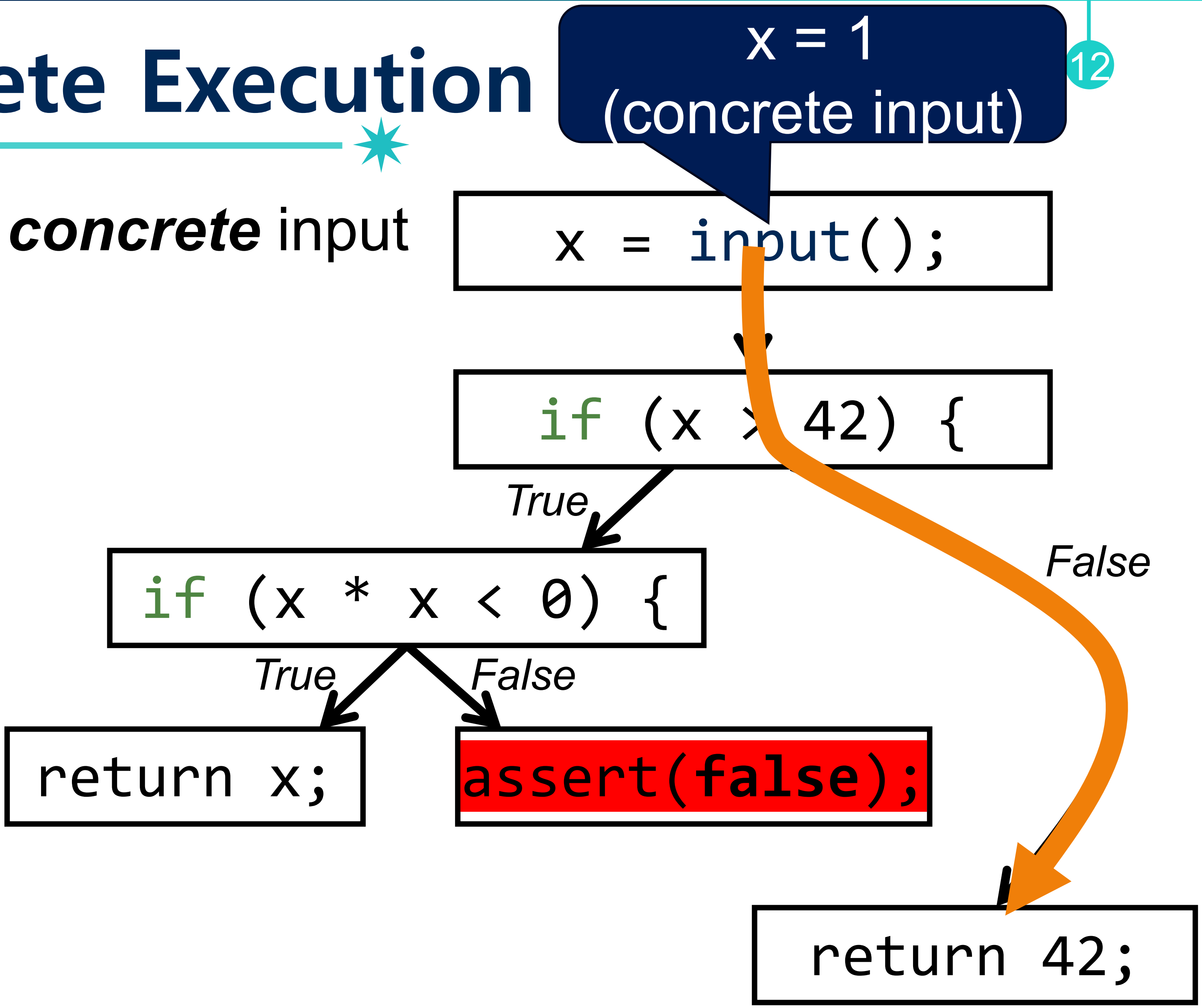
```
x = input();
if (x > 42) {
    if (x * x < 0)
        return x;
    else {
        assert(false);
    }
} else {
    return 42;
}
```

How we find this bug?



Example: Concrete Execution

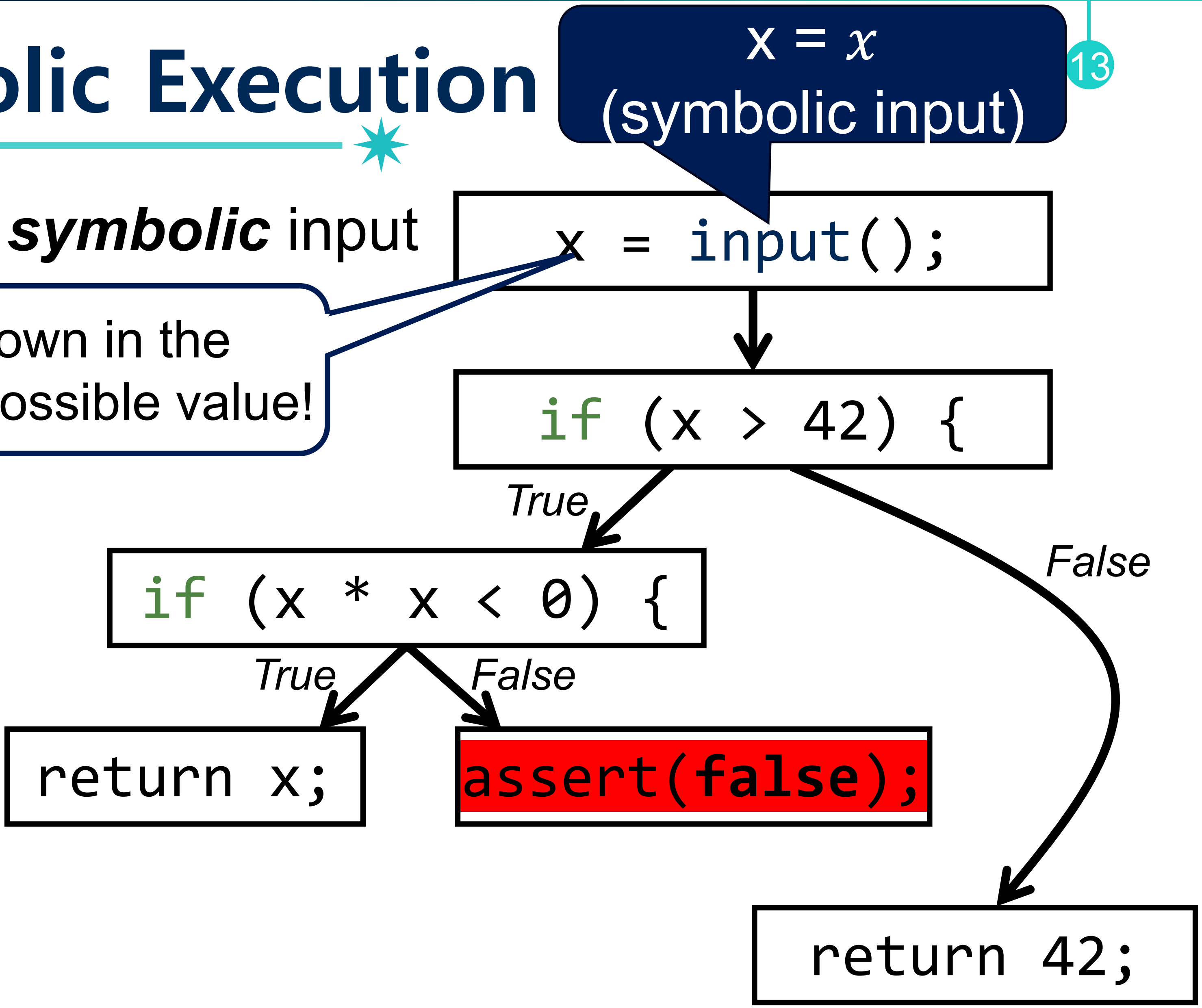
- Runs a program with a ***concrete*** input



Example: Symbolic Execution

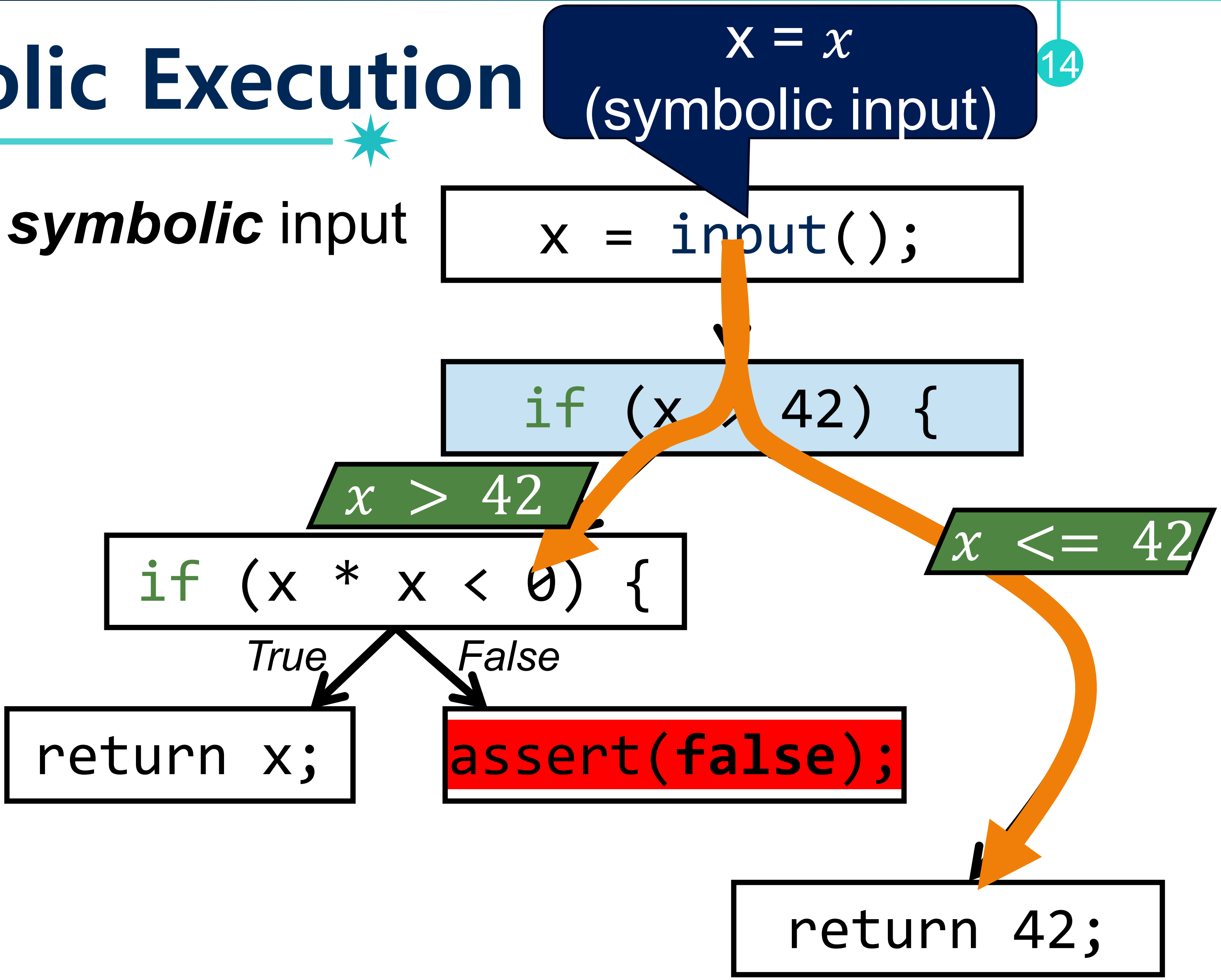
- Runs a program with a ***symbolic*** input

Let's treat it as an unknown in the equation. It can have any possible value!



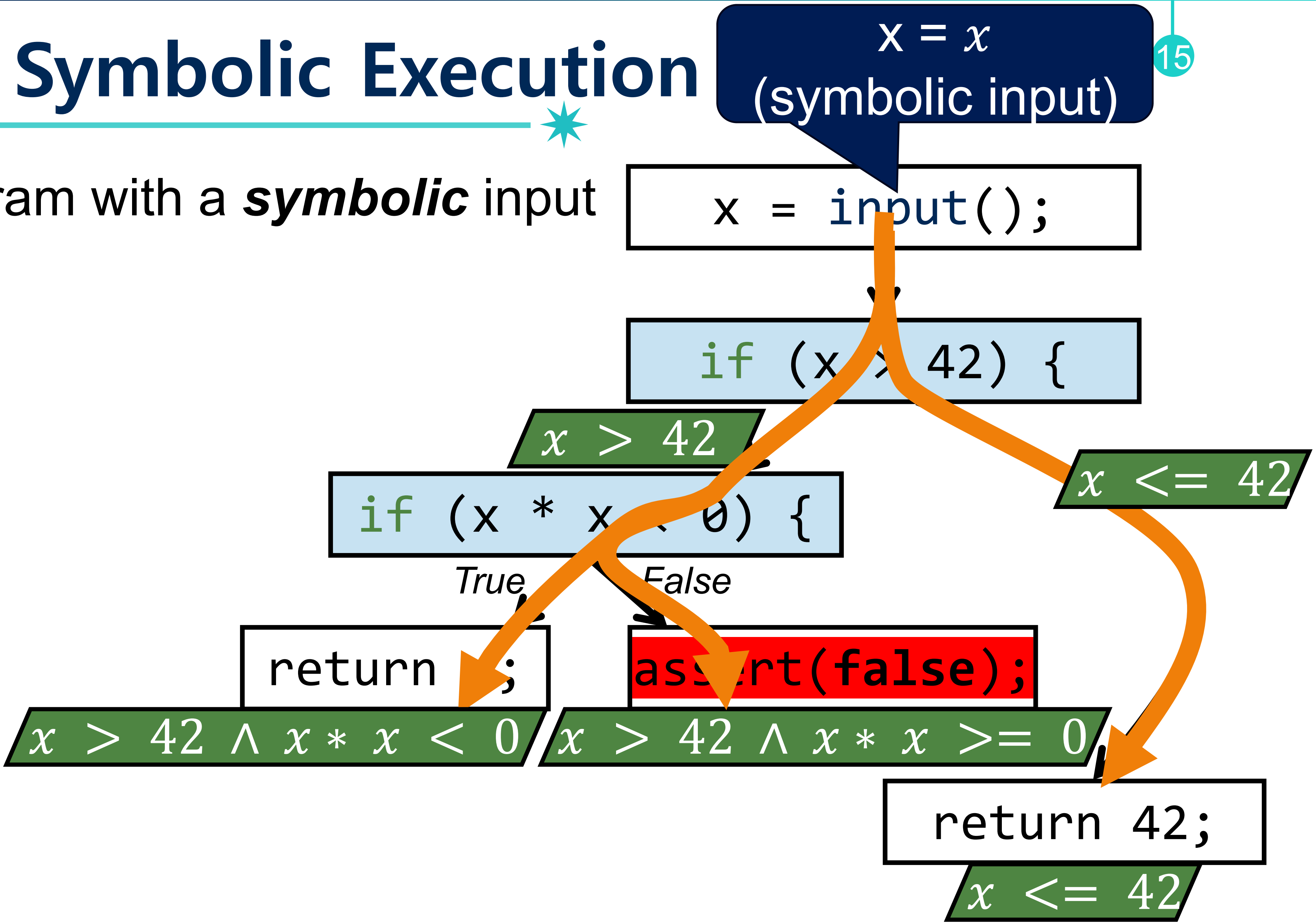
Example: Symbolic Execution

- Runs a program with a ***symbolic*** input



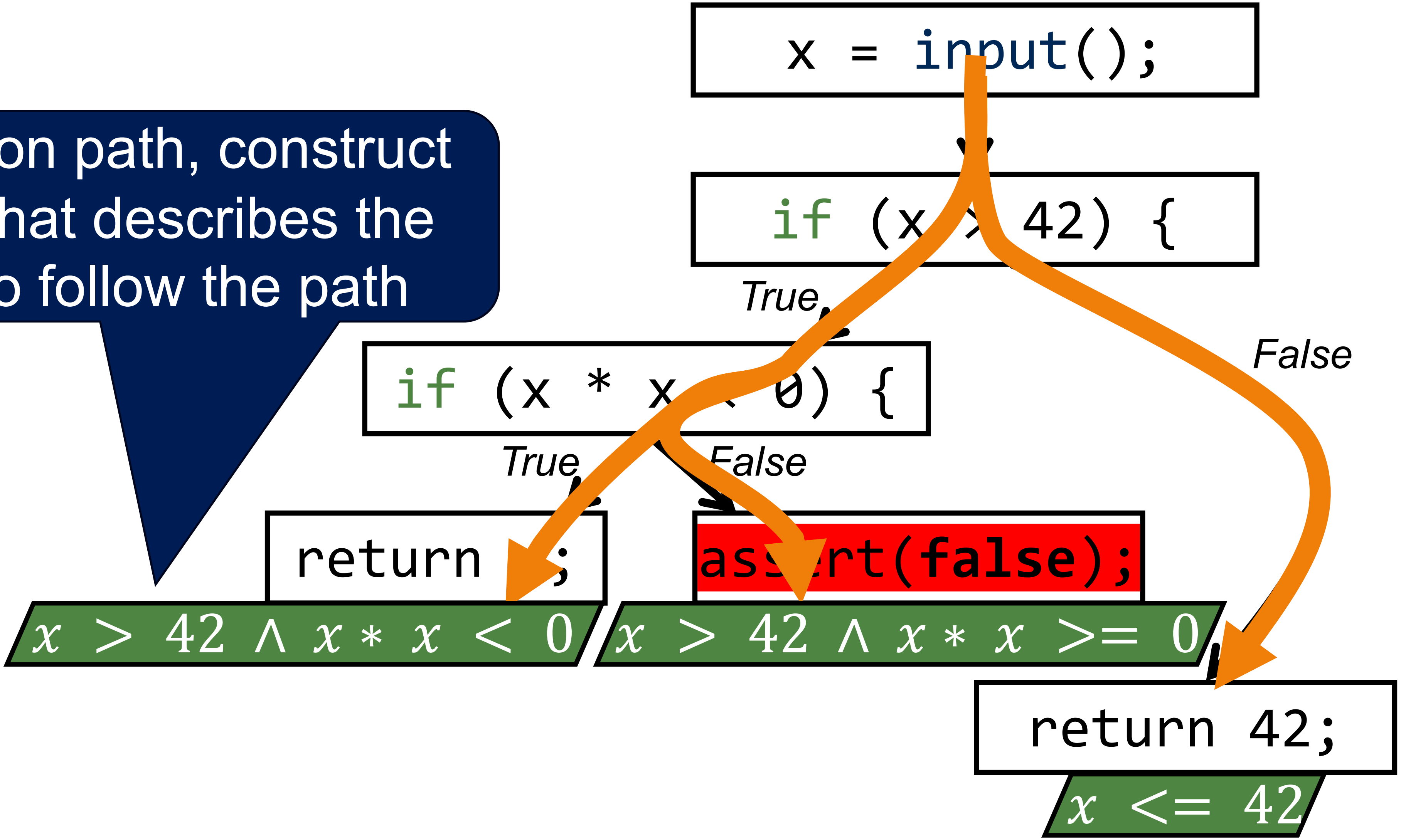
Example: Symbolic Execution

- Runs a program with a ***symbolic*** input

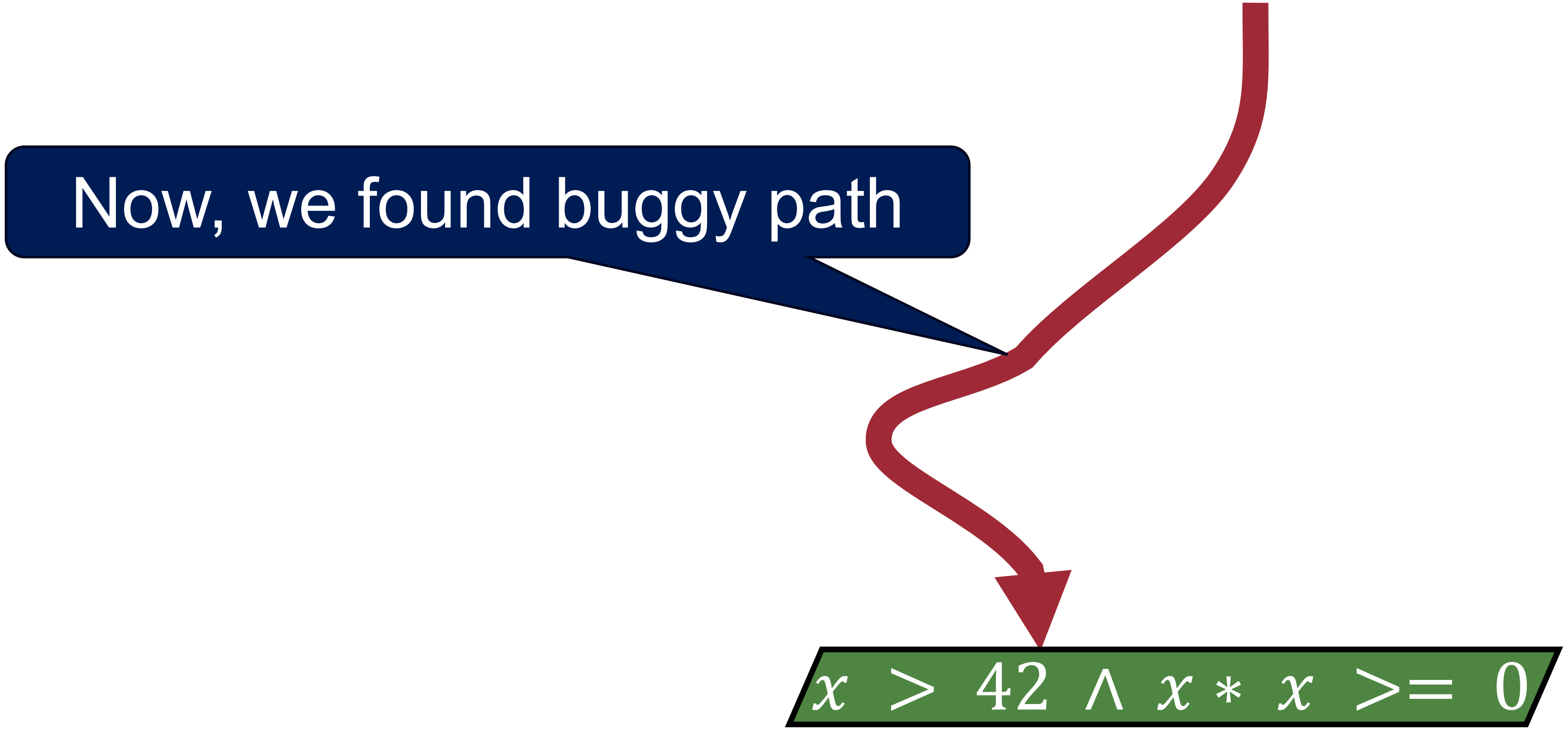


Path Formulas

For each execution path, construct a ***path formula*** that describes the input constraint to follow the path



Example: Symbolic Execution (Revisited)



How to generate a concrete test case to explore this path?

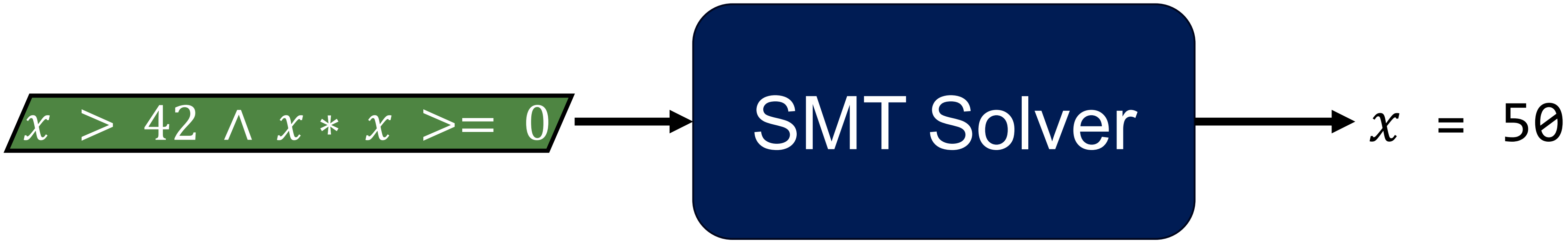
Generate Input to Explore a Path

Key idea: finding a solution from a path formula

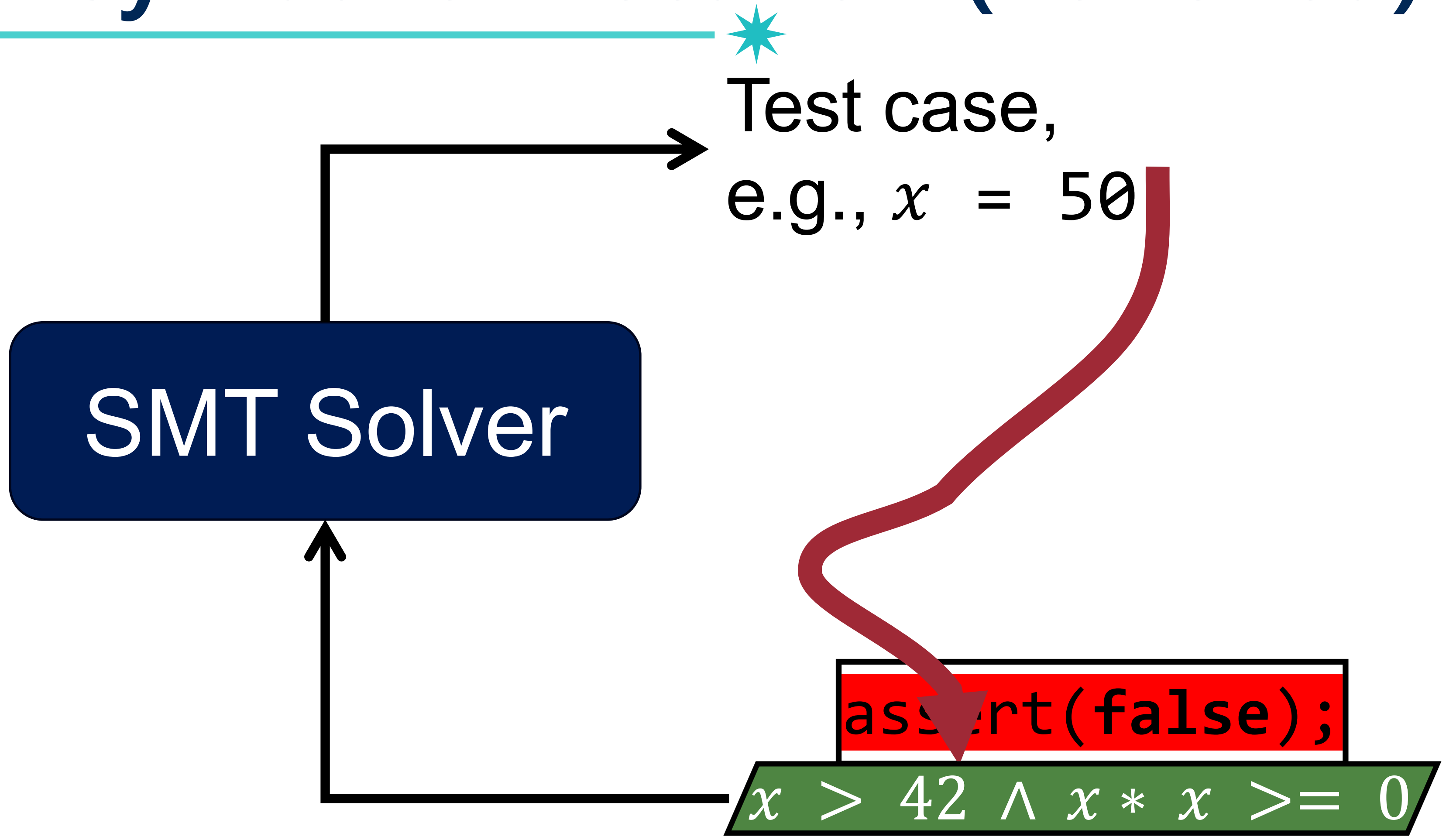
$x > 42 \land x * x >= 0$

How do We Solve a Path Formula?

- Leverage constraint Satisfiability Modulo Theory (SMT) Solver
  - Compute satisfying answers from a given formula
  - E.g., Z3, Boolector



Example: Symbolic Execution (Revisited)



Symbolic Execution

- A program analysis technique that executes a program with symbolic – rather than concrete – input values.

- Popular for finding software bugs and vulnerabilities:
  - e.g., In Microsoft, 30% of bugs are discovered by symbolic execution
  - Symbolic execution is the key technique used in DARPA Cyber Grand Challenge

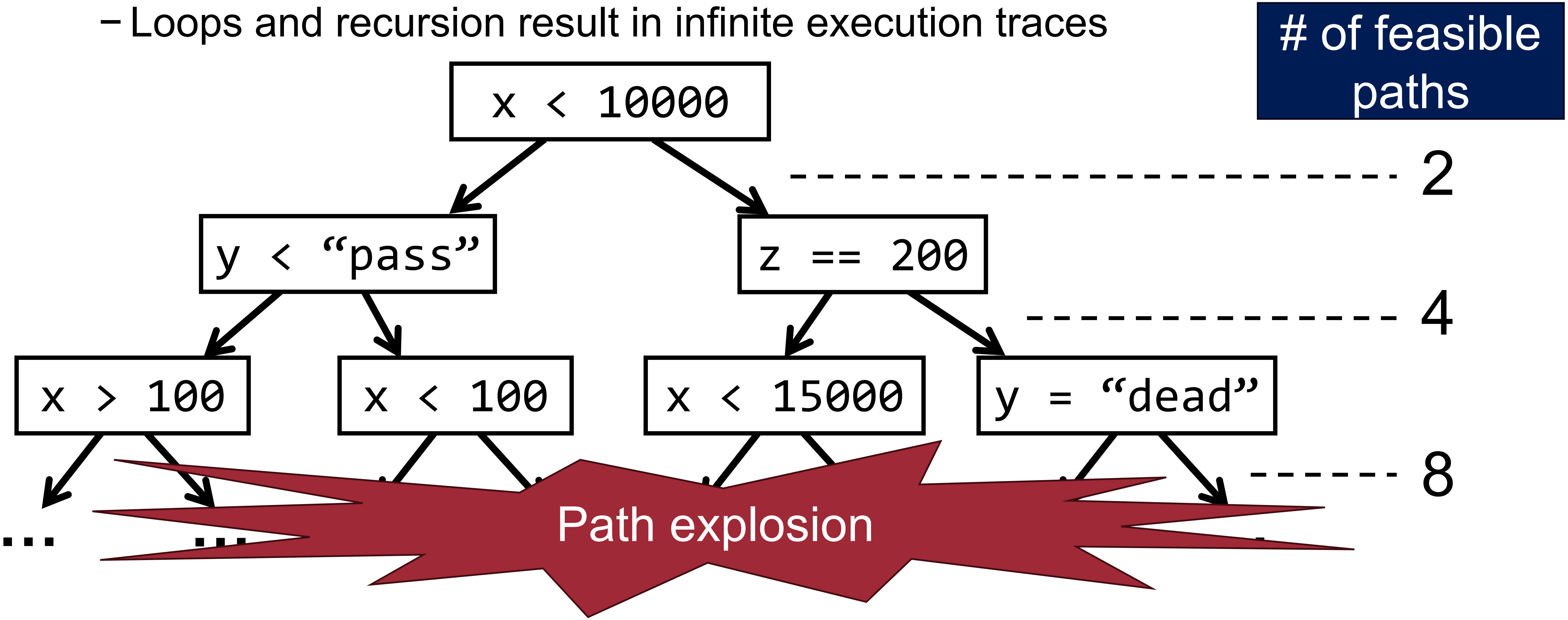
- Symbolic execution tools:
  - Stanford: KLEE
  - NASA: PathFinder
  - Microsoft: SAGE
  - UC Berkeley: CUTE
  - …

Symbolic Execution: Pros and Cons

- Pros
  - High coverage: we can symbolically execute all possible paths
- Cons
  - Main problem: **Path explosion**

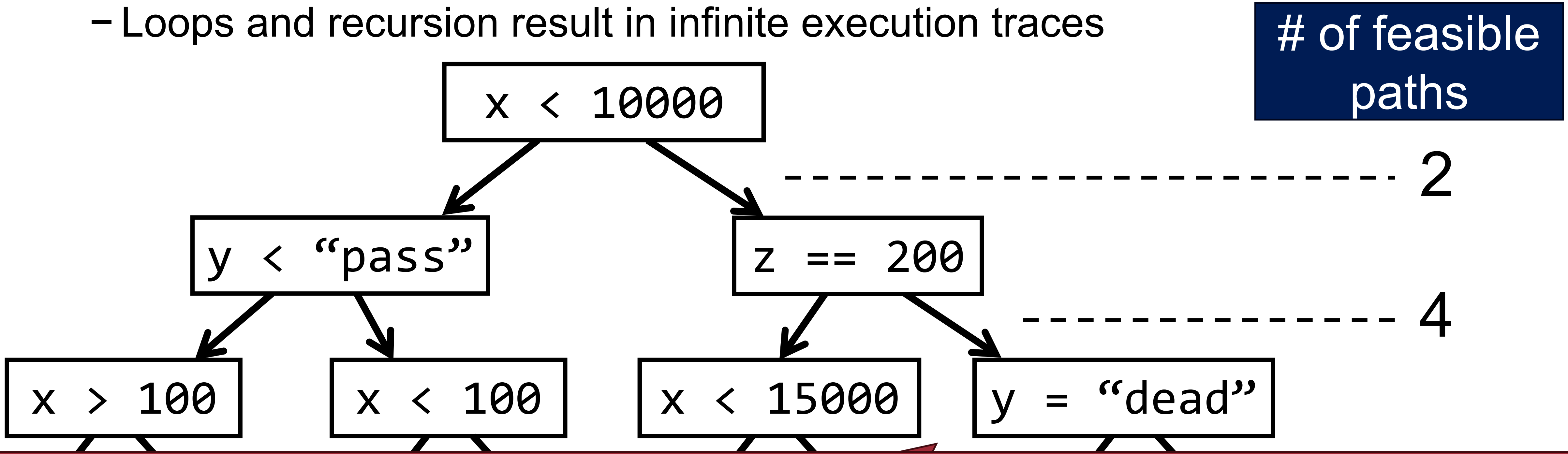
Path Explosion

- The number of feasible paths ***grows exponentially***
  - Each condition doubles the number of paths
  - Loops and recursion result in infinite execution traces



Path Explosion

- The number of feasible paths ***grows exponentially***
  - Each condition doubles the number of paths
  - Loops and recursion result in infinite execution traces



Symbolically executing all program paths does not scale to large programs

Symbolic Execution: Pros and Cons

- Pros
  - Maximizing code coverage: we can check all possible paths

- Cons
  - Main problem: **path explosion**
  - Environment modeling (system calls are complex)
    - E.g., `fd = open(filename)`
  - The process of solving constraints can be computationally expensive

Concolic (Concrete+Symbolic) Execution

- Also called Dynamic Symbolic Execution (DSE)
- Program is simultaneously executed with concrete and symbolic inputs

- Implementations
  - SAGE
  - CREST
  - Angr
  - Triton

- We will cover it in the next class

Taint Analysis

Taint Analysis

- More abstract than Symbolic Execution
- Basic idea: identify whether “tainted” values can ***reach*** “sensitive” points in the program
  - Tainted source: input values that come from the user
  - Sensitive sink: any point in the program where a value is

Sources and Sinks: Example in Web Security

- Sources:
  - `$_GET`, `$_POST`, `$_REQUEST`, `$_SERVER`, `$_FILES`, …

- Sinks:
  - SQL Injection: `mysql_query`, `pg_query`, …
  - File Inclusion: `include`, `require`, …
  - XSS: `echo`, `print`, …

Example Code: SQL Injection Vulnerability


```
<?php
  $id = $_POST['id'];
  $id2 = $id;
  $query = "SELECT * FROM users WHERE id='$id2'";
  $r = mysql_query($query);
?>
```

Taint Analysis Procedure

1. Identify source:
where you get a user input value

```
<?php
  $id = $_POST['id'];
  $id2 = $id;
  $query = "SELECT * FROM users WHERE id='$id2'";
  $r = mysql_query($query);
?>
```

Taint Analysis Procedure

```
<?php
  $id = $_POST['id'];
  $id2 = $id;
  $query = "SELECT * FROM
  $r = mysql_query($query);
?>
```

1. Identify source:
where you get a user input value

3. Build data flows
from source to sink

2. Identify sink:
where a query is fired

Build Data Flows From Source to Sink

```
$id:      Untainted
$id2:     Untainted
$query:   Untainted
```

Source

```
$id = $_POST['id'];
```

```
$id2 = $id;
```

```
$query = "SELECT * FROM users WHERE id='$id2'";
```

Sink

```
$r = mysql_query($query);
```

Build Data Flows From Source to Sink

```
$id:     Untainted
$id2:    Untainted
$query:  Untainted
```

```
$id:     Tainted
$id2:    Untainted
$query:  Untainted
```

Source `$id = $_POST['id'];`

`$id2 = $id;`

`$query = "SELECT * FROM users WHERE id='$id2'";`

Sink `$r = mysql_query($query);`

Build Data Flows From Source to Sink

$id:	Untainted
$id2:	Untainted
$query:	Untainted

Source
```
$id = $_POST['id'];
```

$id:	Tainted
$id2:	Untainted
$query:	Untainted

```
$id2 = $id;
```

$id:	Tainted
$id2:	Tainted
$query:	Untainted

```
$qu… id='$id2'";
```

Taint propagation:
taint status propagates as data flow

Sink
```
$r = mysql_query($query);
```

Build Data Flows From Source to Sink

$id:	Untainted
$id2:	Untainted
$query:	Untainted

Source `$id = $_POST['id'];`

$id:	Tainted
$id2:	Untainted
$query:	Untainted

`$id2 = $id;`

$id:	Tainted
$id2:	Tainted
$query:	Untainted

`$query = "SELECT * FROM users WHERE id='$id2'";`

$id:	Tainted
$id2:	Tainted
$query:	Tainted

Sink `$r = mysql_query($query);`

Build Data Flows From Source to Sink

$id:	Untainted
$id2:	Untainted
$query:	Untainted

Source
```
$id = $_POST['id'];
```

$id:	Tainted
$id2:	Untainted
$query:	Untainted

```
$id2 = $id;
```

$id:	Tainted
$id2:	Tainted
$query:	Untainted

Vulnerable:
Tainted value is used at a sink function!

```
RE id='$id2'";
```

$id:	Tainted
$id2:	Tainted
$query:	Tainted

Sink
```
$r = mysql_query($query);
```

Case of the Input Sanitization

Source

```
$id = $_POST['id'];
```

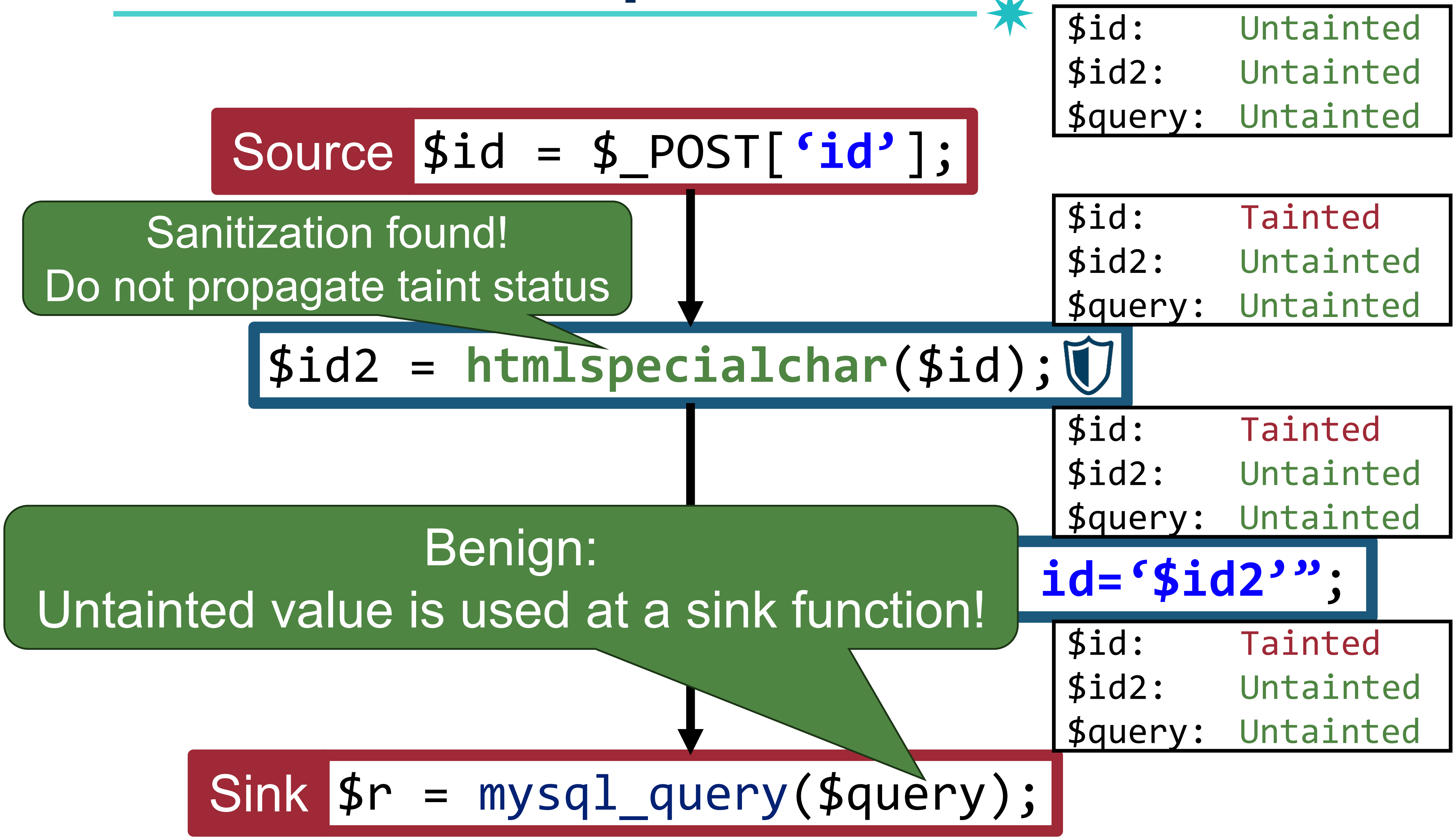
```
$id2 = htmlspecialchar($id);
```

```
$query = "SELECT * FROM users WHERE id='$id2'";
```

Sink

```
$r = mysql_query($query);
```

Case of the Input Sanitization



Intra-procedural Analysis

- A mechanism for performing analysis ***for each function***

```
<?php
  $id = $_POST['id'];
  $query = "SELECT * FROM users WHERE id='$id'";
  $query2 = "SELECT * FROM users WHERE id=123";
  $result = foo($query2)
  $result = foo($query)
?>
```

Analysis for this function

```
<?php
  function foo($fquery) {
    mysql_query($fquery)
  }
?>
```

Analysis for this function

Intra-procedural Analysis

```
$id: Untainted

$id = $_POST['id'];

$id: Tainted

$query = "SELECT * FROM users WHERE id='$id'";

$id: Tainted | $query: Tainted

$query2 = "SELECT * FROM users WHERE id=123";

$id: Tainted | $query: Tainted | $query2: Untainted

$result = foo($query2)

$result = foo($query)

mysql_query($fquery)
```

Intra-procedural Analysis

$id: Untainted

Produce false negatives!

$id: Tainted | $query: Tainted

```
$query2 = "SELECT * FROM users WHERE id=123";
```

$id: Tainted | $query: Tainted | $query2: Untainted

```
$result = foo($query2)
```

```
$result = foo($query)
```

Benign:
No sink

```
mysql_query($fquery)
```

Inter-procedural Analysis

- A mechanism for performing analysis ***across function boundaries***

$id: Untainted

```
$id = $_POST['id'];
```

$id: Tainted

```
$query = "SELECT * FROM users WHERE id='$id'";
```

$id: Tainted | $query: Tainted

```
$query2 = "SELECT * FROM users WHERE id=123";
```

$id: Tainted | $query: Tainted | $query2: Untainted

```
$result = foo($query2)
```

```
$result = foo($query)
```

call edge

```
mysql_query($fquery)
```

Context-insensitive Inter-procedural Analysis


```
$id: Untainted
```

```
$id = $_POST['id'];
```

```
$id: Tainted
```

```
$query = "SELECT * FROM users WHERE id='$id'";
```

```
$id: Tainted | $query: Tainted
```

```
$query2 = "SELECT * FROM users WHERE id=123";
```

```
$id: Tainted | $query: Tainted | $query2: Untainted
```

```
$result = foo($query2)
```

```
$result = foo($query)
```

call edge

Calling context:
`$fquery: Tainted`

```
mysql_query($fquery)
```

Context-insensitive Inter-procedural Analysis

$id: Untainted

```
$id = $_POST['id'];
```

$id: Tainted

Vulnerable: Tainted value is used at a sink function!

Vulnerable: Tainted value is used at a sink function!

```
$qu        = "SELECT * FROM        s WHERE id=123";
```

$id: Tain   d | $query: Tainted | $query2: Untainted

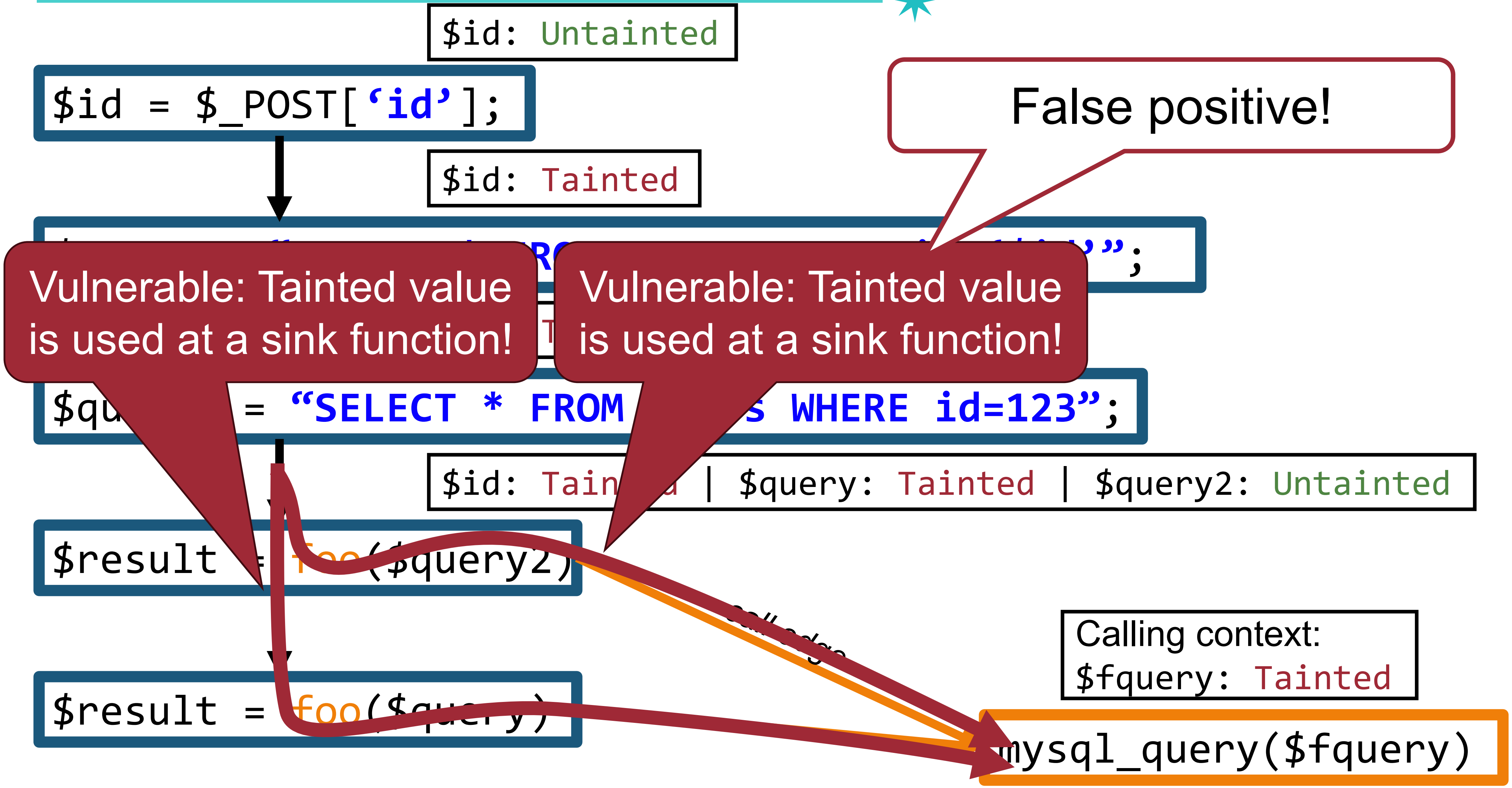
```
$result = foo($query2)
```

```
$result = foo($query)
```

Calling context:
$fquery: Tainted

```
mysql_query($fquery)
```

Context-insensitive Inter-procedural Analysis



Context-sensitive Inter-procedural Analysis


```
$id: Untainted
$id = $_POST['id'];
$id: Tainted
$query = "SELECT * FROM users WHERE id='$id'";
$id: Tainted | $query: Tainted
$query2 = "SELECT * FROM users WHERE id=
$id: Tainted | $query: Tainte
$result = foo($query2)
$result = foo($query)
mysql_query($fquery)
```

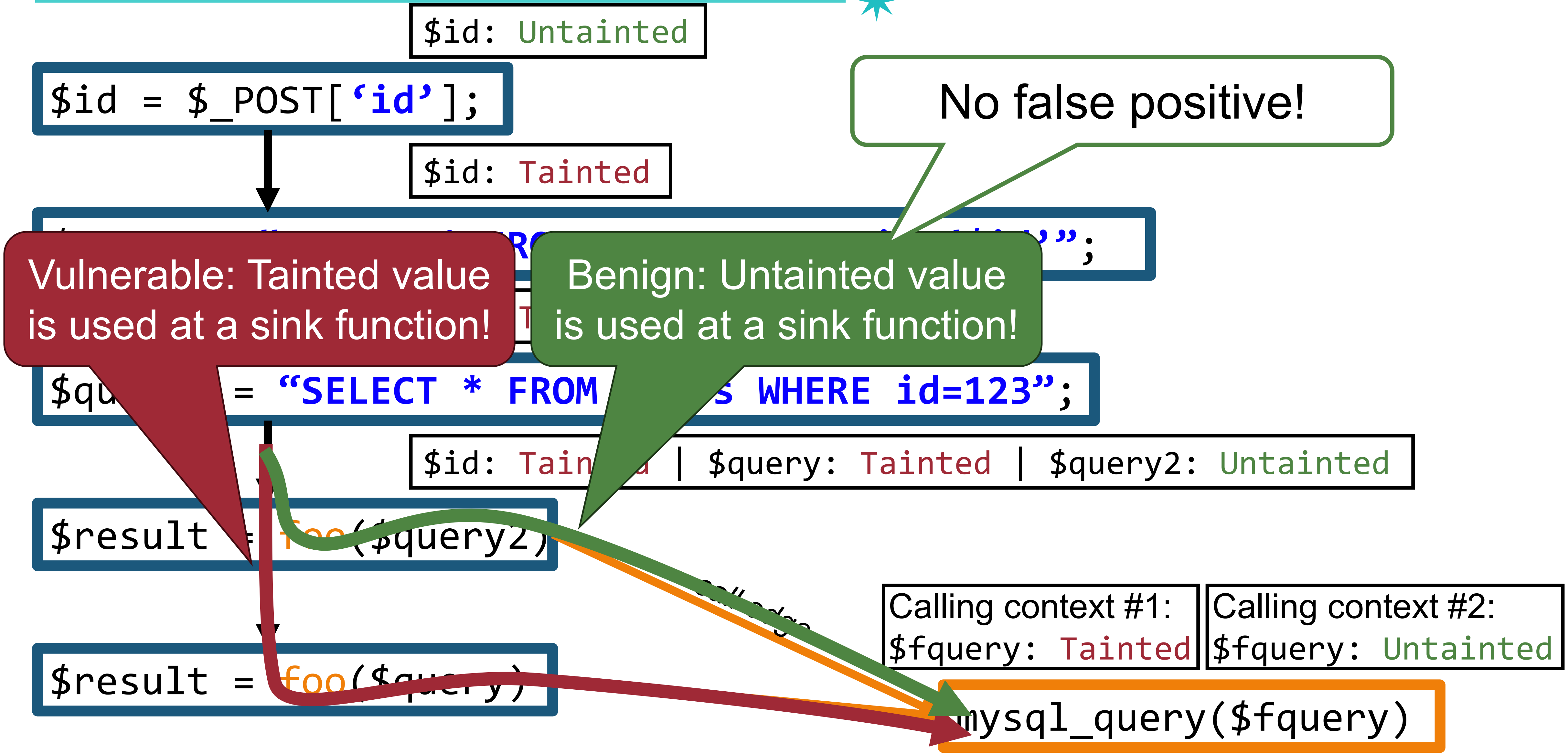
call edge

Maintains context-sensitive calling info.!

Calling context #1:
$fquery: Tainted

Calling context #2:
$fquery: Untainted

Context-sensitive Inter-procedural Analysis



A Limitation of Context-sensitive Analysis

- Problem:
  - Performance: expensive, as it gets deeper…

```
<?php
  $db_query(query1);
  $db_query(query2);
  $db_query(query3);
?>
```

```
<?php
  function db_query($query) {
    foo($query);
    foo($query);
  }
?>
```

```
<?php
  function foo($fquery) {
    mysql_query($fquery)
  }
?>
```

Number of calling context to analyze?

Taint Analysis: Pros and Cons

- Pros
  - Better performance than symbolic execution, enabling scalable testing
    - There is no need to manage symbols
    - There is no need for SMT solving

- Cons
  - Require developer's participations to generate inputs
  - Produce false positives: according to its tracking policy …
  - Produce false negatives: according to its tracking policy …

Conclusion

- **Statis analysis** is a method of detecting potential security bugs without having to execute the program

- Symbolic Execution
  - Executes programs with symbolic values

- Taint Analysis
  - Identify whether the user input can reach sensitive sink