

Who tests the tools we trust?

Deriving oracles for verification infrastructure



Maria Christakis (TU Wien) · KLEE 2026

Program analyzers find software bugs

A **program analyzer** is software that analyzes other software to find bugs



Program analyzers find software bugs

A **program analyzer** is software that analyzes other software to find bugs

Analyzers can **prevent catastrophic defects** in safety-critical software



Program analyzers make correctness claims

*This proof attests to
a valid computation*

*This abstract-domain
operation is **sound***

*This source **flows**
to this sink*

Program Analyzer

*This program is **safe***

*This CHC system is
satisfiable / unsatisfiable*

*This assertion is
reachable / unreachable*

Program analyzers make correctness claims

*This proof attests to
a valid computation*

*This abstract-domain
operation is **sound***

*This source **flows**
to this sink*

Program Analyzer

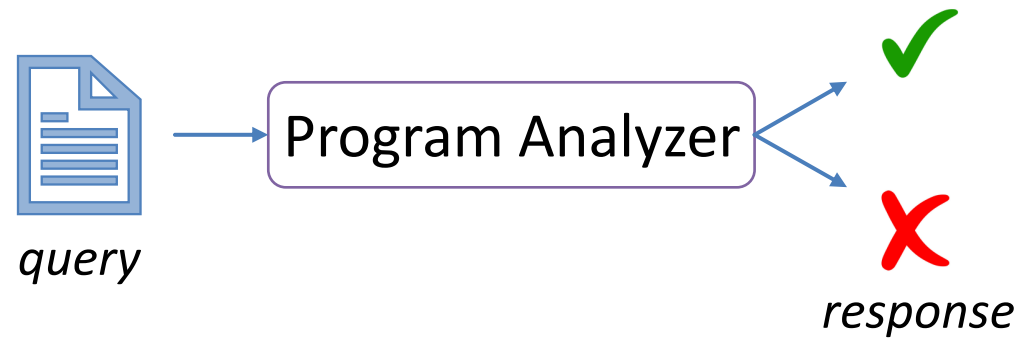
*This program is **safe***

*This CHC system is
satisfiable / unsatisfiable*

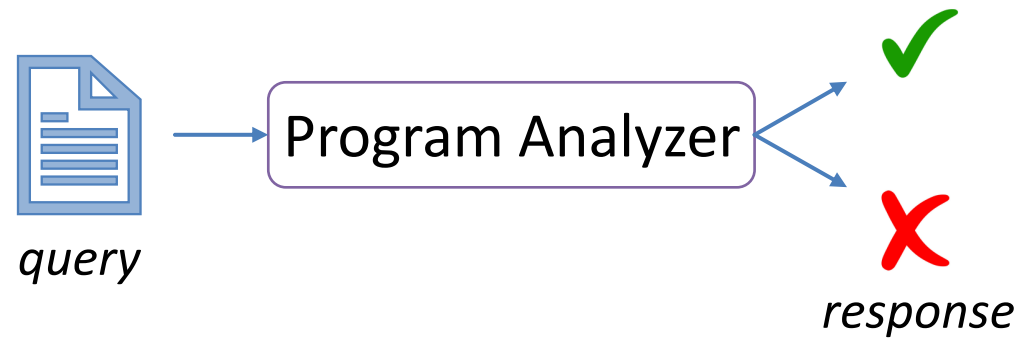
*This assertion is
reachable / unreachable*

These claims are **trusted** by developers and other tools!

Program analyzers are software too



Program analyzers are software too



- Under the hood
 - Frontends, encodings, solvers, reporting logic, optimizations...
- Complex implementations offer many places for **bugs**

Why not verify the program analyzers?

- End-to-end verification of modern analyzers gives the strongest assurance

Why not verify the program analyzers?

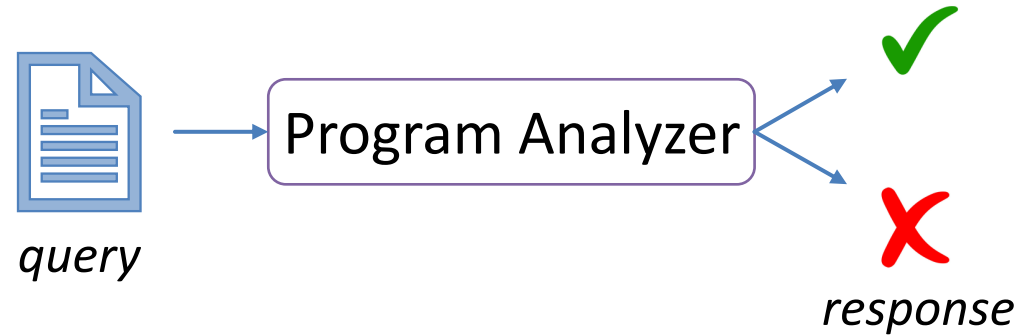
- End-to-end verification of modern analyzers gives the strongest assurance
- End-to-end verification of modern analyzers is rarely realistic
 - Tools are large, complex, layered, and evolving
 - Bugs can hide outside the “core algorithm”

Why not just test them?

Why not just test them?

The oracle problem:

How can we tell if an analyzer is wrong?



Why not just test them?

The oracle problem:

How can we tell if an analyzer is wrong?

If the analyzer says *safe*, is it correct or unsound?

If it reports a bug, is it real or a false alarm?

If it produces an artifact, is it consistent with the response?

An oracle toolbox

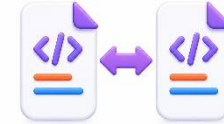
When the oracle is not given, **we engineer it!**



Differential testing:
the oracle comes from disagreement



Program generation:
the oracle is built into the query



Metamorphic testing:
the oracle comes from a relation



Specification-based testing:
the oracle is the specification



Interrogation testing:
the oracle becomes adaptive

This is not a theoretical problem

- Tested > 50 program analyzers
 - Model checkers
 - Abstract interpreters
 - Symbolic execution engines
 - SMT and CHC solvers
 - Datalog engines
 - Information-flow checkers
 - Termination checkers
- Found > 170 critical bugs

This is not a theoretical problem

- Tested > 50 program analyzers
 - Model checkers
 - Abstract interpreters
 - Symbolic execution engines
 - SMT and CHC solvers
 - Datalog engines
 - Information-flow checkers
 - Termination checkers
- Found > 170 critical bugs

The bottleneck was not generating tests, it was **deriving oracles!**

Specification-based testing

A specification makes the oracle **direct**
but only after **we pay the cost of writing it**

- The oracle comes from a **contract**
- The contract states what the implementation must preserve
- The tests check whether generated queries violate the contract



Example: An interval transformer goes wrong

```
int foo(int p) {  
    int a = p + 1;  
    ...  
}
```

- Let's assume an Interval domain for bounded integers
- Without any knowledge about p , its abstract value is $[INT_MIN, INT_MAX]$
- A naïve implementation of the addition transformer would lead to an overflow and produce $[INT_MIN + 1, INT_MIN] \equiv \perp$
- The analyzer treats reachable code as **unreachable, which is unsound**

Example: A necessary property is enough

- If the abstract element $\bar{x}$ (capturing the pre-state of an assignment) is non-bottom, then the post-state of the assignment $v := e$ should also be non-bottom for all variables v and well-formed expressions e :

$$\bar{x} \neq \perp \Rightarrow \text{assign}(\bar{x}, v := e) \neq \perp$$

- The oracle is a **necessary** contract
 - It need not compute the best abstract result

Example: Turning the contract into tests

- Once the contract is **executable**, fuzzing can search for violations
 - Populate a pool of abstract elements
 - Select a random $\bar{x}$
 - Apply $assign(\bar{x}, v := e)$
 - Check $\bar{x} \neq \perp \Rightarrow assign(\bar{x}, v := e) \neq \perp$

Specification-based testing: Lessons

Direct oracles are not free

- Strength
 - Precise, local, directly checkable oracle
- Best fit
 - Components with clean necessary properties
- Cost
 - Writing good specifications requires expertise and effort
 - Hard to scale to whole, evolving analyzers

Program generation

Instead of writing a full specification,
generate queries whose properties are known by construction

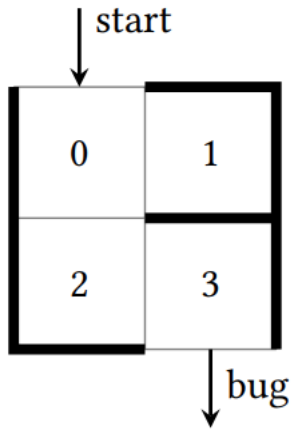
- Generate P such that property φ holds
- Run analyzer on P
- Check whether the analyzer agrees with φ



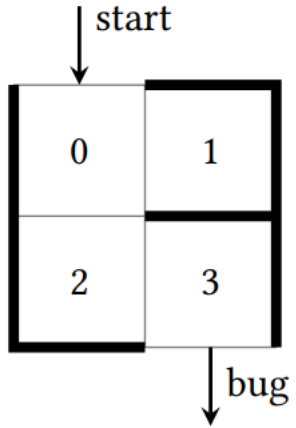
Example: A maze with a known exit

- C programs are generated from SMT constraints
- The program is a maze: execution searches for an exit
- Reaching the exit triggers the bug
- The constraints control whether the exit is reachable

Example: The generated program carries the oracle

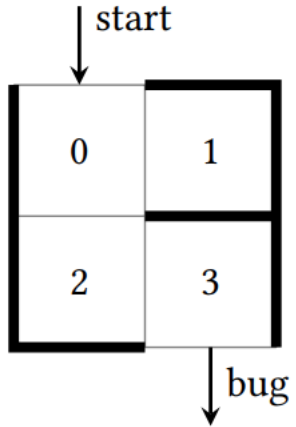


Example: The generated program carries the oracle



```
void func_bug(char *input, ...) { abort(); }
```

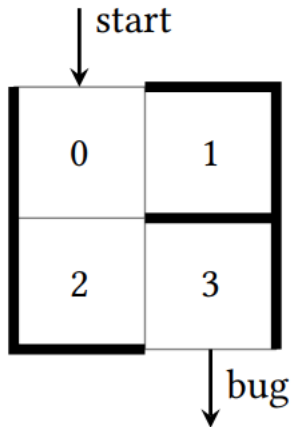
Example: The generated program carries the oracle



```
void func_bug(char *input, ...) { abort(); }

void func_0(char *input, ...){
             // compute a variable used in conditionals
    if (        ) func_1(input, ...); // go to 1
    else if (        ) func_2(input, ...); // go to 2
    else fatal_error("Hit the wall");
}
```

Example: The generated program carries the oracle

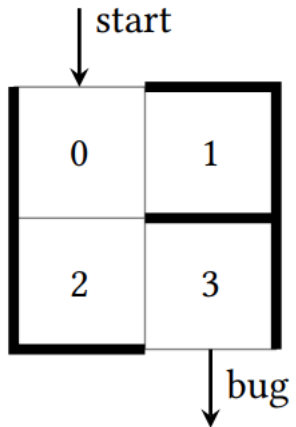


```
void func_bug(char *input, ...) { abort(); }
```

```
void func_0(char *input, ...){  
               // compute a variable used in conditionals  
    if (          ) func_1(input, ...); // go to 1  
    else if (          ) func_2(input, ...); // go to 2  
    else fatal_error("Hit the wall");  
}
```

```
void func_3(char *input, ...){  
               // compute a variable used in conditionals  
    if (          ) func_bug(input, ...); // exit  
    else if (          ) func_2(input, ...); // go to 2  
    else fatal_error("Hit the wall");  
}
```

Example: The generated program carries the oracle



```
void func_bug(char *input, ...) { abort(); }
```

```
void func_0(char *input, ...){  
           // compute a variable used in conditionals  
    if (      ) func_1(input, ...); // go to 1  
    else if (      ) func_2(input, ...); // go to 2  
    else fatal_error("Hit the wall");  
}
```

```
void func_3(char *input, ...){  
           // compute a variable used in conditionals  
    if (      ) func_bug(input, ...); // exit  
    else if (      ) func_2(input, ...); // go to 2  
    else fatal_error("Hit the wall");  
}
```

(UN)SAT constraints
make the exit / bug
(un)reachable

Example soundness bug in CPAchecker

```
int main () {  
    unsigned int x = *;  
    if (x < 10) return 0;  
    if (x > 21) return 0;  
    if ((20 / x) >= 2) return 0;  
    if (x < 21) return 0;  
    assert(0);  
}
```

Example soundness bug in CPAchecker

```
int main () {  
    unsigned int x = *;  
    if (x < 10) return 0;  
    if (x > 21) return 0;  
    if ((20 / x) >= 2) return 0;  
    if (x < 21) return 0;  
    assert(0);  
}
```



Program generation: Lessons

The program generator must preserve the oracle

- Strength
 - Oracle is built into the query, no specification required
- Best fit
 - Inputs whose relevant property can be constructed
 - Settings where restricted programs are still challenging
- Cost
 - Generated programs must be restricted enough to preserve ground truth
 - Generated programs should be realistic and diverse enough to expose bugs

Differential testing

When neither a specification nor constructed ground truth is available,
use disagreement as the oracle

- Run multiple analyzers on the same program
- Compare the claims they make
- Check whether those claims disagree
- A disagreement means: at least one claim is wrong



Example: The same program, many analyzers

```
int main() {  
    int i = 0;  
    while (i < 100000) {  
  
        i = i + 1;  
    }  
  
    return i;  
}
```

- Start from a correct, assertion-free program

Example: The same program, many analyzers

```
int main() {  
    int i = 0;  
    while (i < 100000) {  
        assert(i != 13);  
        i = i + 1;  
    }  
  
    return i;  
}
```

- Insert an assertion over variables in scope

Example: The same program, many analyzers

```
int main() {  
    int i = 0;  
    while (i < 100000) {  
        assert(i != 13);  
        i = i + 1;  
    }  
  
    return i;  
}
```

- Run several analyzers on the same variant
- Compare their claims about the assertion

Example: Disagreement exposes the bug

Most analyzers
report a violation

CBMC returns
unknown

```
int main() {  
    int i = 0;  
    while (i < 100000) {  
        assert(i != 13);  
        i = i + 1;  
    }  
  
    return i;  
}
```

SMACK returns
safe

Example: Disagreement exposes the bug

Most analyzers
report a violation

CBMC returns
unknown

```
int main() {  
    int i = 0;  
    while (i < 100000) {  
        assert(i != 13);  
        i = i + 1;  
    }  
  
    return i;  
}
```

SMACK returns
safe

Soundness issue

Example: Disagreement exposes the bug

```
int main() {  
    int i = 0;  
    while (i < 100000) {  
  
        i = i + 1;  
    }  
    assert(i != 10);  
    return i;  
}
```

Example: Disagreement exposes the bug

Most analyzers
verify the assertion

```
int main() {  
    int i = 0;  
    while (i < 100000) {  
        i = i + 1;  
    }  
    assert(i != 10);  
    return i;  
}
```

SMACK returns
unsafe

Example: Disagreement exposes the bug

Most analyzers
verify the assertion

```
int main() {  
    int i = 0;  
    while (i < 100000) {  
        i = i + 1;  
    }  
    assert(i != 10);  
    return i;  
}
```

SMACK returns
unsafe

Precision issue

Differential testing: Lessons

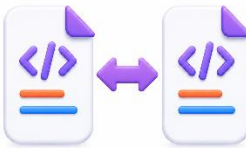
Disagreement reveals a problem, but not which analyzer is wrong

- Strength
 - No specification or constructed ground truth required
- Best fit
 - Multiple analyzers for the same task with comparable claims
 - Queries where disagreement is meaningful
- Cost
 - Requires more than one implementation
 - Disagreement does not identify the faulty analyzer

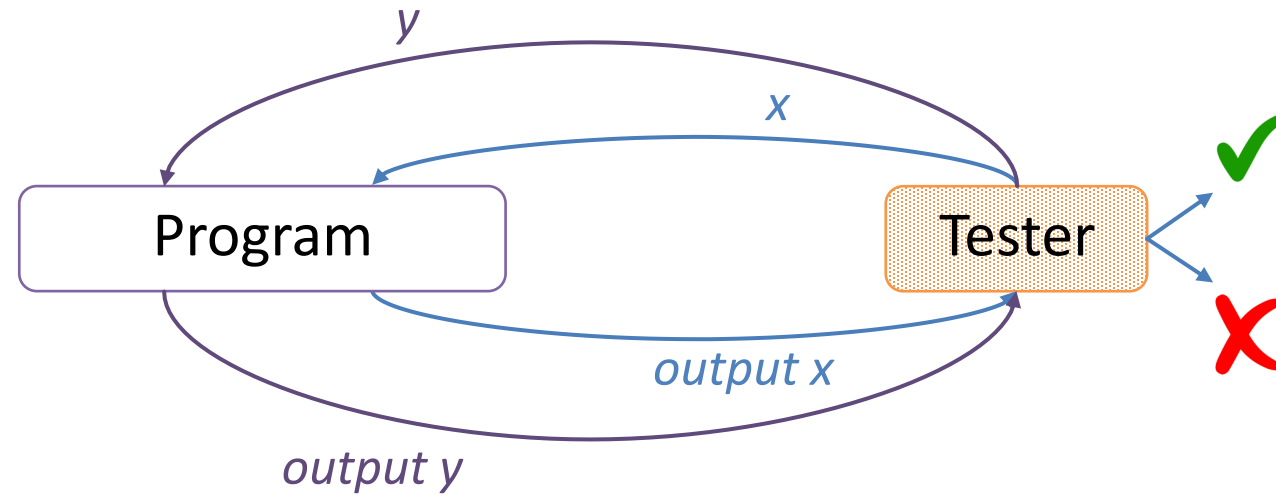
Metamorphic testing

When one analyzer is available,
derive an oracle from a relation between queries and responses

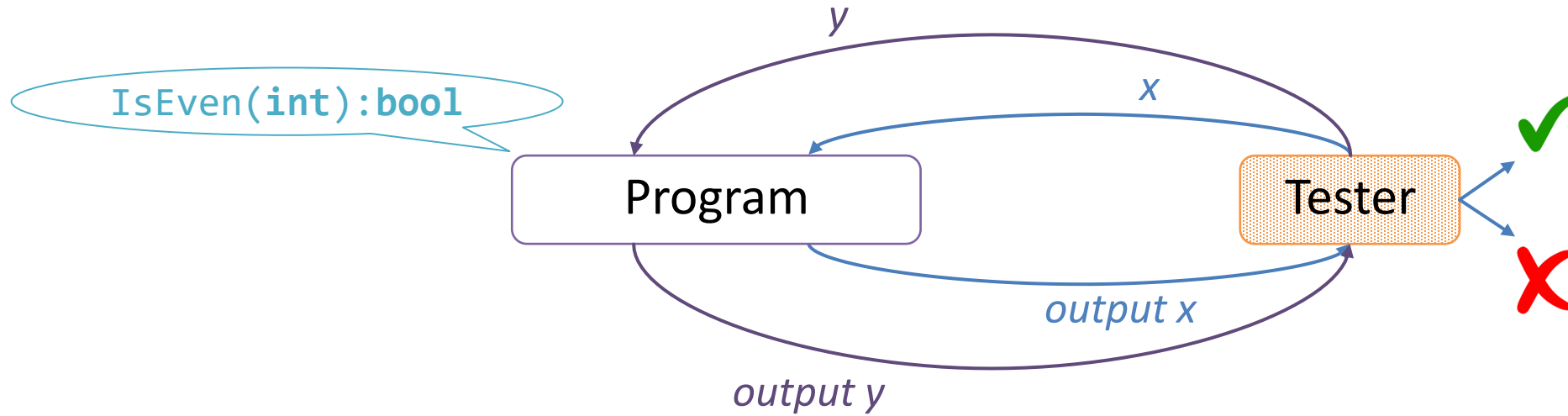
- Run the analyzer on query x
- Transform x into related query y
- Predict the analyzer response to y
- Run the same analyzer on y
- A violated prediction exposes an **incorrect response**



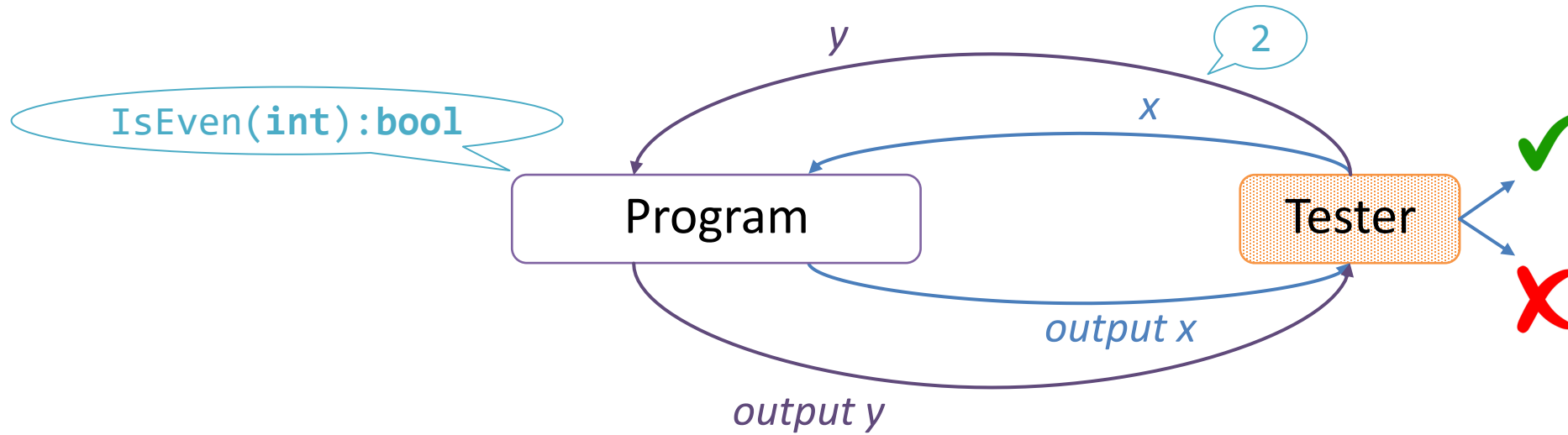
Example: A relation between inputs



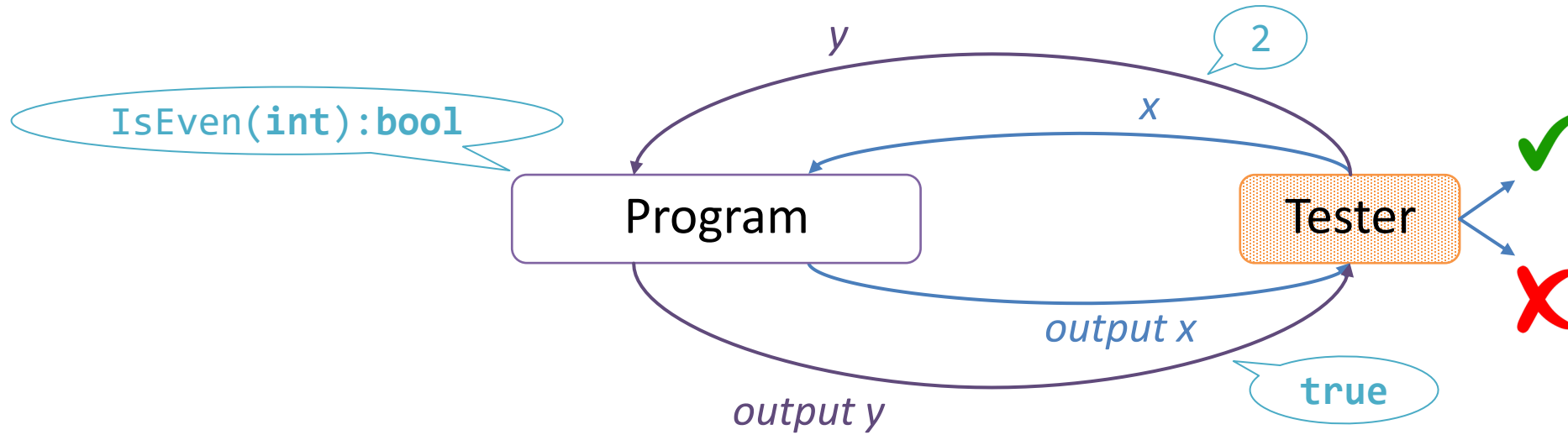
Example: A relation between inputs



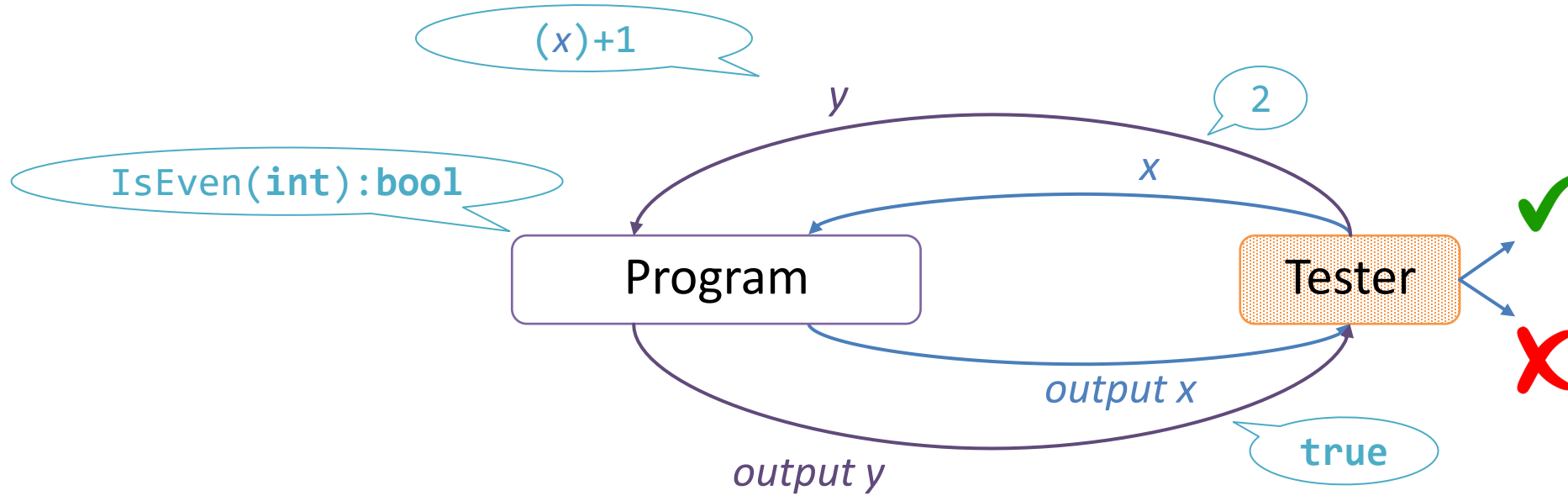
Example: A relation between inputs



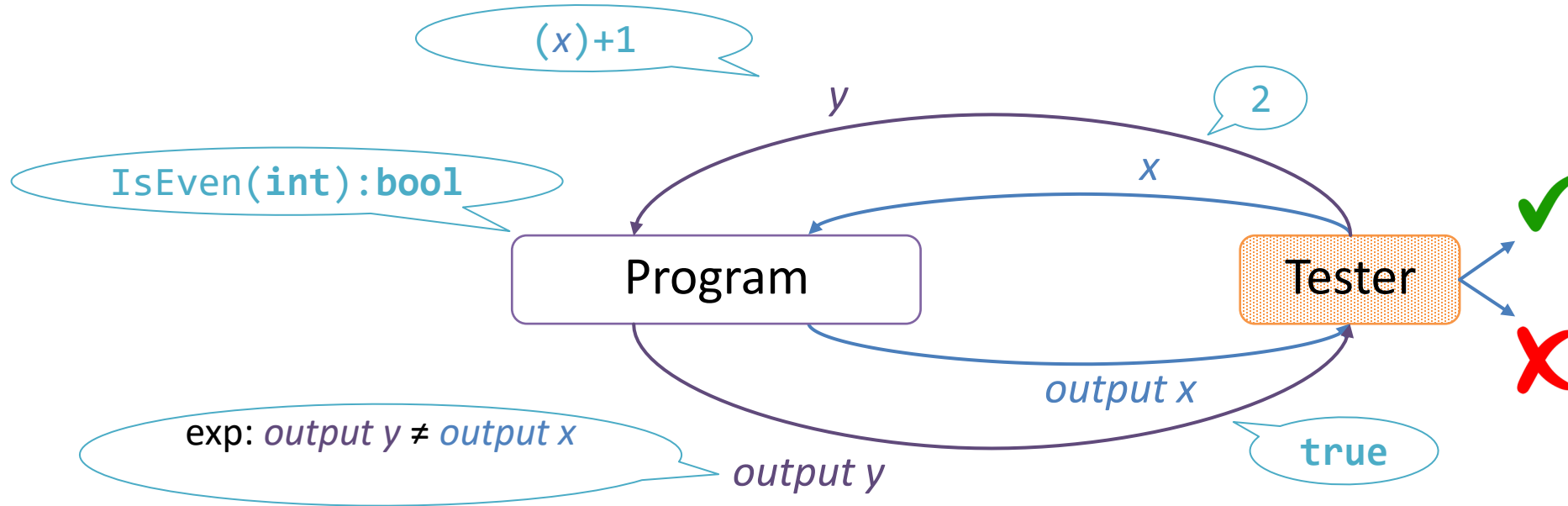
Example: A relation between inputs



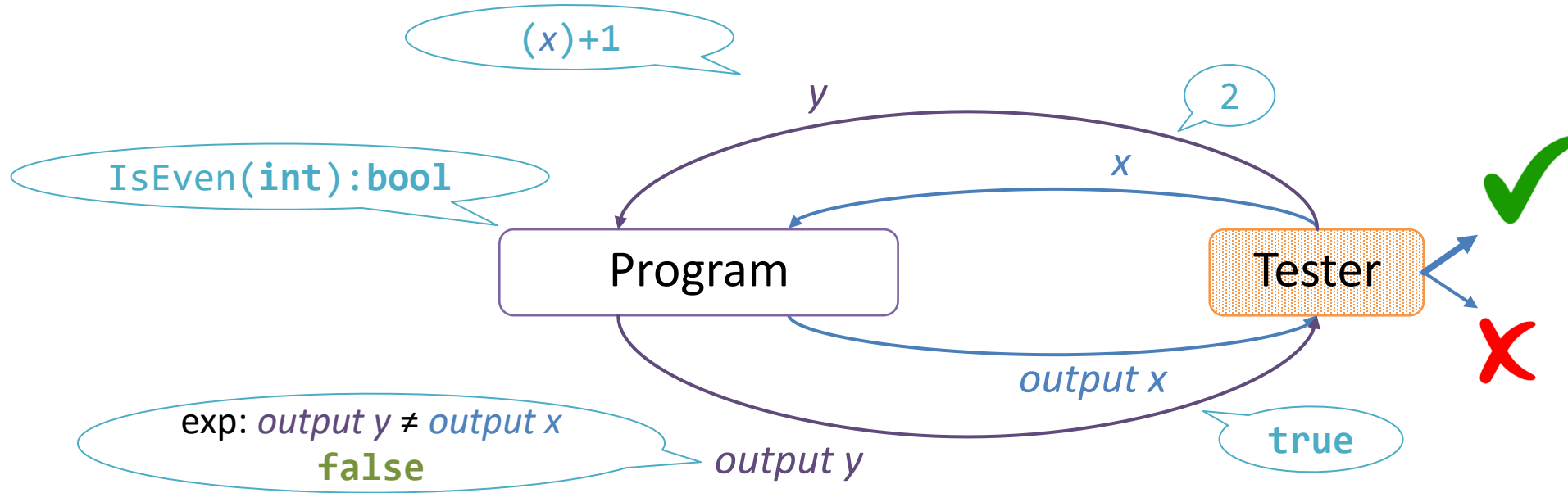
Example: A relation between inputs



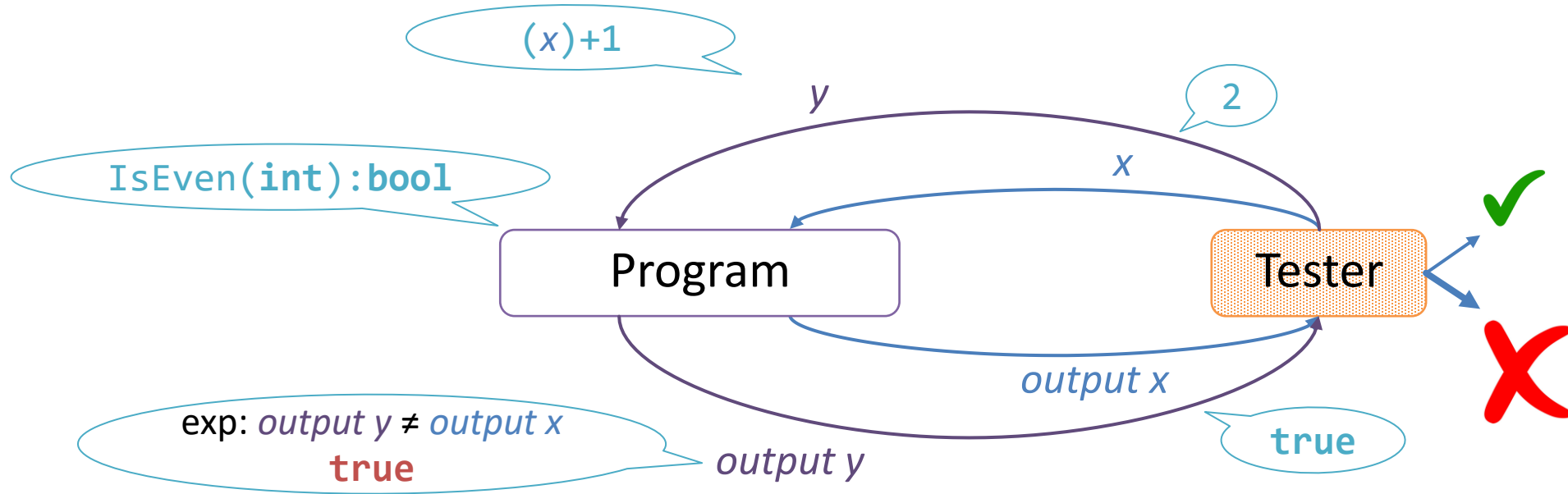
Example: A relation between inputs



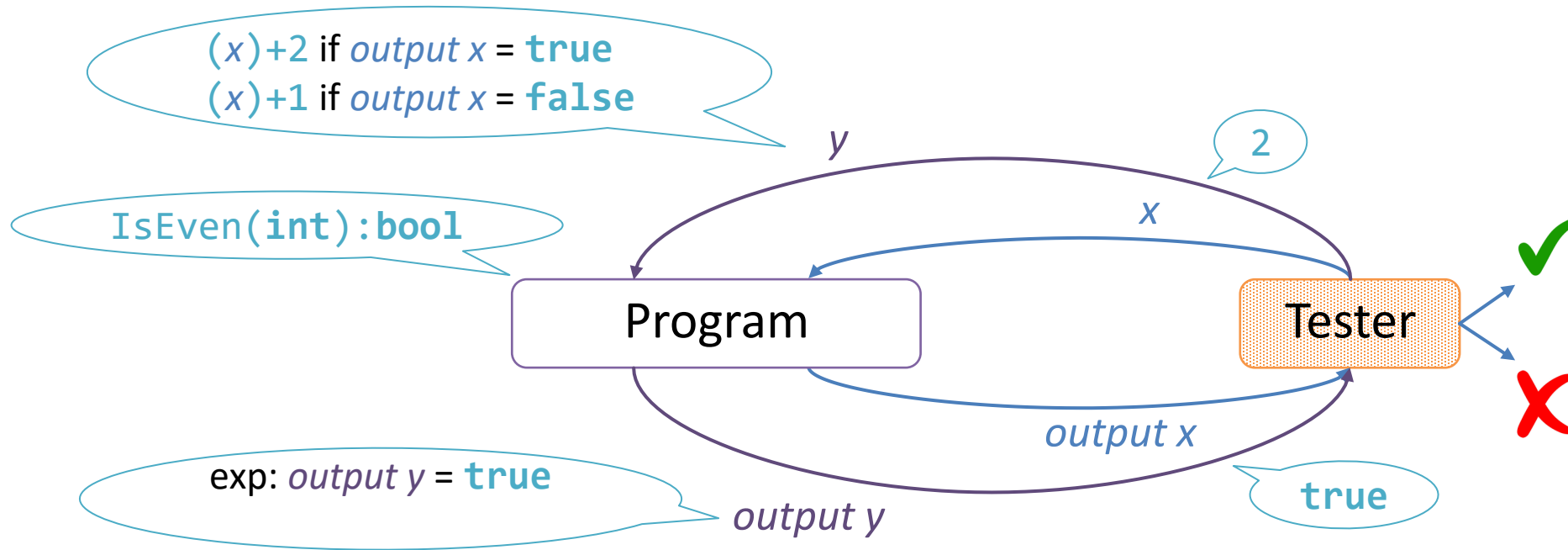
Example: A relation between inputs



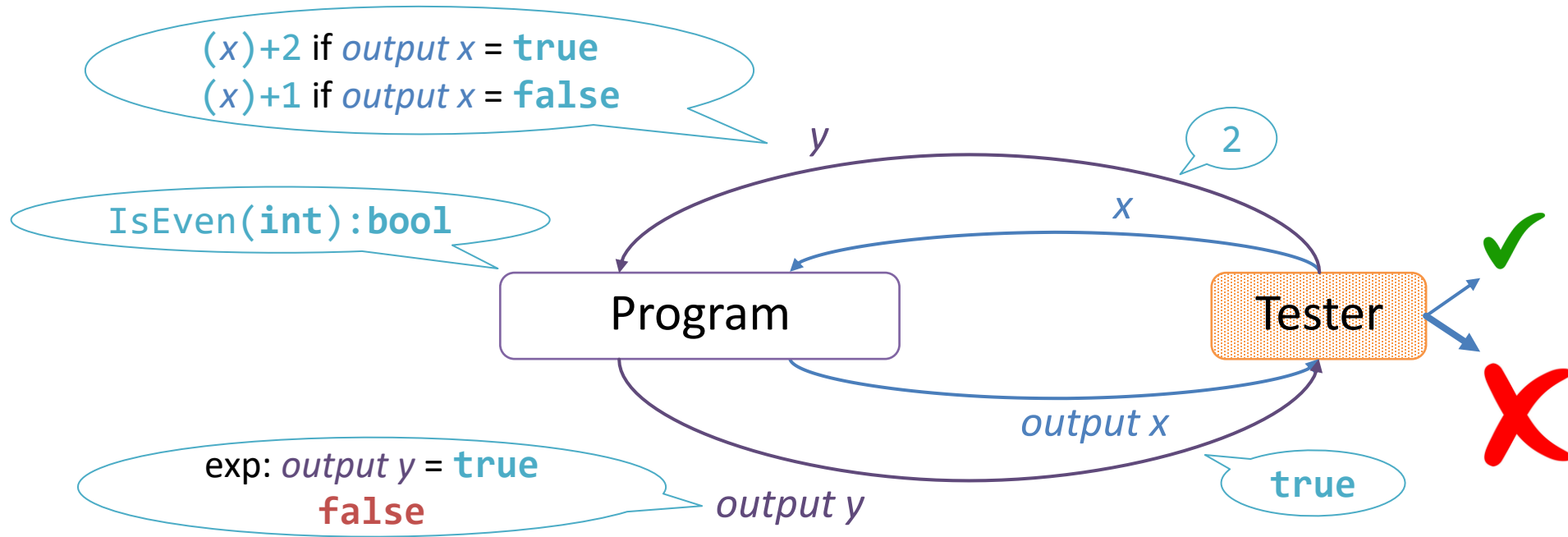
Example: A relation between inputs



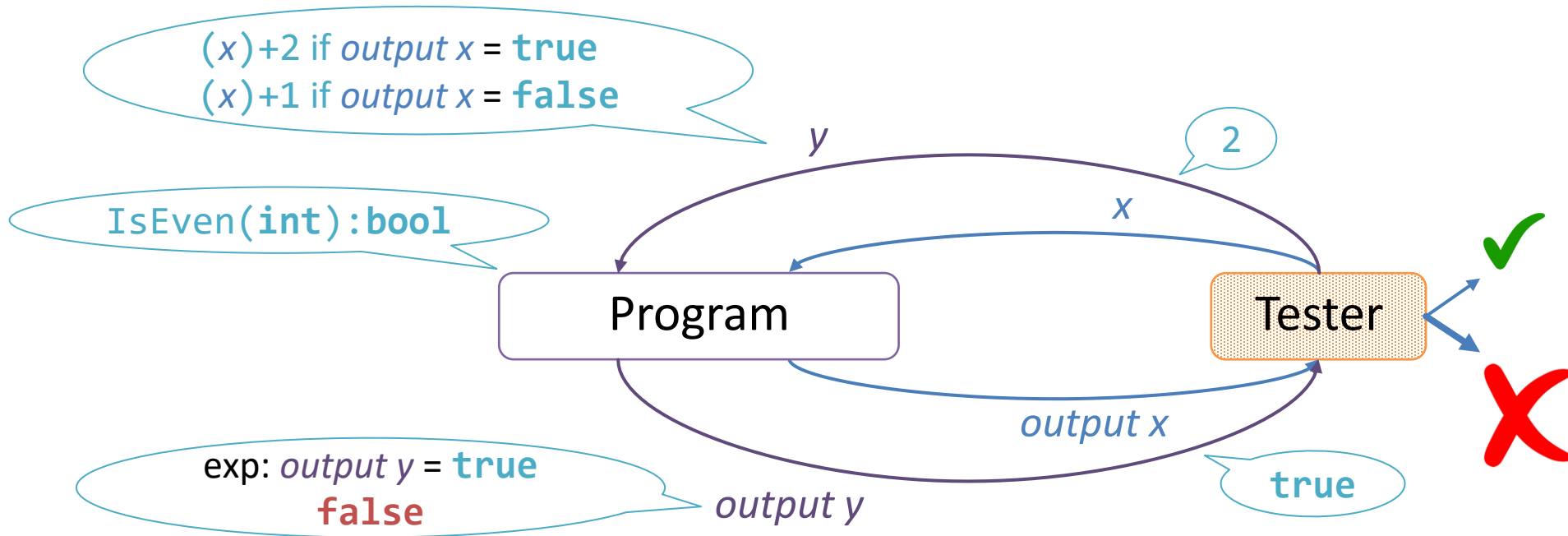
Example: A relation between inputs



Example: A relation between inputs



Example: A relation between inputs



We may not know every output, but we may know how outputs should change

The relation becomes the oracle

- Metamorphic testing specifies a **metamorphic relation**
 - A relation R' over inputs and a corresponding necessary relation R^O over outputs

The relation becomes the oracle

- Metamorphic testing specifies a **metamorphic relation**
 - A relation R^I over inputs and a corresponding necessary relation R^O over outputs

For our example with `IsEven(int):bool`, we have:

$$R^I(i_x, i_y) := i_y = i_x + 1$$

$$R^O(o_x, o_y) := o_y \neq o_x$$

The relation becomes the oracle

- Metamorphic testing specifies a **metamorphic relation**
 - A relation R^I over inputs and a corresponding necessary relation R^O over outputs

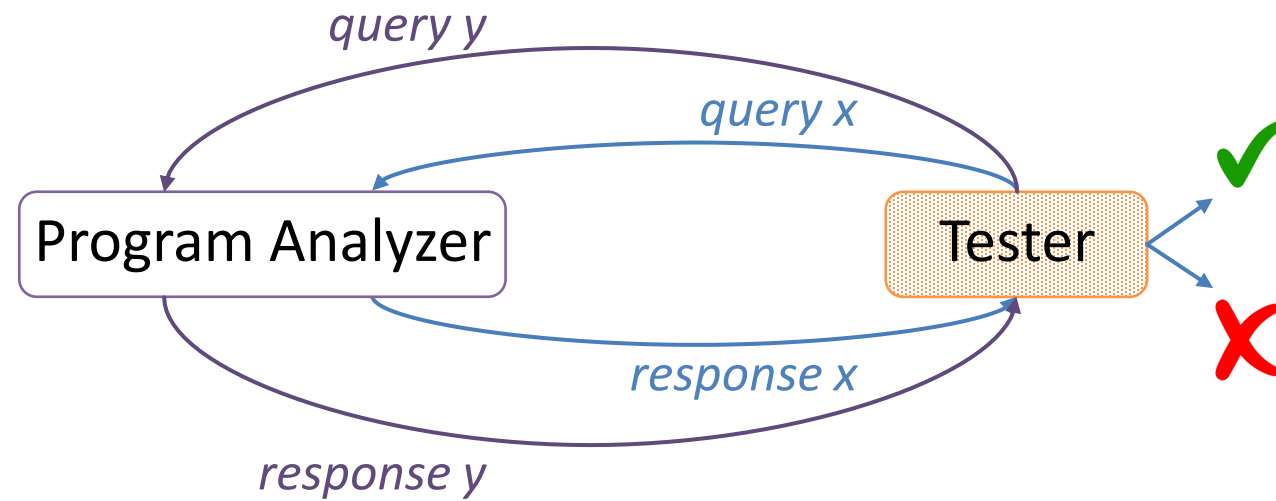
For our example with `IsEven(int):bool`, we have:

$$R^I(i_x, i_y) := i_y = i_x + 1$$

$$R^O(o_x, o_y) := o_y \neq o_x$$

- $R^I(i_x, i_y) \Rightarrow R^O(o_x, o_y)$ provides an **oracle for correct behavior** of the program

From programs to program analyzers



- The program is now the **analyzer**
- Inputs become **queries**
- Outputs become **analyzer responses**

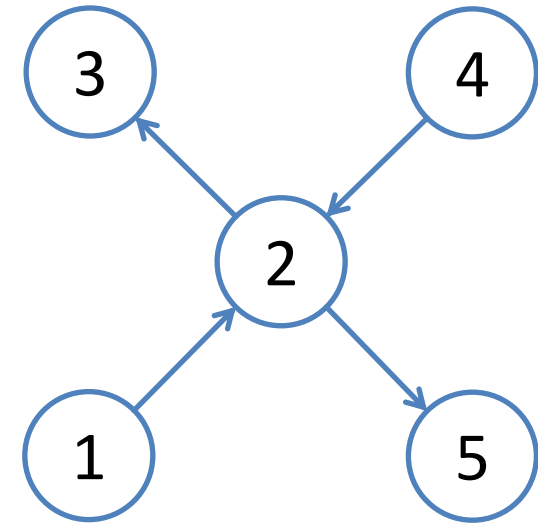
Example: Datalog engines

```
// facts  
edge(1,2).  
edge(2,3).  
edge(4,2).  
edge(2,5).  
  
// rules  
reachable(X,Y) :- edge(X,Y).  
reachable(X,Z) :- edge(X,Y), reachable(Y,Z).
```

Example: Datalog engines

```
// facts
edge(1,2).
edge(2,3).
edge(4,2).
edge(2,5).

// rules
reachable(X,Y) :- edge(X,Y).
reachable(X,Z) :- edge(X,Y), reachable(Y,Z).
```



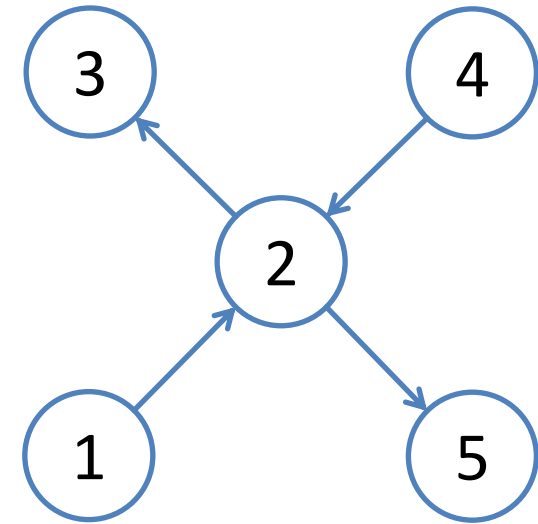
Example: Datalog engines

```
// facts
edge(1,2).
edge(2,3).
edge(4,2).
edge(2,5).

// rules
reachable(X,Y) :- edge(X,Y).
reachable(X,Z) :- edge(X,Y), reachable(Y,Z).
```

Response:

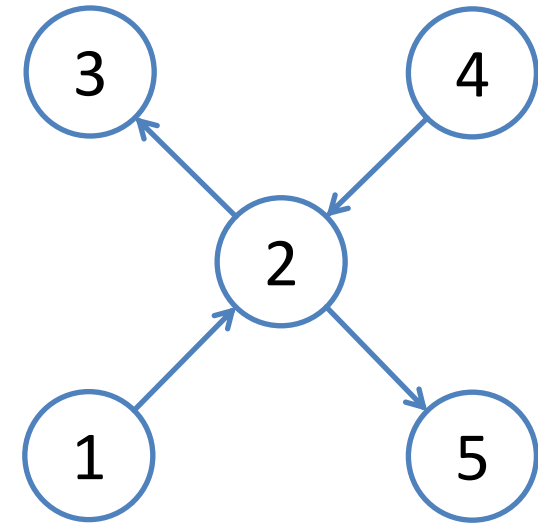
```
(1,2) (1,3) (1,5) (2,3)
(2,5) (4,2) (4,5) (4,3)
```



Example: Datalog engines

```
// facts
edge(1,2).
edge(2,3).
edge(4,2).
edge(2,5).

// rules
reachable(X,Y) :- edge(X,Y), edge(W,Y).
reachable(X,Z) :- edge(X,Y), reachable(Y,Z).
```



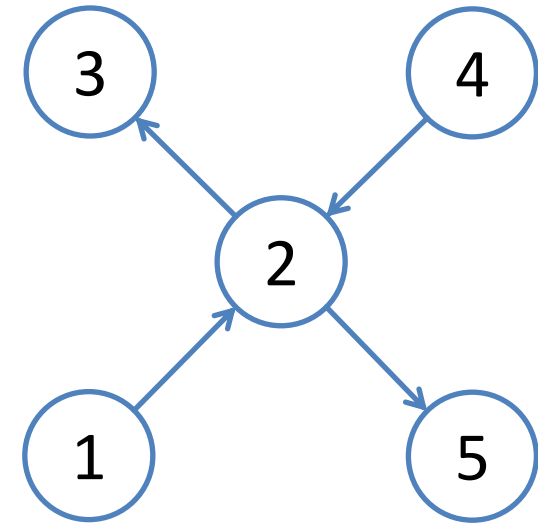
Response:

```
(1,2) (1,3) (1,5) (2,3)
(2,5) (4,2) (4,5) (4,3)
```

Example: Datalog engines

```
// facts
edge(1,2).
edge(2,3).
edge(4,2).
edge(2,5).

// rules
reachable(X,Y) :- edge(X,Y), edge(W,Y).
reachable(X,Z) :- edge(X,Y), reachable(Y,Z).
```



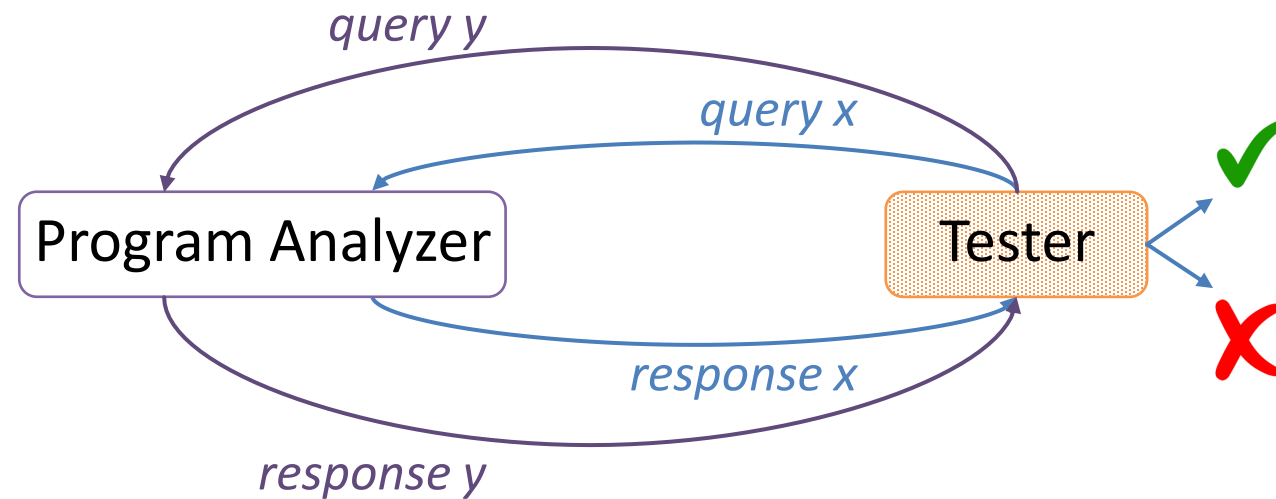
The transformation **changes the query text while preserving the response**

Metamorphic testing: Lessons

Relations remove the need for multiple analyzers,
but the relation itself must be designed

- Strength
 - No full functional specification, constructed ground truth, or multiple implementations required
- Best fit
 - Queries with meaningful transformations
 - Known relations between the expected responses
- Cost
 - Requires designing relations that can expose deep bugs
 - Uses only limited information from the first run

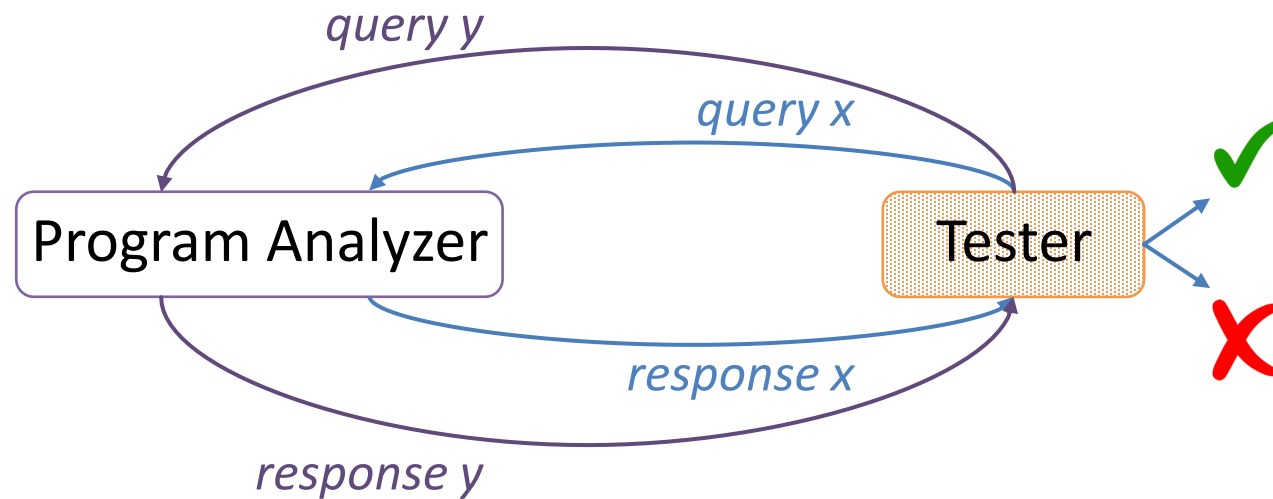
Metamorphic testing leaves information on the table



Query *x* is generated from scratch

Query *y* is at best generated using *response x*

Metamorphic testing leaves information on the table



Query *x* is generated from scratch

Query *y* is at best generated using *response x*

Metamorphic testing is **stateless** across test pairs and **artifact-blind**

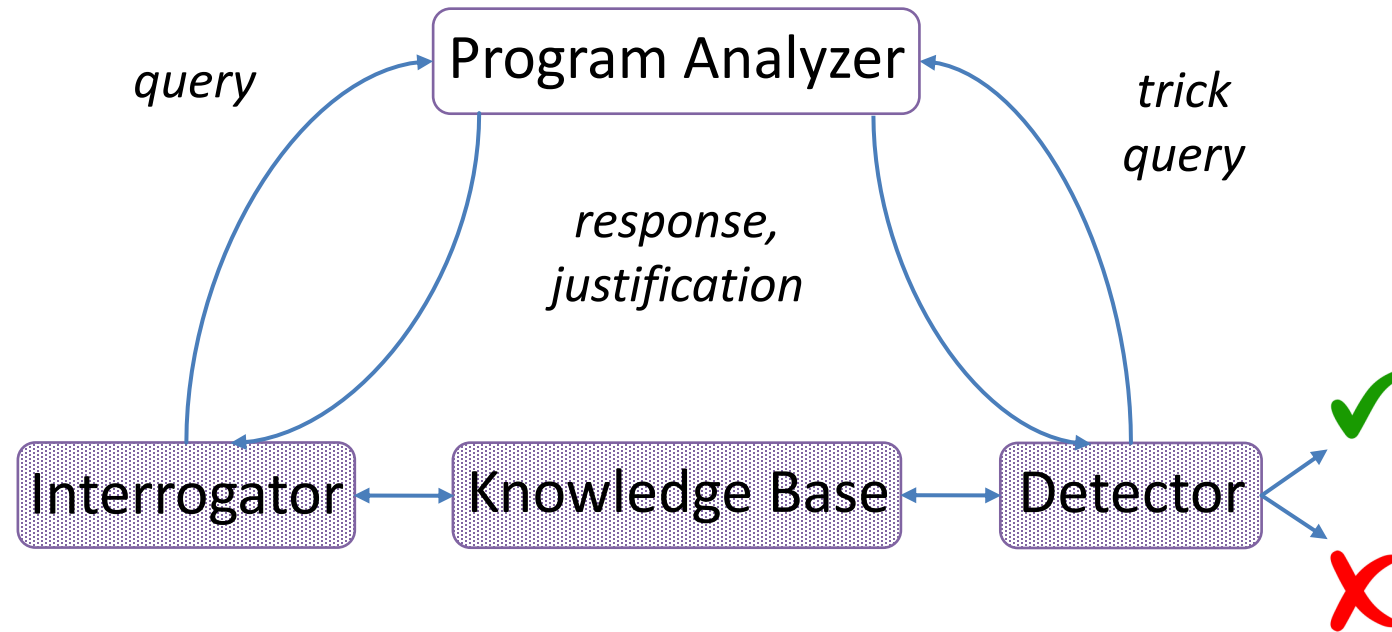
Interrogation testing

Use the analyzer's **previous responses and artifacts** to accumulate knowledge and generate stronger follow-up queries

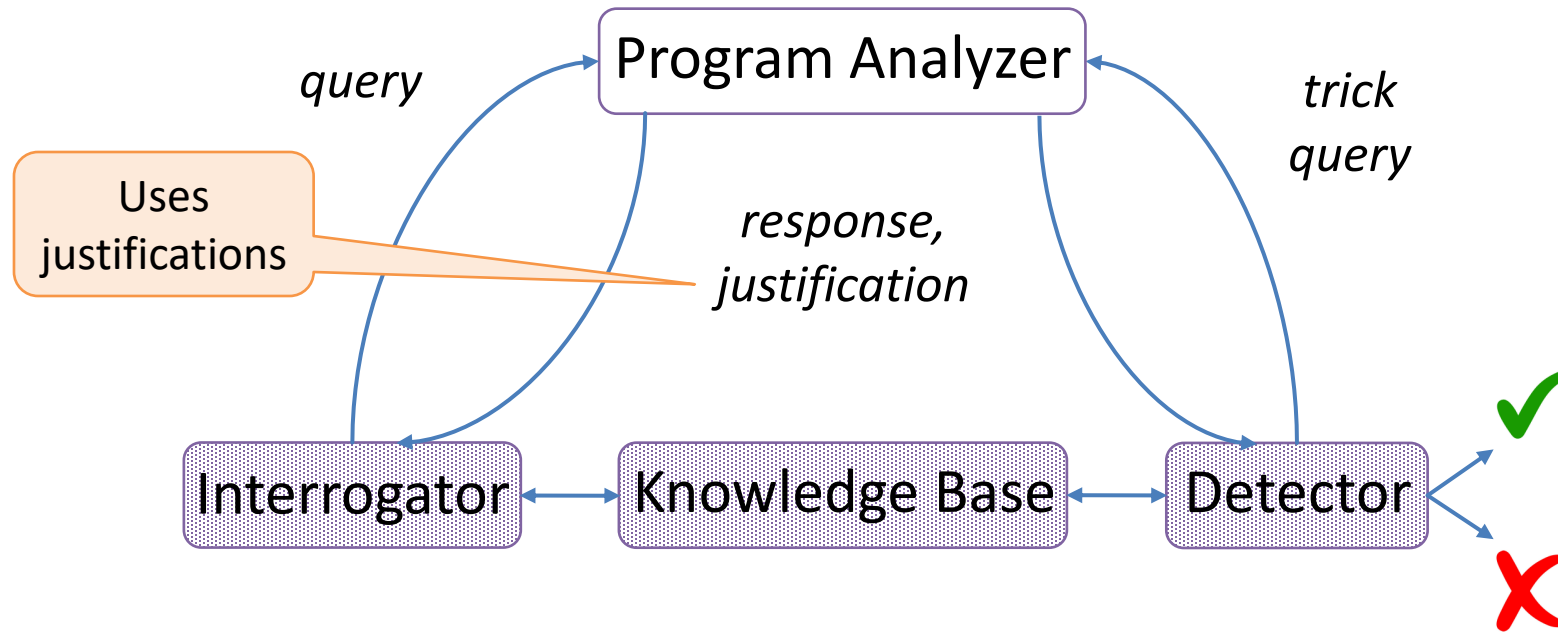
- Ask query x and inspect the **response and artifacts**
- Store what was learned in a **knowledge base**
- Construct follow-up query y
- Check whether the tool **contradicts accumulated knowledge**



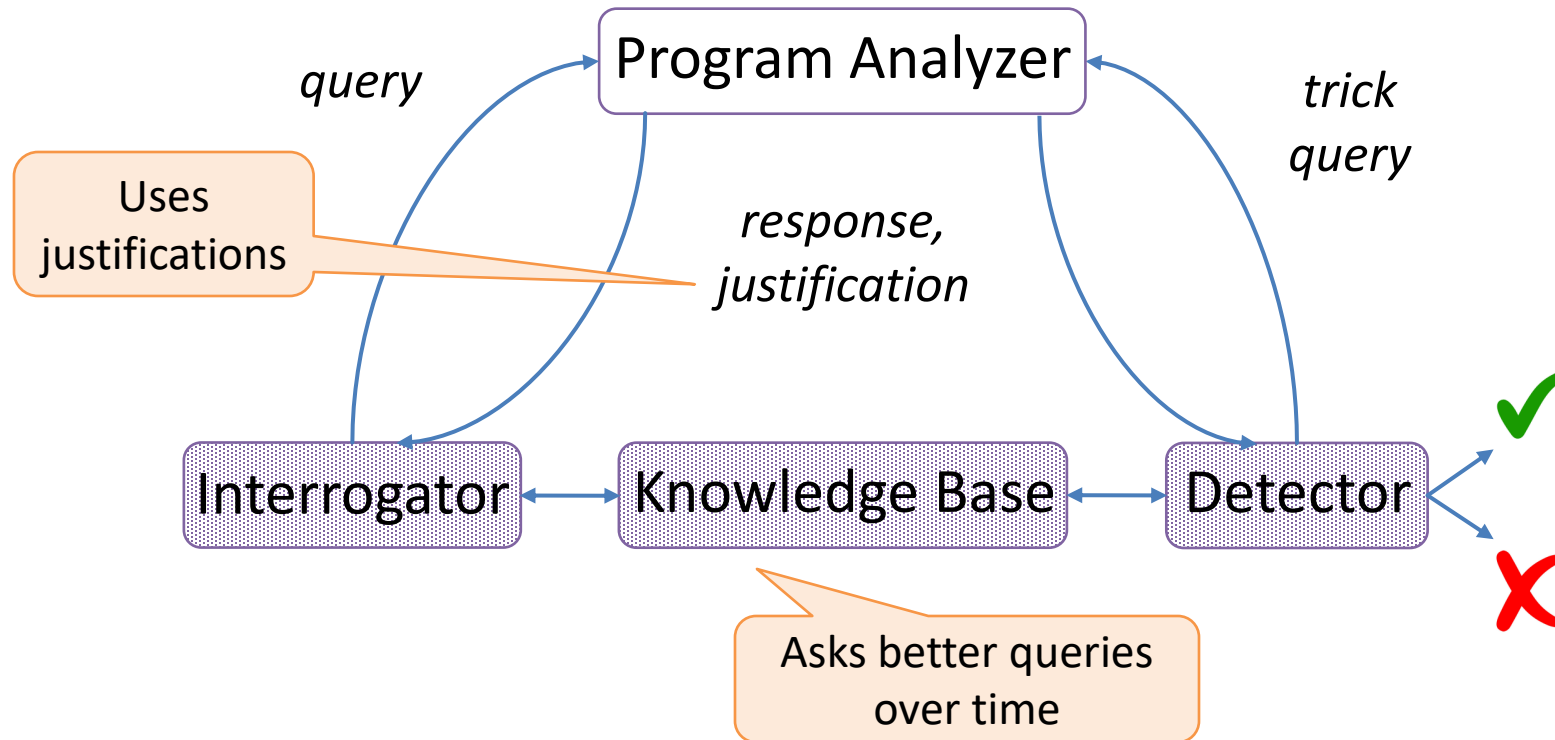
Interrogating the analyzer



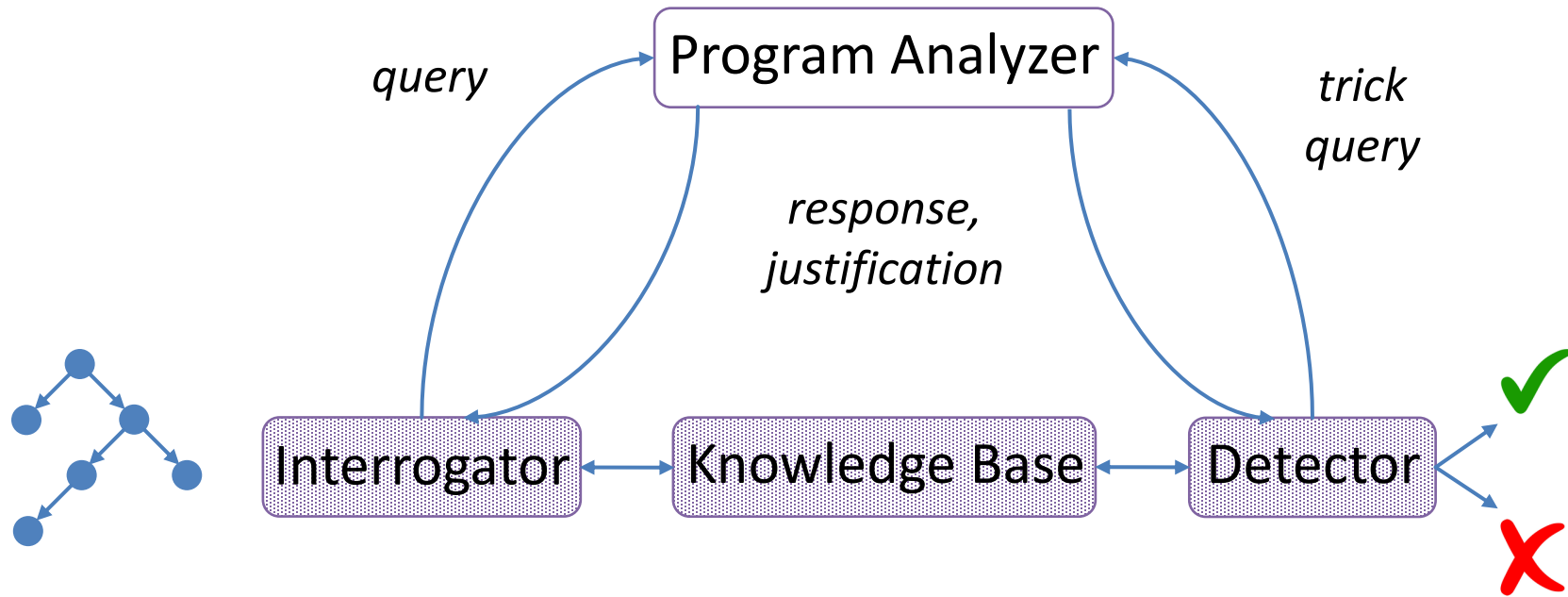
Interrogating the analyzer



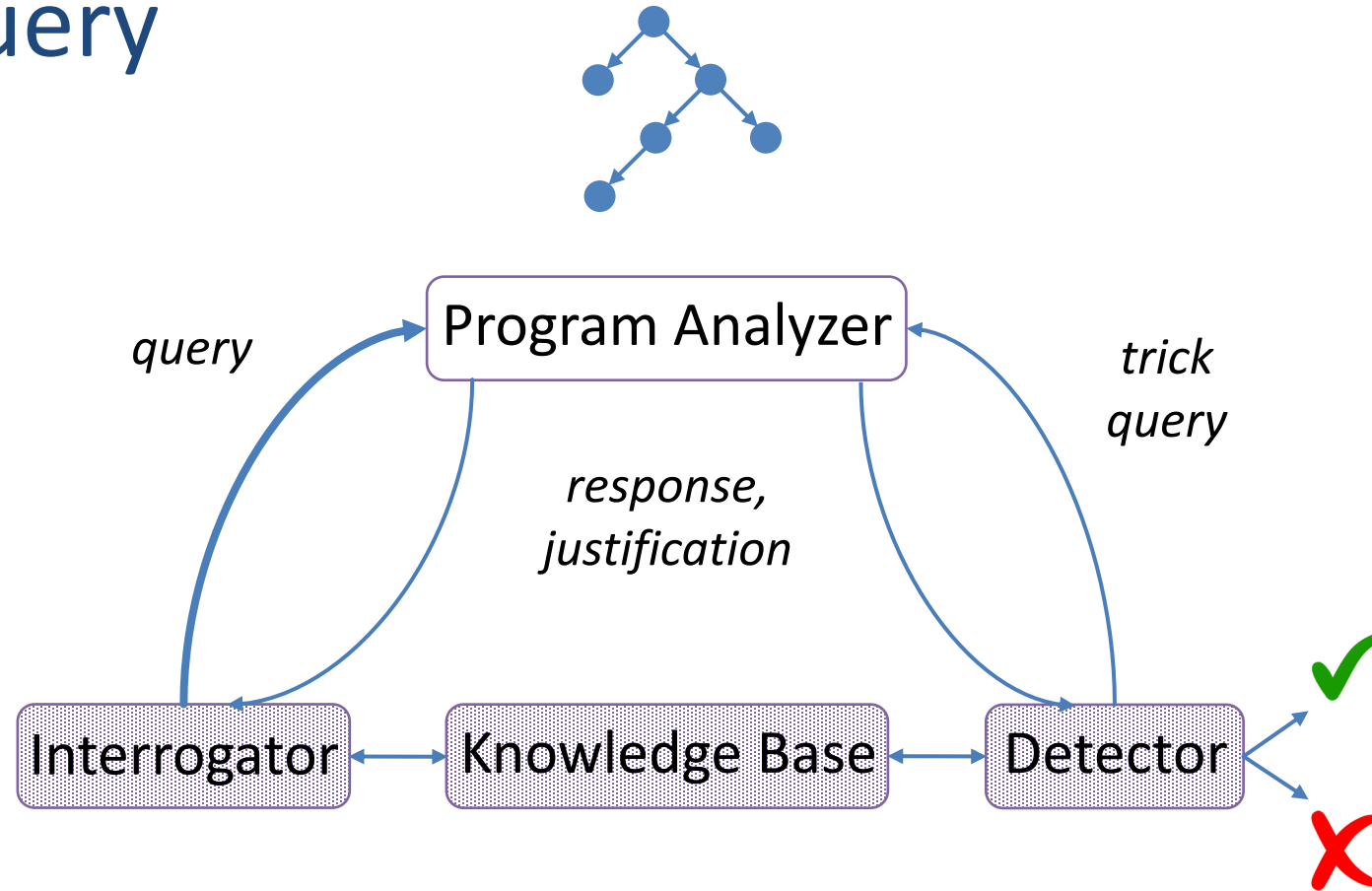
Interrogating the analyzer



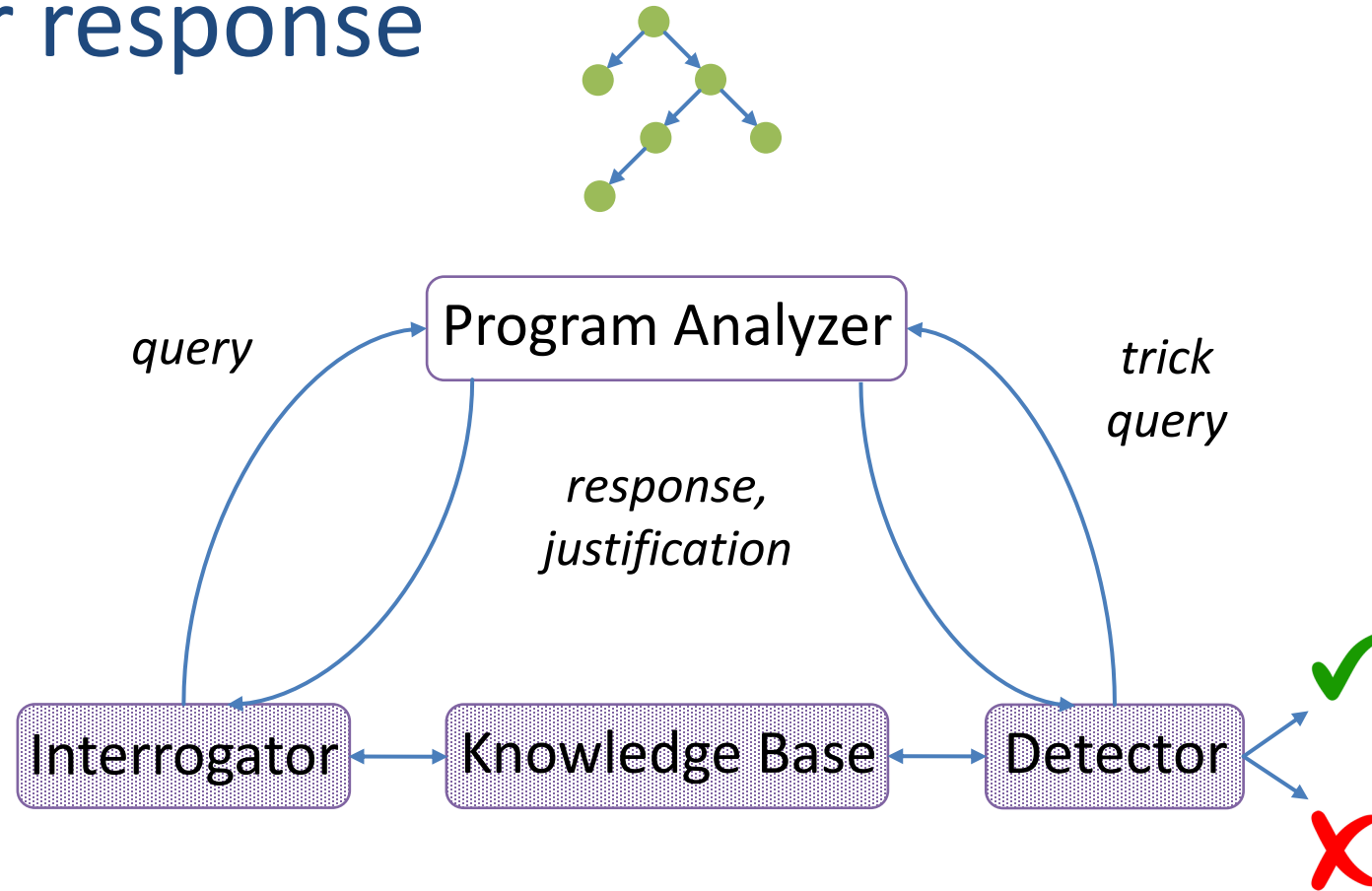
Initial query



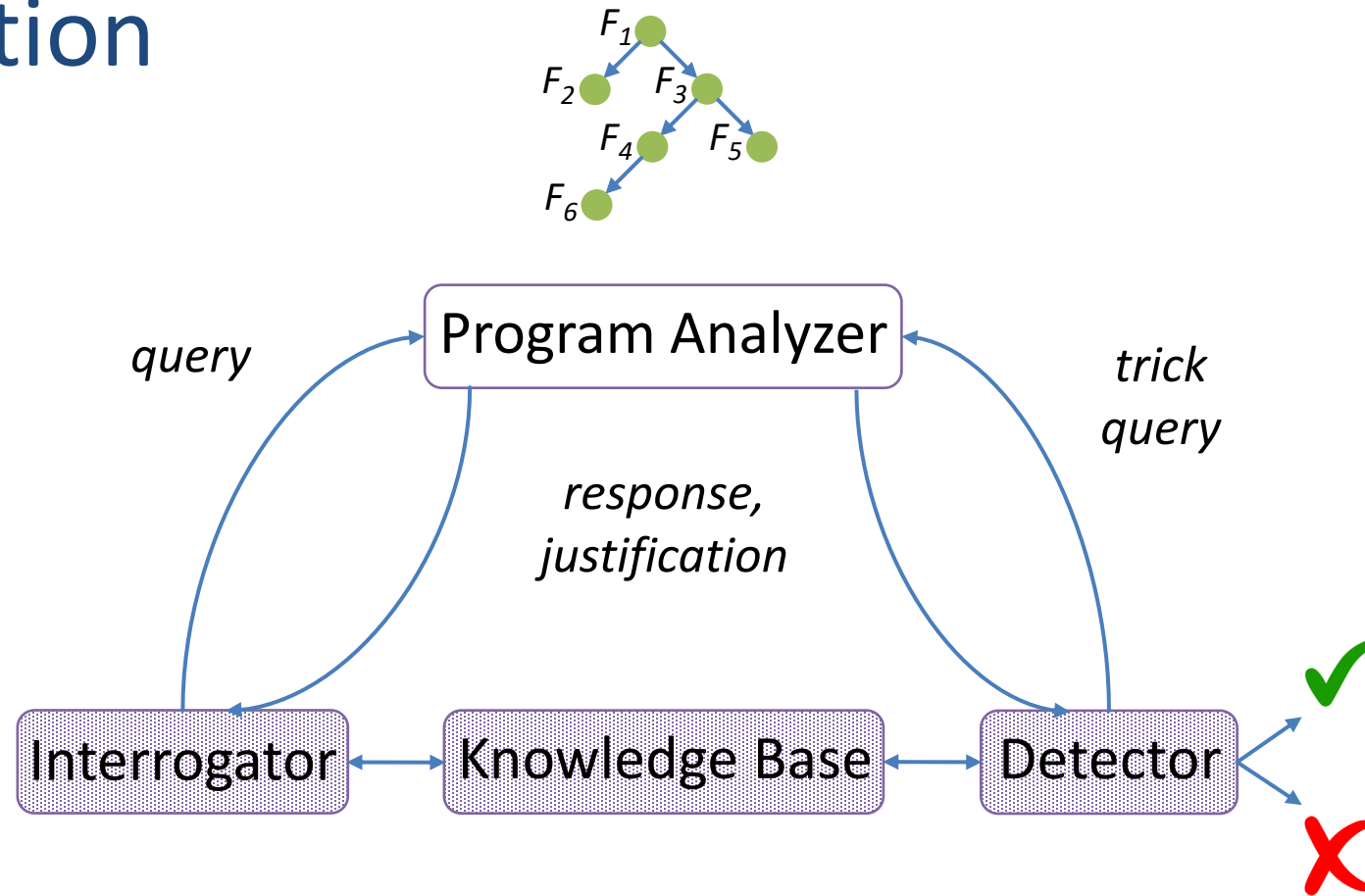
Initial query



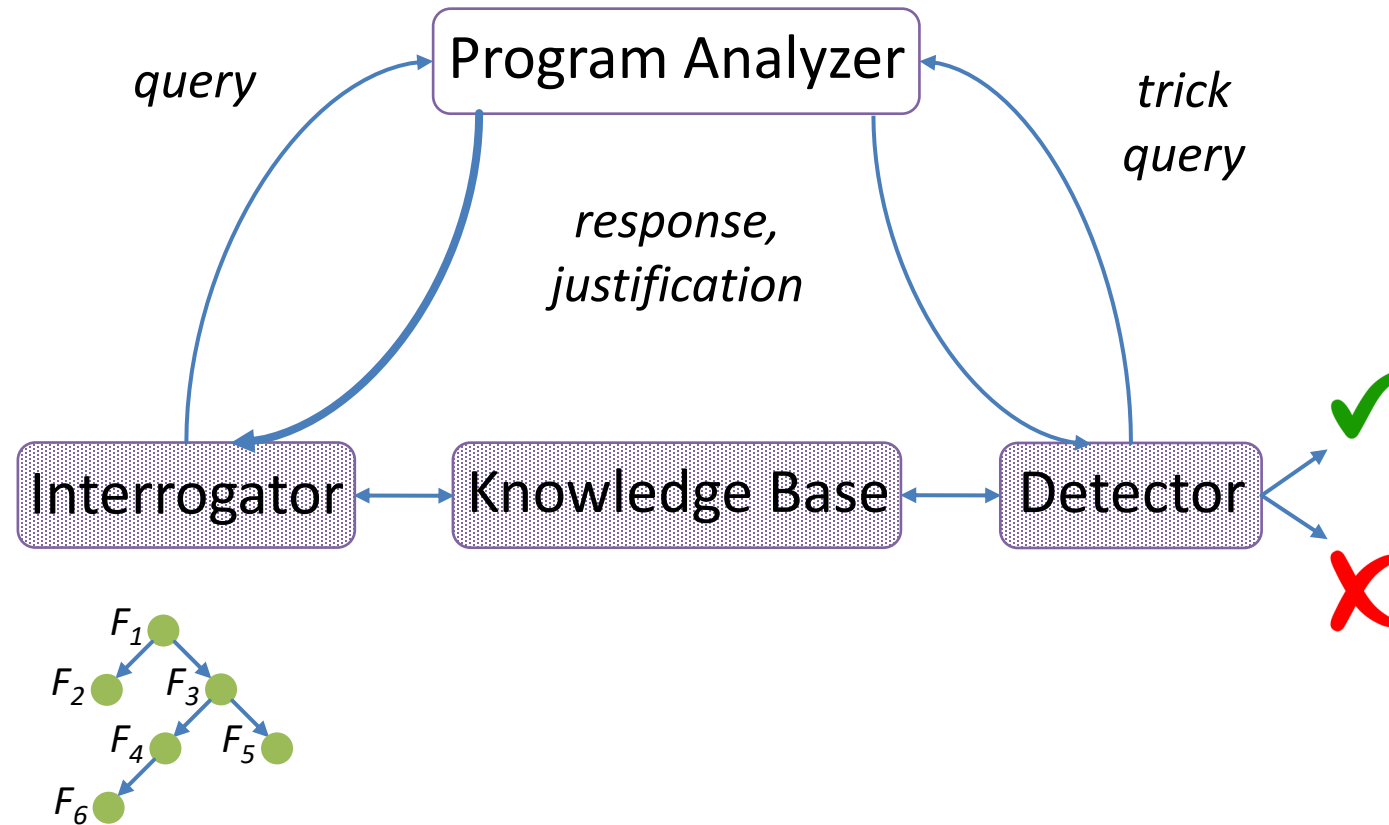
Analyzer response



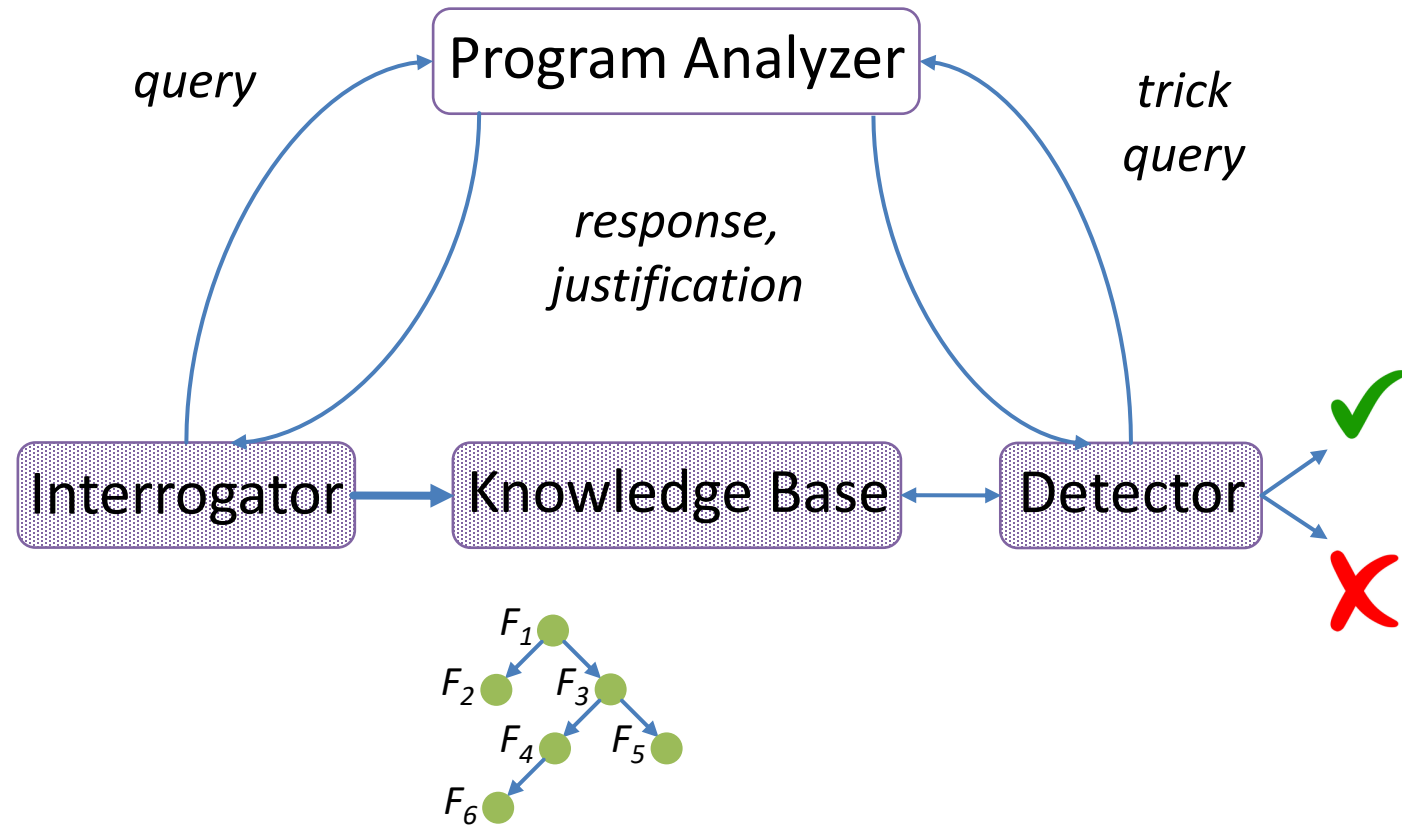
Justification



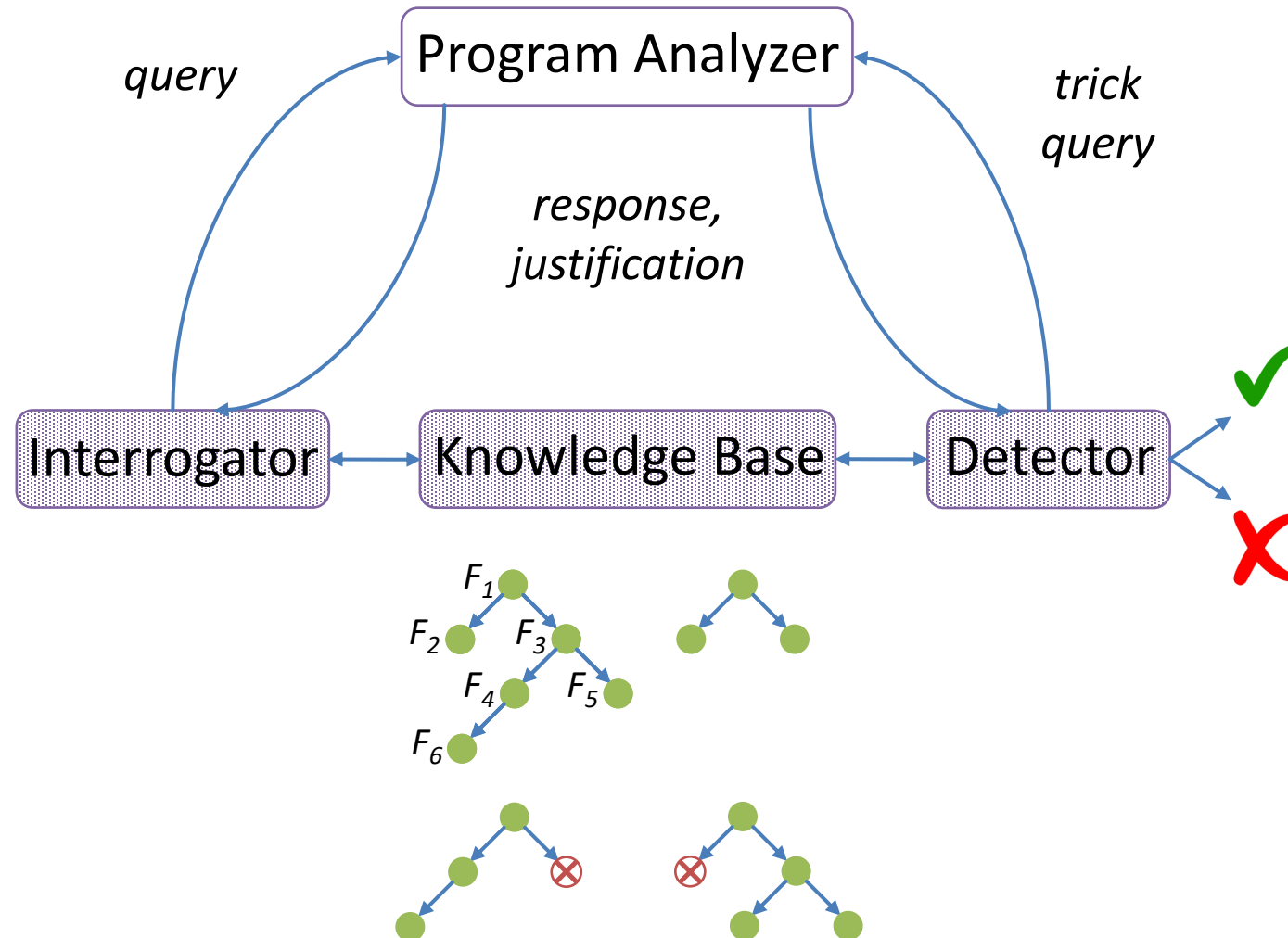
Knowledge base



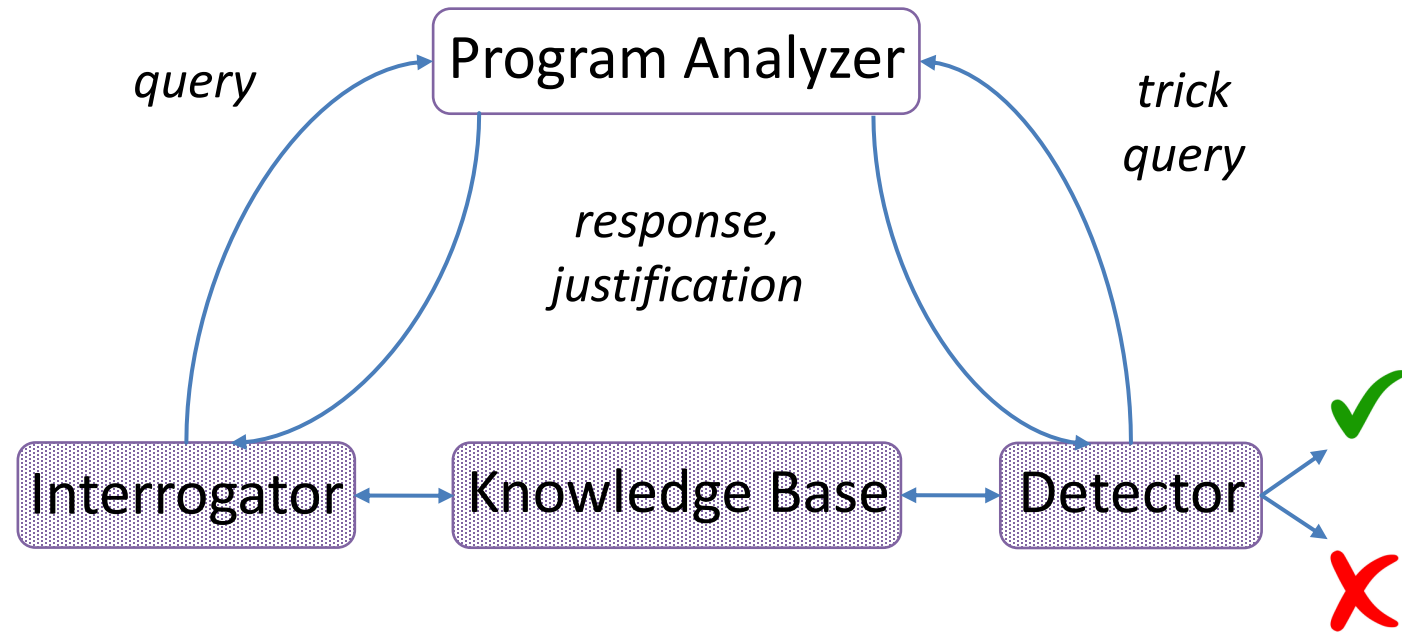
Knowledge base



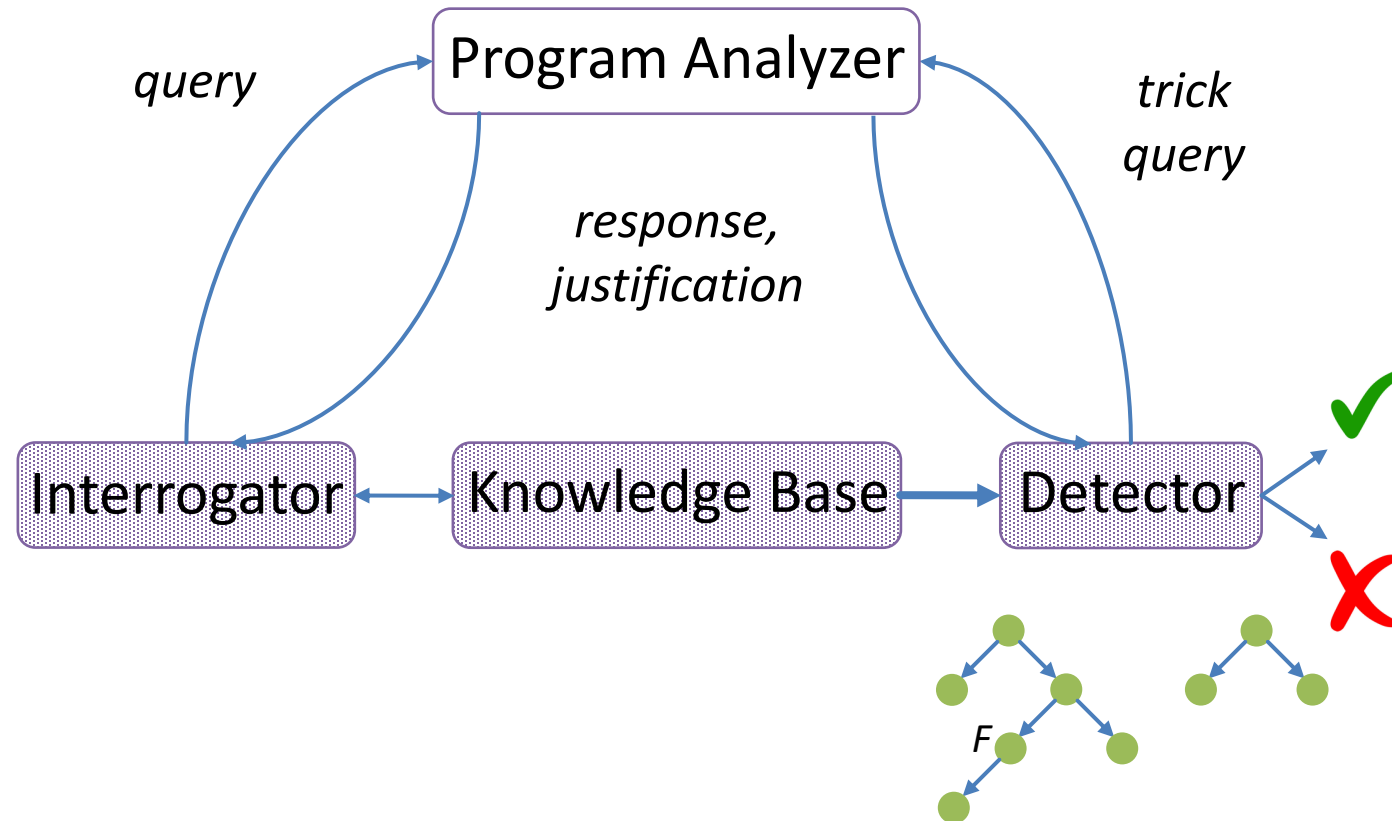
Knowledge base



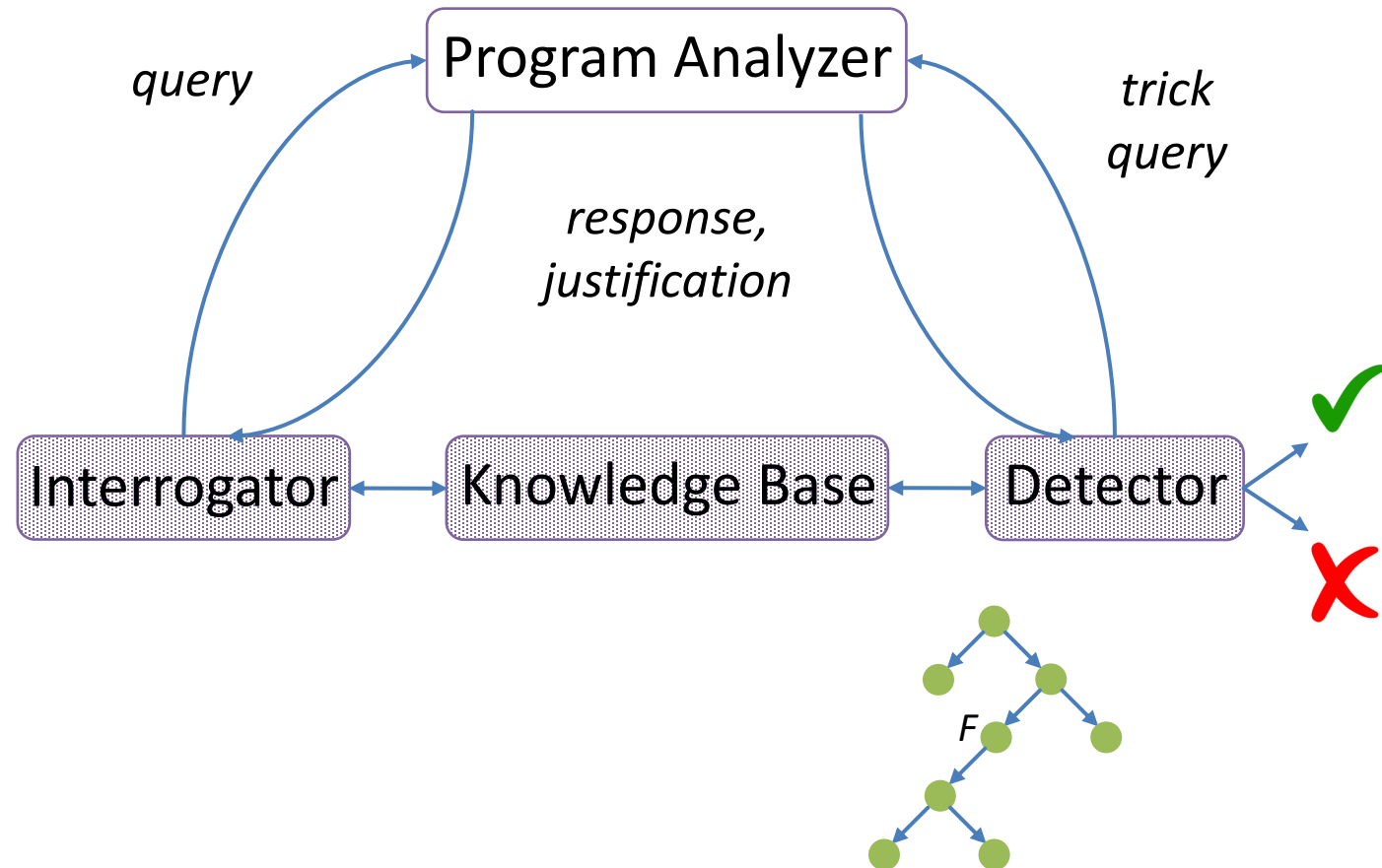
Trick query



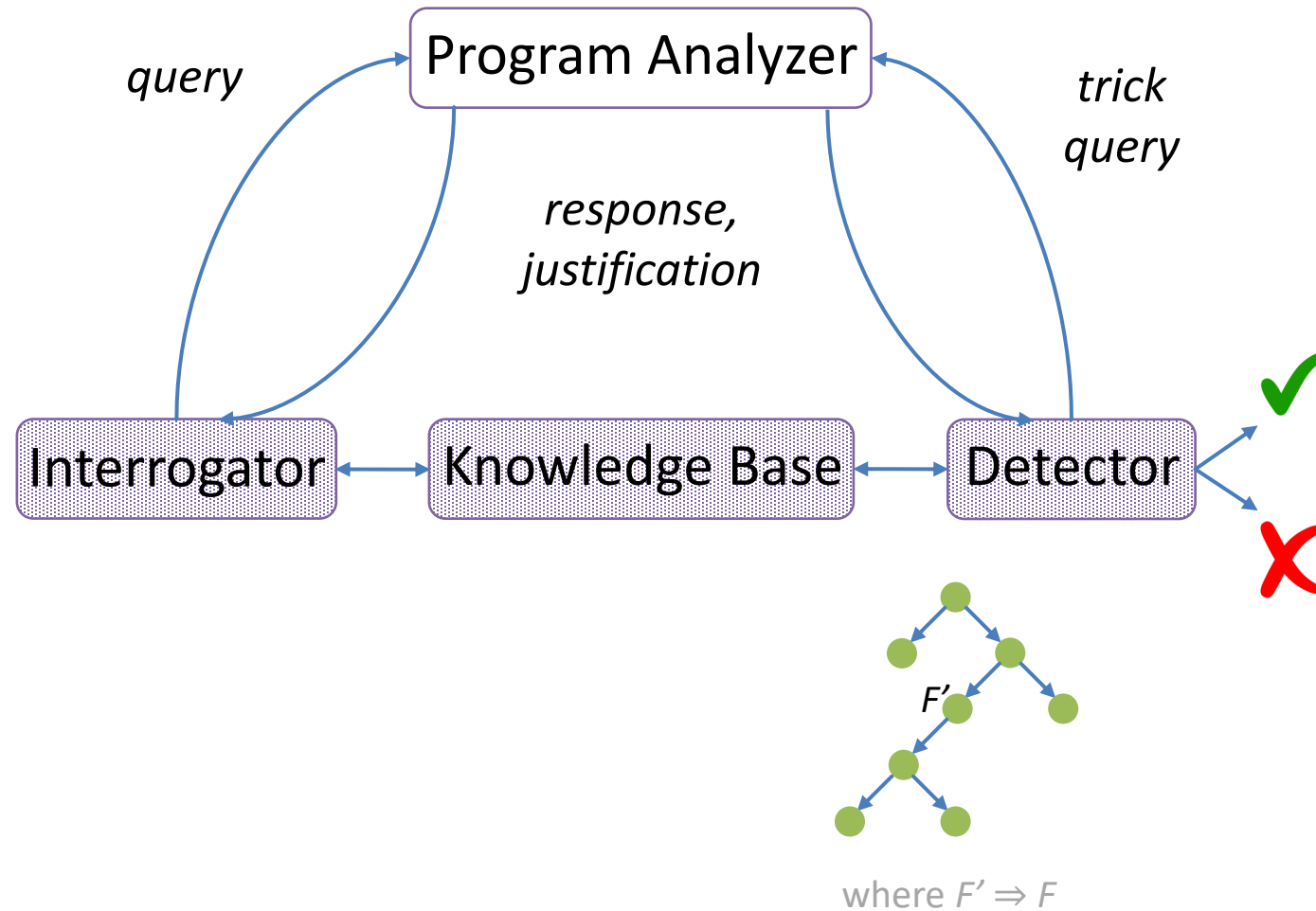
Trick query



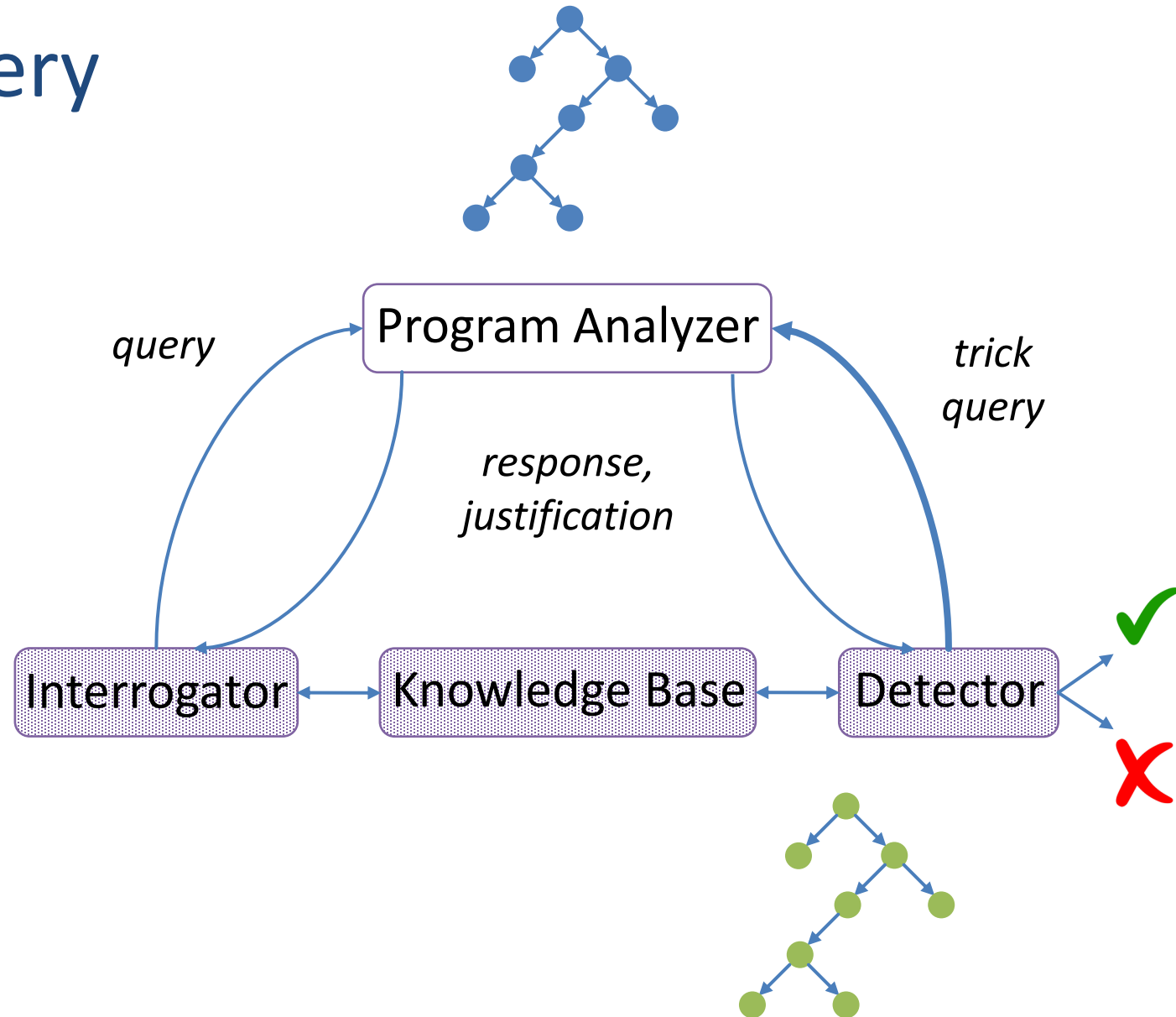
Trick query



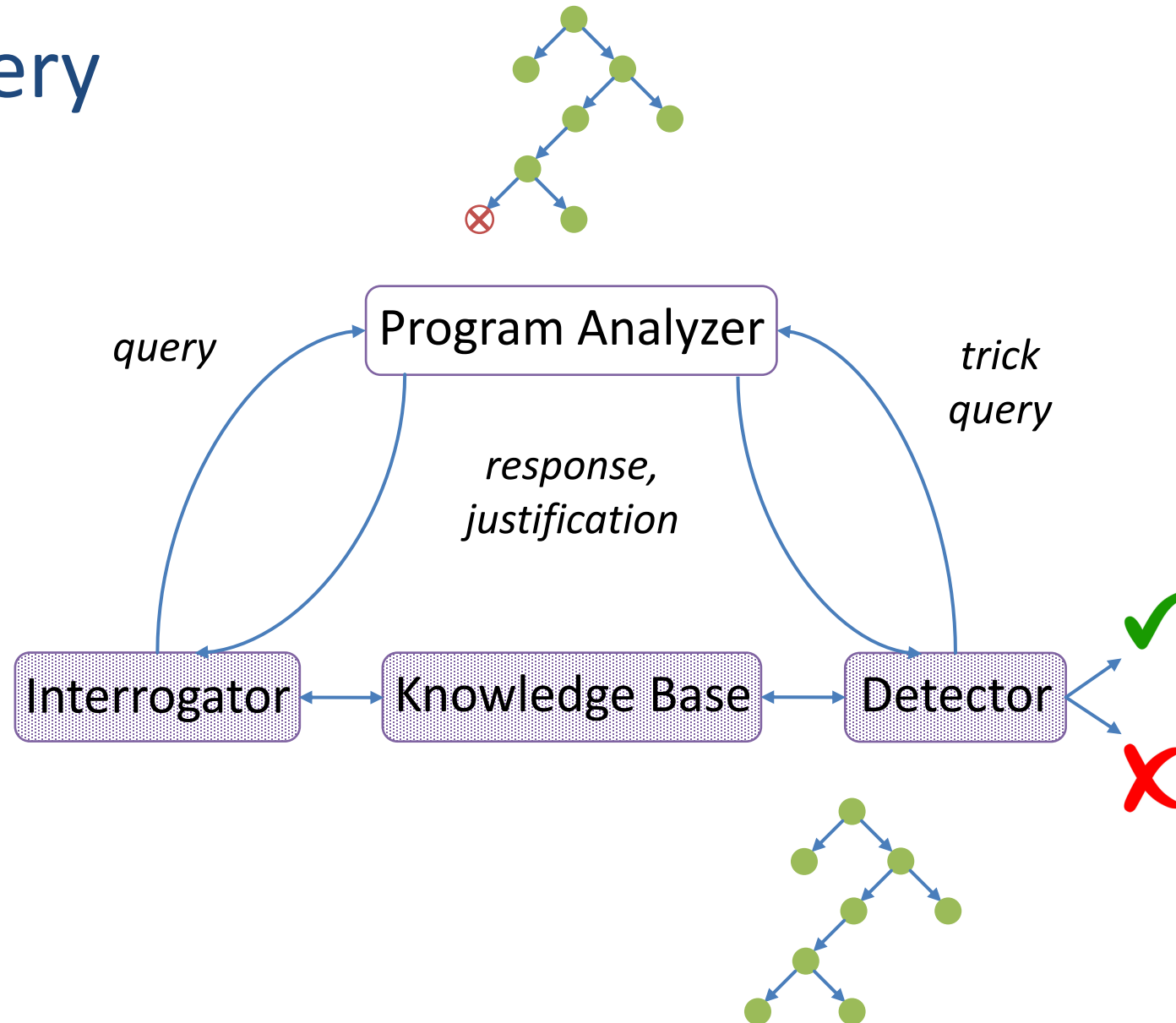
Trick query



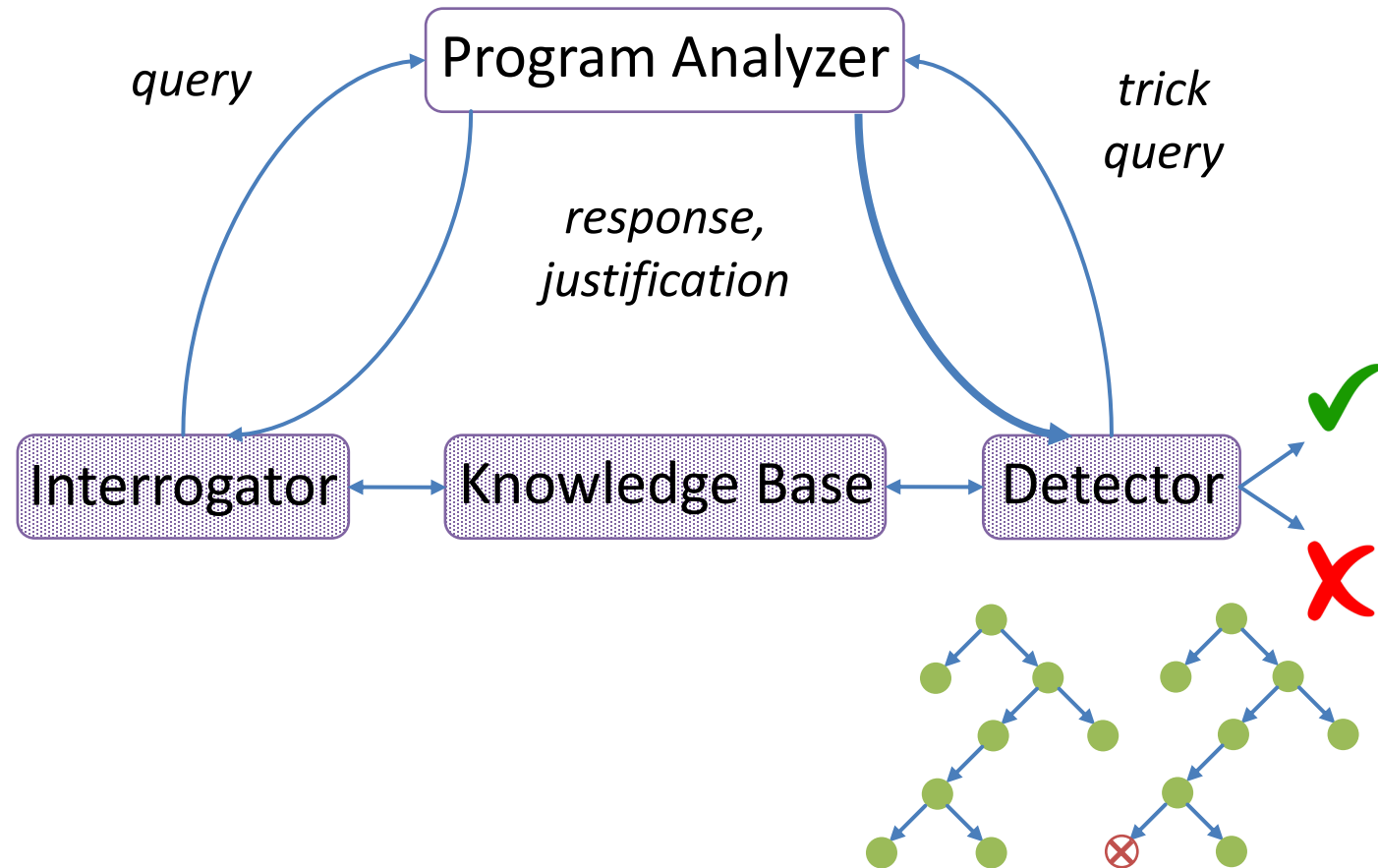
Trick query



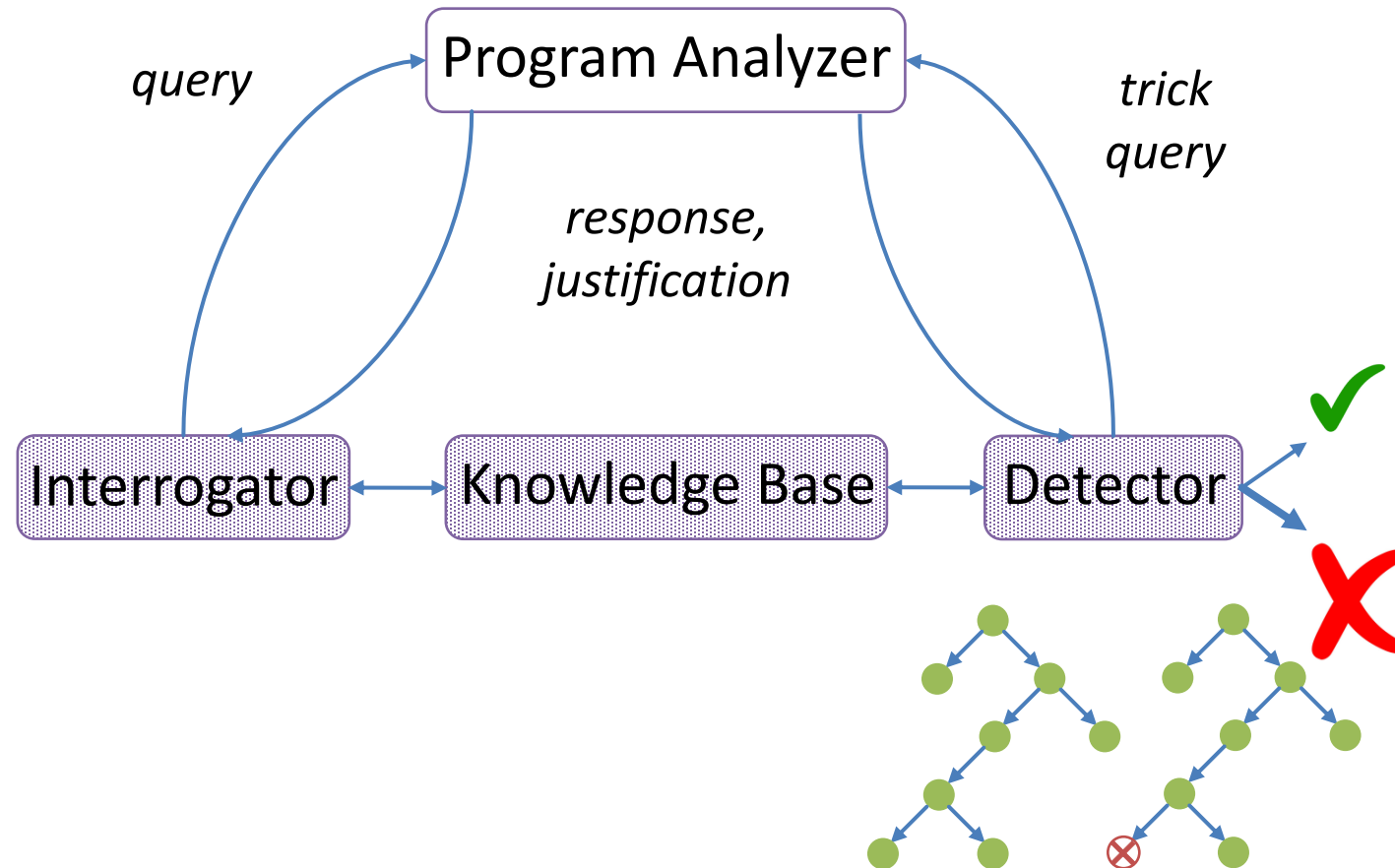
Trick query



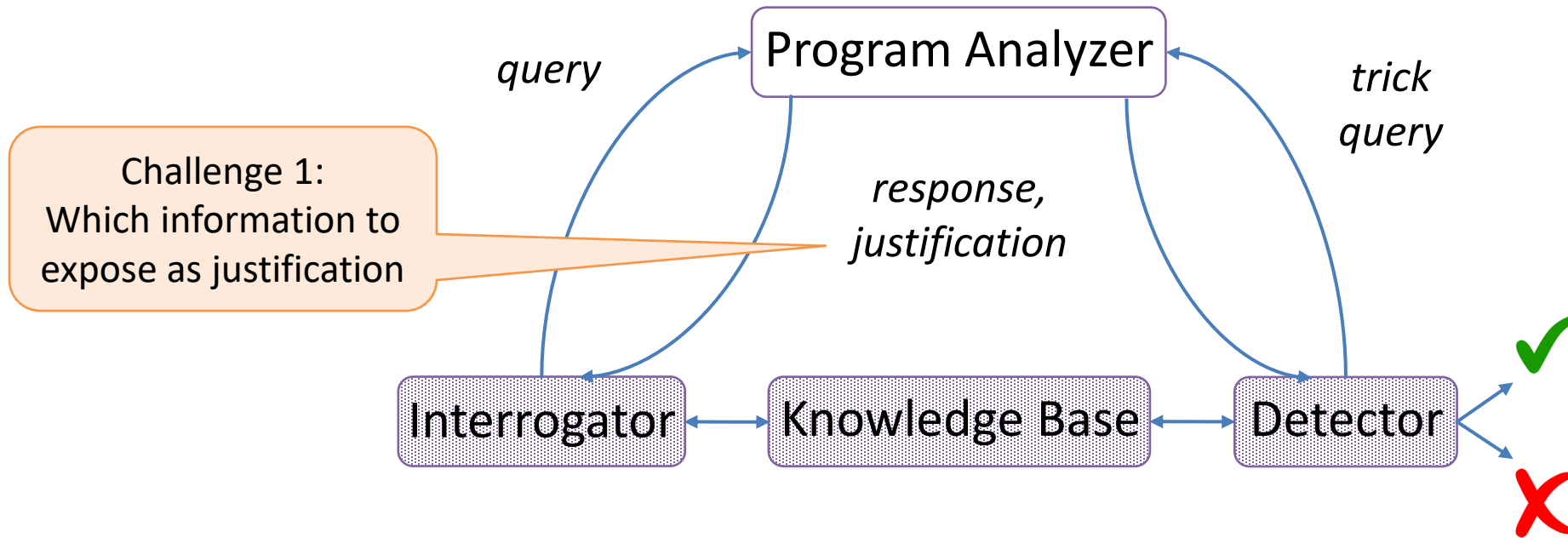
Contradiction



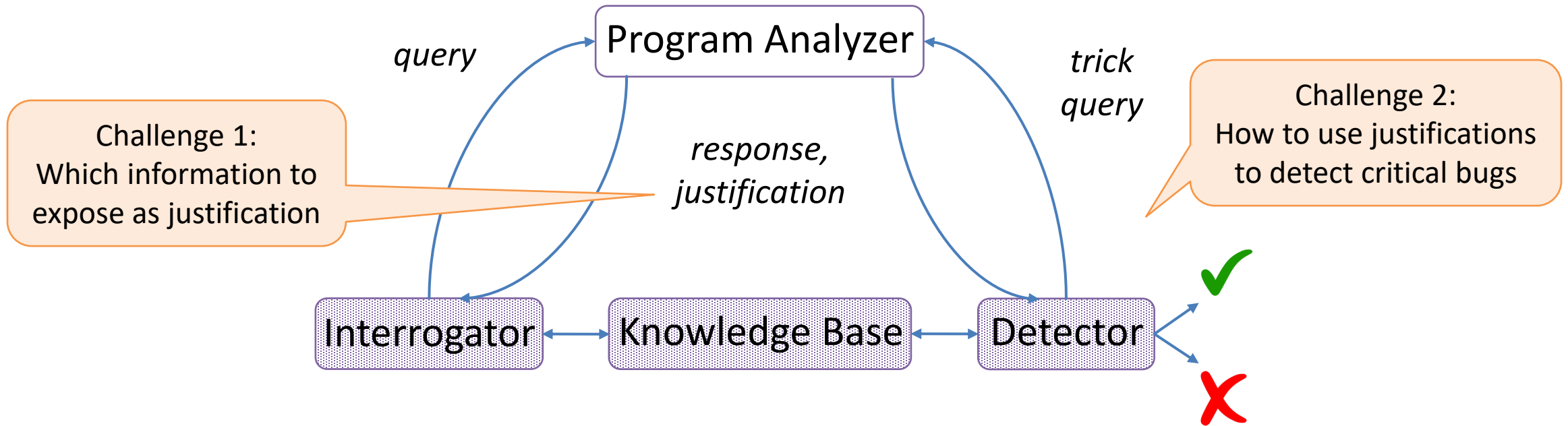
Contradiction



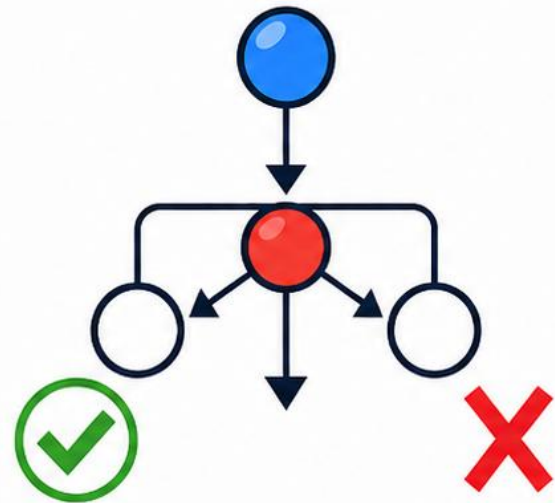
Which artifacts matter?



How can artifacts become oracles?

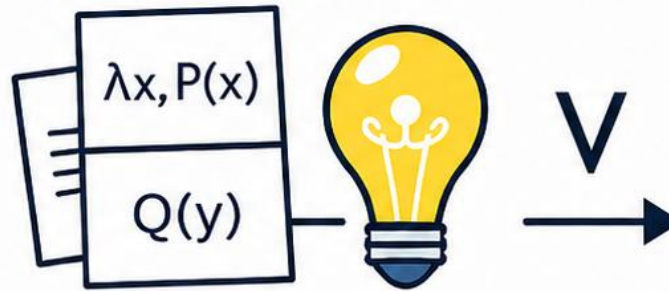


Interrogation testing across analyzer domains



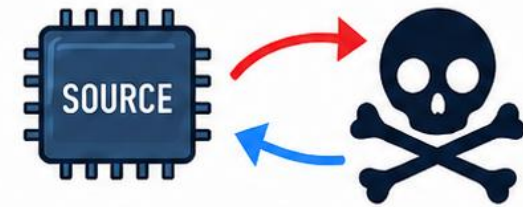
Reachability analyzers

[ASE'24]



CHC solvers

[FSE'26]



Taint analyzers

[ISSTA'26]

Sherlock: Testing reachability analyzers

Instantiating interrogation testing for reachability analyzers

Sherlock: Testing reachability analyzers

Instantiating interrogation testing for reachability analyzers



CBMC



SeaHorn/Clam



CPAchecker



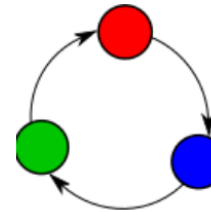
ESBMC



KLEE



MOPSA



Symbiotic



Ultimate
Automizer

Example soundness bug in CBMC

```
void main () {  
    int z = *;  
    int k = *;  
    assume(1 < z);  
    assume(1 < z && k <= 1);  
    assert(0);  
}
```

Example soundness bug in CBMC

```
void main () {  
  int z = *;  
  int k = *;  
  assume(1 < z);  
  assume(1 < z && k <= 1);  
  assert(0);  
}
```



Example soundness bug in CBMC

```
void main () {  
  int z = *;  
  int k = *;  
  assume(1 < z);  
  assume(1 <= z && k <= 1);  
  assert(0);  
}
```

Weaken
assumption

Example soundness bug in CBMC

```
void main () {  
    int z = *;  
    int k = *;  
    assume(1 < z);  
    assume(1 <= z && k <= 1);  
    assert(0);  
}
```



Example soundness bug in MOPSA

```
int main () {  
    int a = *;  
    assume(a <= 1);  
    double x = a / 2.0; ...  
    assert(x <= 0.0); ...  
}
```

Example soundness bug in MOPSA

```
int main () {  
    int a = *;  
    assume(a <= 1);  
    double x = a / 2.0; ...  
    assert(x <= 0.0); ...  
}
```



Example soundness bug in MOPSA

```
int main () {  
    int a = *;  
    assume(a <= 1 && a >= 1);  
    double x = a / 2.0; ...  
    assert(x <= 0.0); ...  
}
```

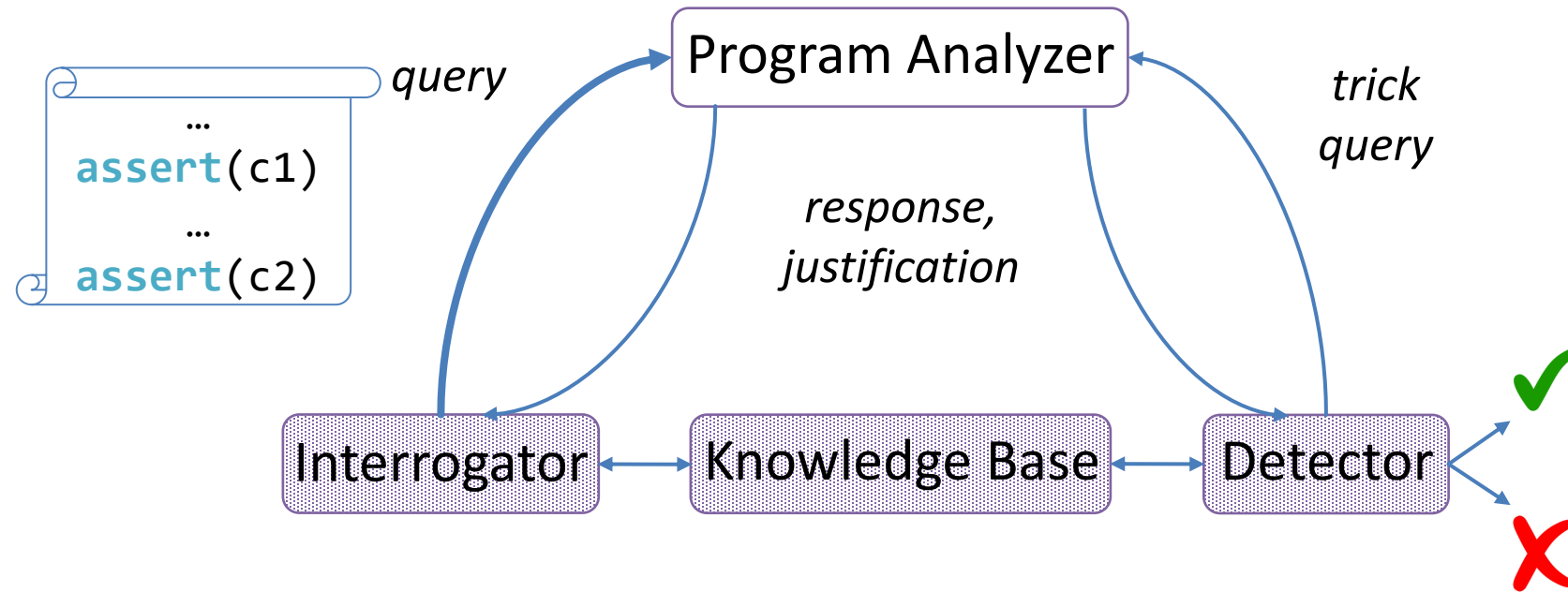
Strengthen
assumption

Example soundness bug in MOPSA

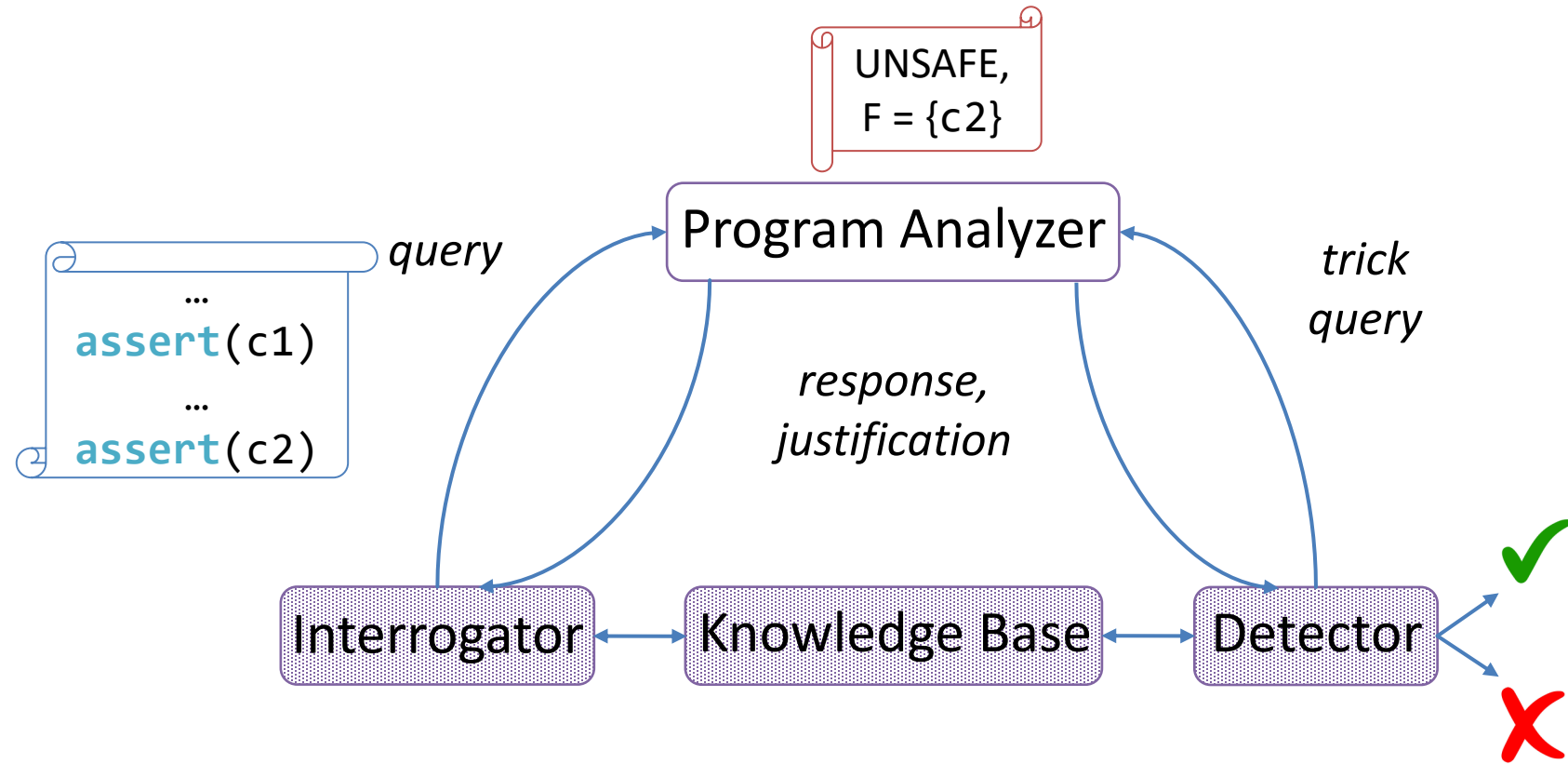
```
int main () {  
    int a = *;  
    assume(a <= 1 && a >= 1);  
    double x = a / 2.0; ...  
    assert(x <= 0.0); ...  
}
```



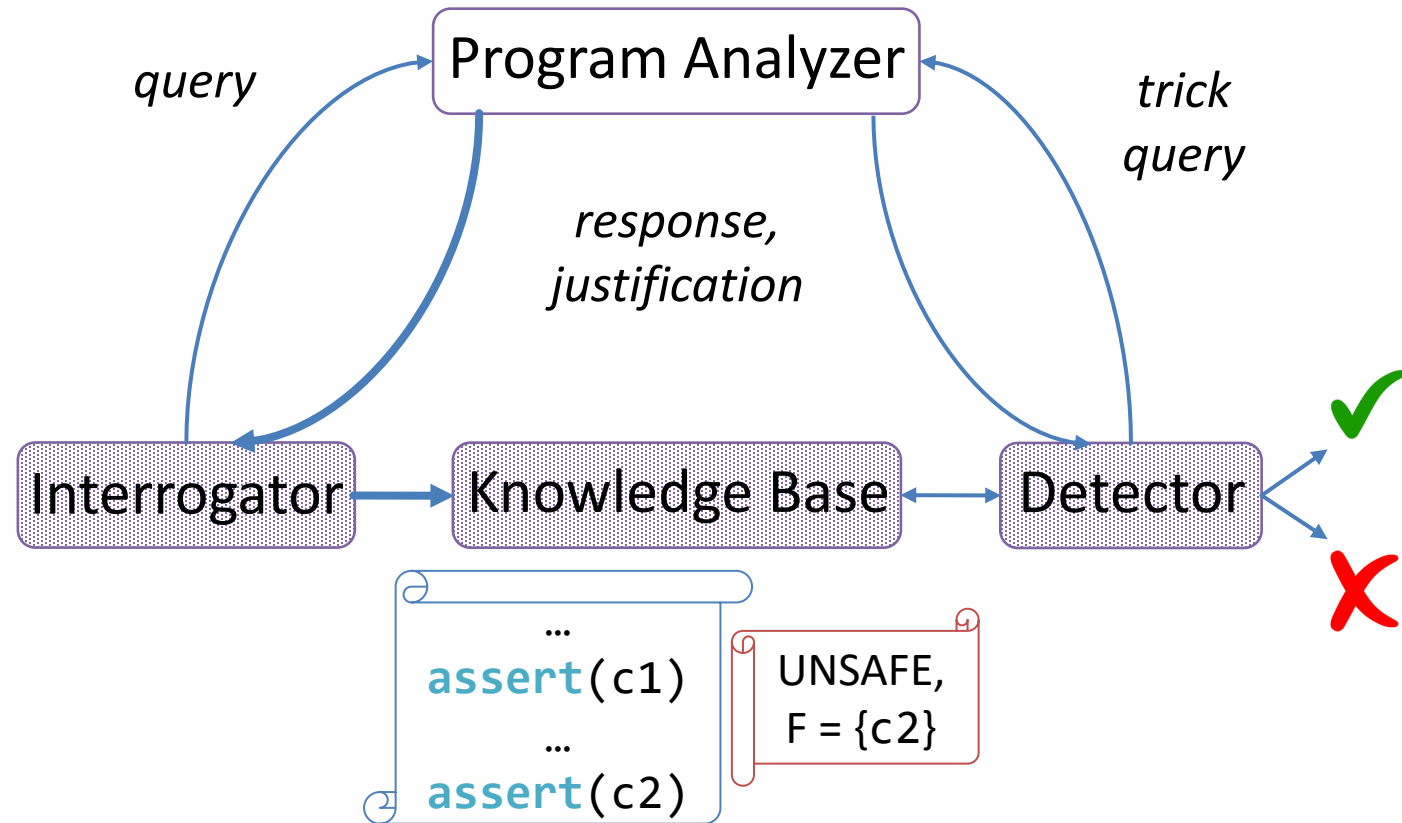
Sherlock



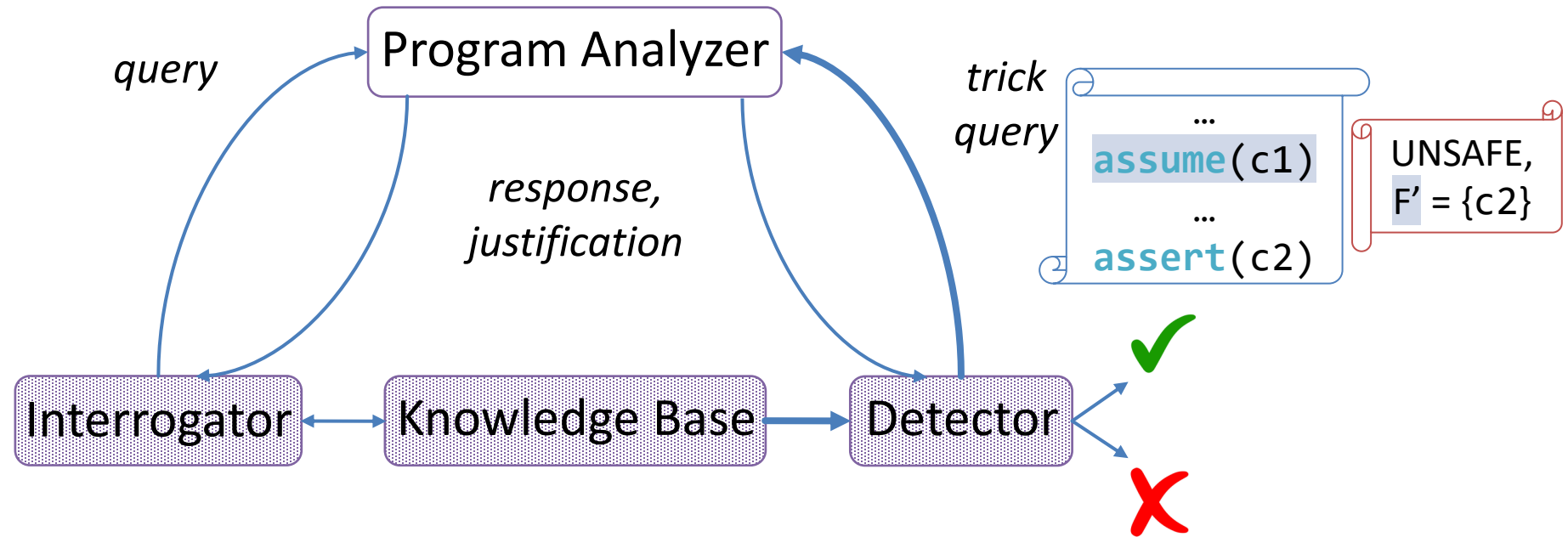
Sherlock



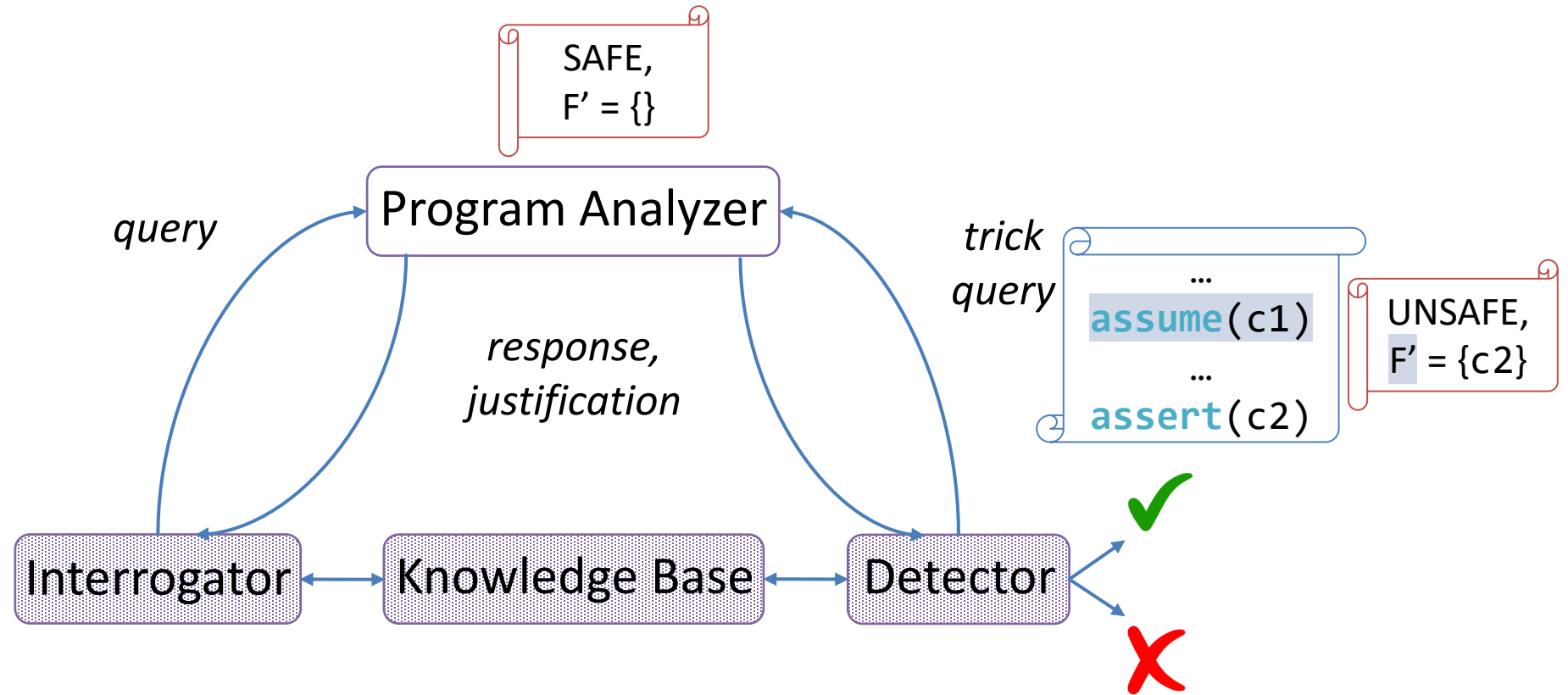
Sherlock



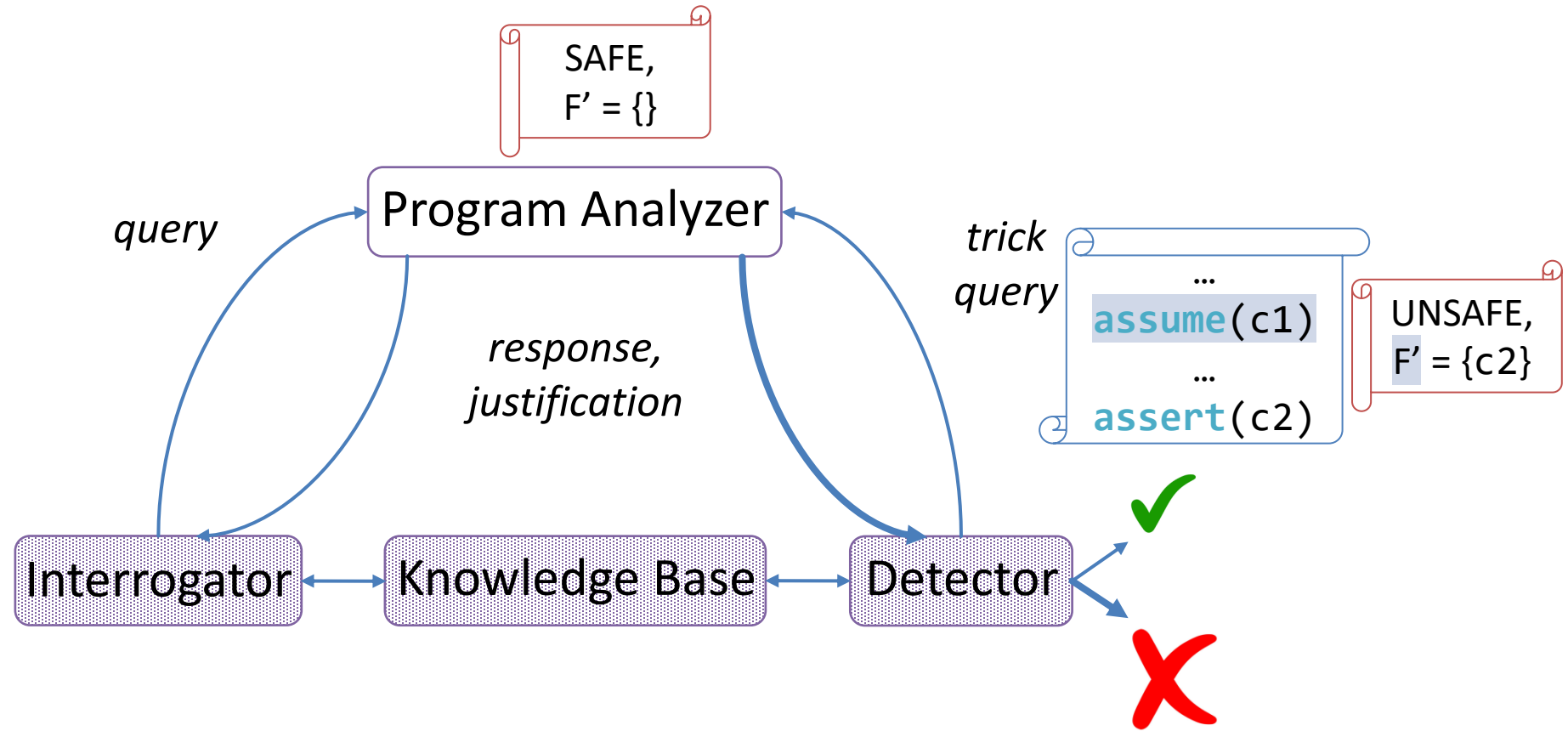
Sherlock



Sherlock



Sherlock



Sherlock: Transformations and oracles

Type	Transformation	Oracle
Strengthen	add assert add assume strengthen assert strengthen assume	$F' \setminus A_{tr} \subseteq F$
Weaken	weaken assert weaken assume	$F \setminus A_{tr} \subseteq F'$
Even	weaken invariant assume invariant	$F = F'$

Sherlock: Bugs found

Bugs found in **all** analyzers we tested

Bugs	Soundness	Precision
Fixed	9	3
Confirmed	6	3
Total	15	6

*Ignores unconfirmed and side bugs

Sherlock vs metamorphic testing

	MT	MT + J	Sherlock
Bugs (consistently detected)	60%	70%	100%
Time-to-bug slowdown	7 - 906x	7 - 20x	

* **MT:** Metamorphic testing *without* justification

MT + J: Metamorphic testing *with* justification

Sherlock: MT + J *with* knowledge base

HornGator: Testing CHC solvers

Instantiating interrogation testing for CHC solvers

HornGator: Testing CHC solvers

Instantiating interrogation testing for **CHC solvers**



Eldarica

Golem

Golem



Z3/Spacer



Theta



Ultimate Tree
Automizer

Example bug in Golem

$$\begin{aligned}(x \wedge y) \wedge P(1) &\rightarrow P(4) \\ P(2) &\rightarrow \perp\end{aligned}$$

Example bug in Golem

$$(x \wedge y) \wedge P(1) \rightarrow P(4)$$
$$P(2) \rightarrow \perp$$

↓ SAT-to-SAT transformation

$$(x \wedge y) \wedge \neg x \wedge P(1) \rightarrow P(4)$$
$$P(2) \rightarrow \perp$$

Example bug in Golem

$$(x \wedge y) \wedge P(1) \rightarrow P(4)$$
$$P(2) \rightarrow \perp$$



SAT-to-SAT transformation

$$(x \wedge y) \wedge \neg x \wedge P(1) \rightarrow P(4)$$
$$P(2) \rightarrow \perp$$

Golem says
UNSAT

Example transformation with model

$$x \leq 5 \wedge P(x, y) \wedge Q(y) \rightarrow R(x)$$

where $M(R) = x \geq 0$

Example transformation with model

$$x \leq 5 \wedge P(x, y) \wedge Q(y) \rightarrow R(x)$$

$$\text{where } M(R) = x \geq 0$$

↓ SAT-to-SAT transformation

$$x \leq 5 \wedge P(x, y) \wedge Q(y) \wedge \neg(x \geq 0) \rightarrow \perp$$

Example transformation with refutation proof

$$x \leq 0 \rightarrow P(x) \quad (1)$$

$$x < 2 \wedge P(x) \rightarrow P(x + 1) \quad (2)$$

$$x \geq 2 \wedge P(x) \rightarrow \perp \quad (3)$$

Example transformation with refutation proof

$$x \leq 0 \rightarrow P(x) \quad (1)$$

$$x < 2 \wedge P(x) \rightarrow P(x + 1) \quad (2)$$

$$x \geq 2 \wedge P(x) \rightarrow \perp \quad (3)$$

Refutation proof: $\top \xrightarrow{(1)} P(0) \xrightarrow{(2)} P(1) \xrightarrow{(2)} P(2) \xrightarrow{(3)} \perp$

Example transformation with refutation proof

$$x \leq 0 \rightarrow P(x) \quad (1)$$

$$x < 2 \wedge P(x) \rightarrow P(x + 1) \quad (2)$$

$$x \geq 2 \wedge P(x) \rightarrow \perp \quad (3)$$

Refutation proof: $\top \xrightarrow{(1)} P(0) \xrightarrow{(2)} P(1) \xrightarrow{(2)} P(2) \xrightarrow{(3)} \perp$

$$\top \rightarrow P(0)$$

$$x < 2 \wedge P(x) \rightarrow P(x + 1)$$

$$x \geq 2 \wedge P(x) \rightarrow \perp$$

UNSAT-to-UNSAT
transformation

HornGator: Bugs found

Bugs found in **all** solvers we tested

Bugs	Soundness	Precision	Wrong artifact
Fixed	6	10	2
Confirmed	2	0	1
Total	8	10	3

*Ignores unconfirmed and side bugs

HornGator vs metamorphic testing

	MT	MT + J	MT + KB	HornGator
Bugs (consistently detected)	56%	81%	81%	100%
Time-to-bug slowdown	>1 - 42x	>1 - 14x	>1 - 6x	

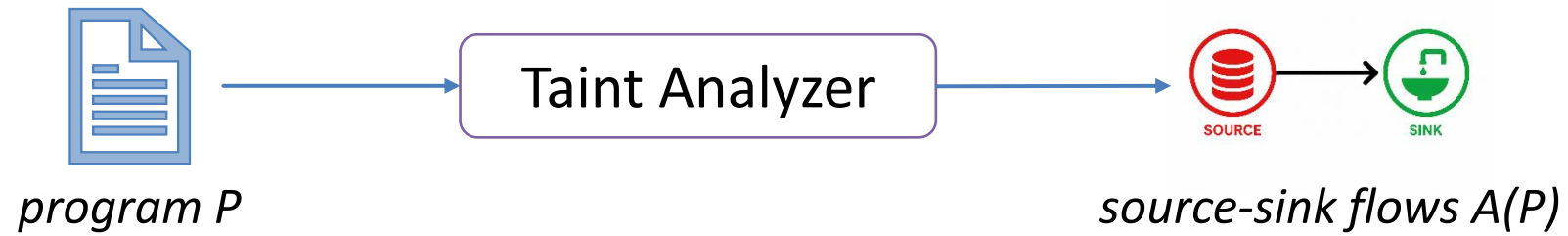
- * **MT:** Metamorphic testing *without* justification or knowledge base
- MT + J:** Metamorphic testing *with* justification *without* knowledge base
- MT + KB:** Metamorphic testing *with* knowledge base *without* justification
- HornGator:** Metamorphic testing *with* justification and knowledge base

How weak can the input relation be?

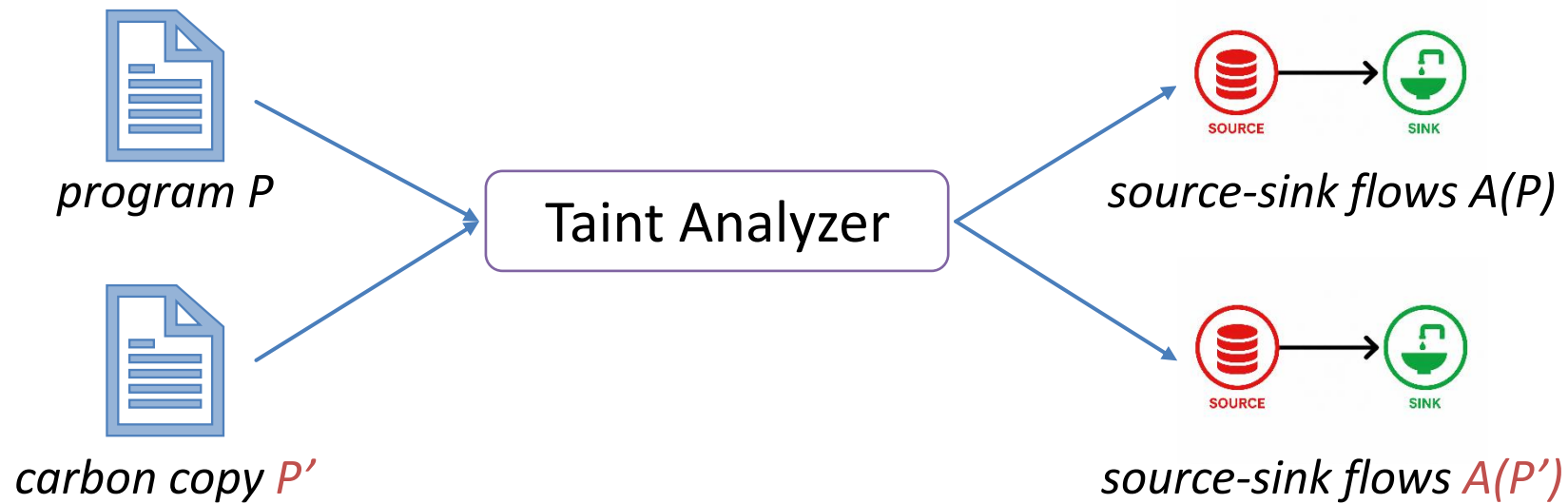
It need only preserve what the analyzer claims

- Semantic equivalence is often stronger than necessary
- Many analyzers make narrower claims
- A weaker relation can preserve only the property relevant to the claim
- Weaker relations enable more diverse follow-up queries

Taint analyzers

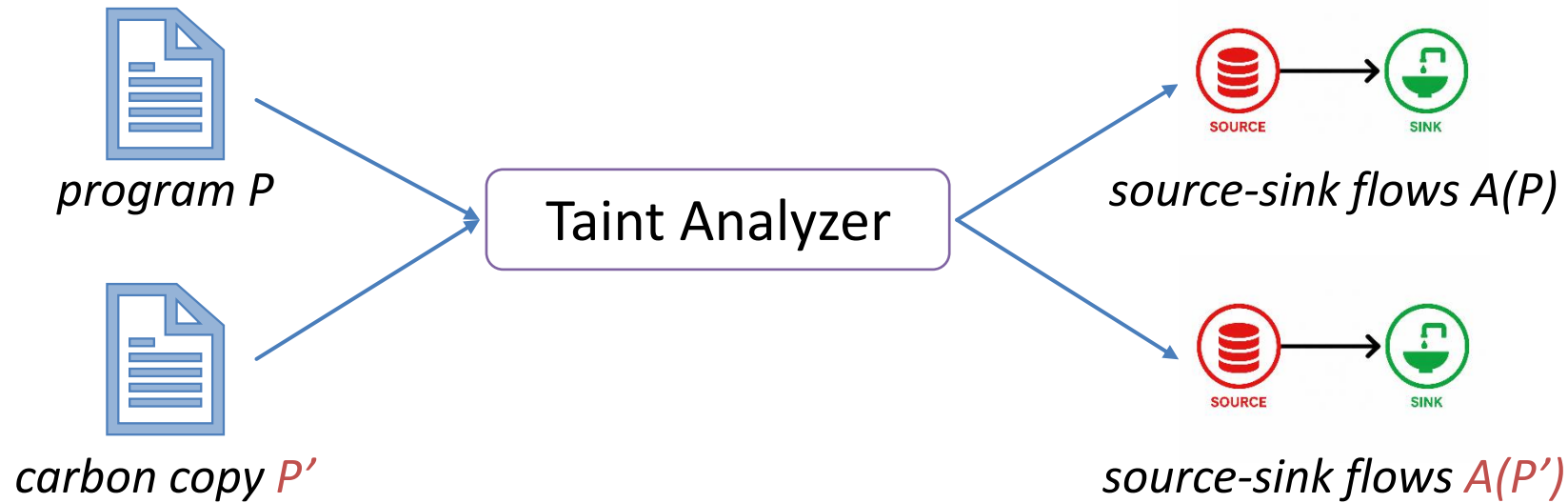


TaintCC: Equivalence modulo taint



- P' is a carbon copy of P by preserving its source-sink flows: $P \equiv_{EMT} P'$

TaintCC: Equivalence modulo taint



- P' is a carbon copy of P by preserving its source-sink flows: $P \equiv_{EMT} P'$
- The **oracle** for correct analyzer behavior is $A(P) = A(P')$

EMT transformations

TaintCC generates carbon copies using transformations with [different strengths](#)

Type	What it preserves
Semantically equivalent	Full program behavior
Taint oblivious	Taint behavior by construction
Taint aware	Reported flows using analyzer feedback

Semantically equivalent transformations

```
s = source()  
t = 0  
if x < 42:  
    t = s  
sink(t)
```

$\equiv_{EMT}$

```
s = source()  
t = 0  
if x < ld(42):  
    t = s  
sink(t)
```

- Preserve program behavior, and therefore, source-sink flows
- Useful for diversity
- Cannot exploit taint-specific flexibility

Taint-oblivious transformations

```
s = source()  
t = 0  
if x < 42:  
    t = s  
sink(t)
```

$\equiv_{EMT}$

```
s = source()  
t = 0  
if x < 42:  
    t = s + 77  
sink(t)
```

- May change semantics
- Do not use the analyzer's reported flows
- Preserve taint behavior by construction

Taint-aware transformations

```
s = source ()  
t = 0  
if x < 42:  
    t = s  
  
sink(t)  
u = x + s  
sink(u)
```

$\equiv$ *EMT*

```
s = source ()  
t = 0  
if x < 42:  
    t = s  
    v = t  
sink(t)  
u = x + s  
sink(v)
```

- May change semantics
- Use the analyzer's reported flows or witness paths
- Choose mutations based on analyzer feedback

Taint-aware transformations

```
s = source ()  
t = 0  
if x < 42:  
    t = s  
    sink(t)  
    t = t + 1
```

$\equiv$ *EMT*

```
s = source ()  
t = 0  
if x < 42:  
    t = s  
    sink(t)  
t = t + 1
```

- May change semantics
- Use the analyzer's reported flows or witness paths
- Choose mutations based on analyzer feedback

TaintCC: Testing taint analyzers

TaintCC instantiates interrogation testing and EMT
for mature Java and Python **taint analyzers**



FlowDroid



Mariana
Trench



Pysa



Semgrep

Example soundness bug in FlowDroid

```
List<String> taint_list = new ArrayList<String>();  
taint_list.add(this.source());
```



```
Integer dummy_counter = -1;  
List<Boolean> unused_list = new ArrayList<Boolean>();
```

```
this.sink(taint_list);
```



Example soundness bug in FlowDroid

```
List<String> taint_list = new ArrayList<String>();  
taint_list.add(this.source());
```



```
Integer dummy_counter = 0;
```

```
List<Boolean> unused_list = new ArrayList<Boolean>();
```

```
this.sink(taint_list);
```



TaintCC: Bugs found

Bugs found in [all](#) analyzers we tested

Bugs	Soundness	Precision
Fixed	3	2
Confirmed	5	6
Total	8	8

*Ignores unconfirmed and side bugs

TaintCC: Why weakening matters

Semantic equivalence is too conservative

- Semantically equivalent transformations preserve EMT automatically
- But preserving full behavior leaves less room to stress taint reasoning
- Taint-specific transformations reach behaviors semantic equivalence cannot

TaintCC: Differential testing is too noisy

Differential testing finds many disagreements, but too many to triage

- Taint analyzers differ in languages, models, configurations, and precision choices
- Disagreement may reflect a real bug or an intentional modeling difference
- EMT violations are self-contradictions within one analyzer

There is no universal oracle

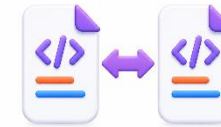
Oracle design is not one-size-fits-all!



Differential testing:
the oracle comes from disagreement



Program generation:
the oracle is built into the query



Metamorphic testing:
the oracle comes from a relation



Specification-based testing:
the oracle is the specification



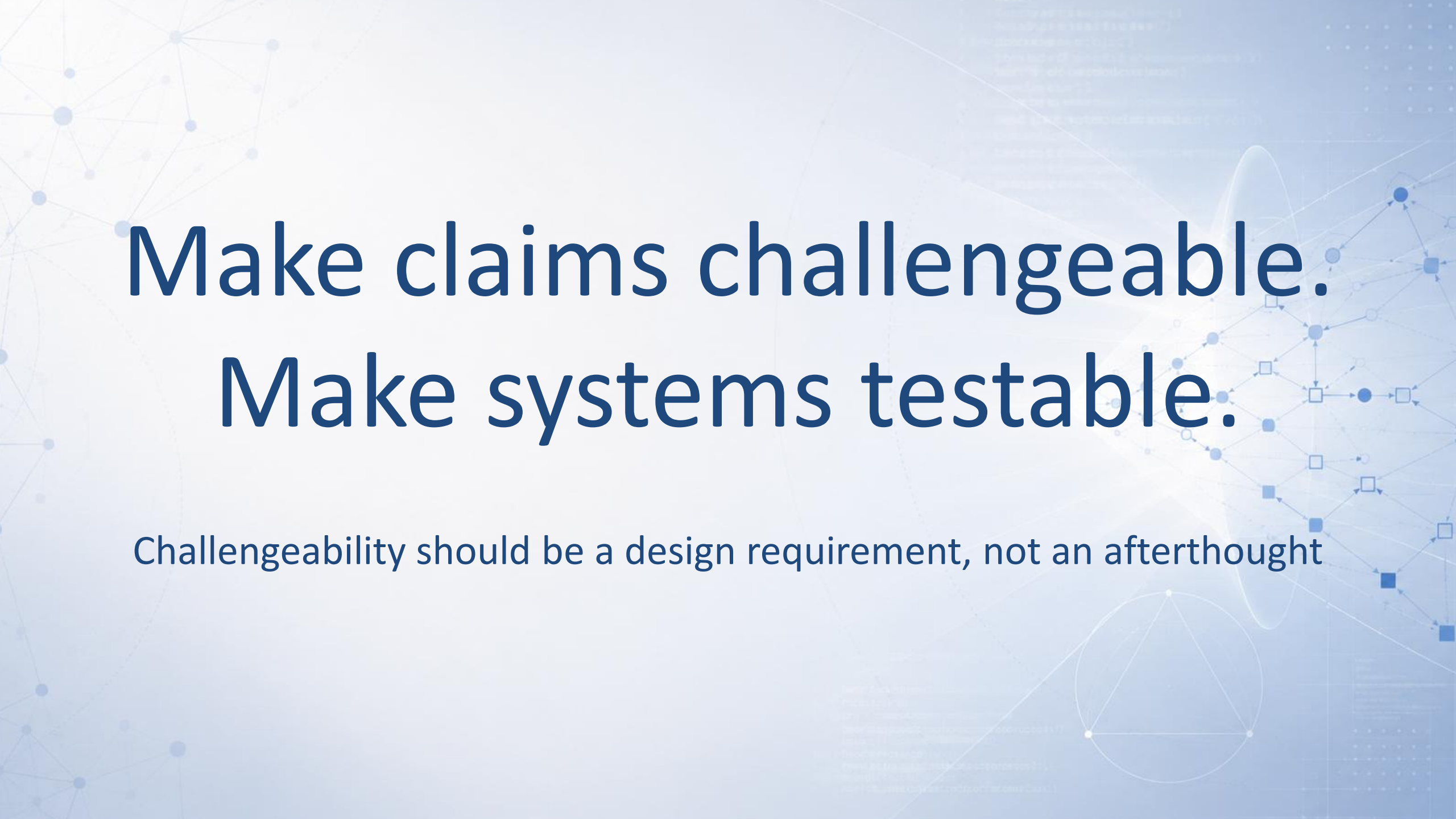
Interrogation testing:
the oracle becomes adaptive

The same question keeps coming back

Different systems make different claims,
but every claim needs a way to become checkable

- Zero-knowledge systems [CCS'25, USENIX Security'26]
 - Does the proof correspond to the intended computation?
- Multi-party computation compilers [FSE'26]
 - Does the compiled protocol preserve the source program?
- Machine-learning models [ISSTA'22, IJCAI'23, AAAI'24, IJCAI'25]
 - Does the prediction respect expected behavior?

Different domains, same oracle problem

The background features a light blue gradient with abstract network diagrams. On the left, a cluster of nodes is connected by lines. On the right, a more complex network with various node shapes (circles, squares) and arrows is visible. Faint, illegible code snippets are scattered in the upper right and lower right areas.

Make claims challengeable. Make systems testable.

Challengeability should be a design requirement, not an afterthought