Grammar-Based Fuzzing + Bug Explanations

Software Engineering II

04.05.2025

Martin Eberlein
Humboldt-Universität zu Berlin, Germany

Agenda

Fuzzing 101
Automatic Input Generation

Grammar based Fuzzing
Advanced Input Generation

Automatic Debugging
Automatically Explaining the Circumstances of Program Faults

Martin Eberlein, Doctoral Researcher, SE 2 (2025)

Grammar-Based Fuzzing

Quality Assurance

Basic Fuzzing

Generating Program Input 101

- **Software Testing Technique**
  Many different flavors: Black-, Grey, and White-box fuzzer

- **Basic Fuzzer *randomly* generate Input**
  Most popular fuzzers: AFL (AFL++), libfuzzer, …

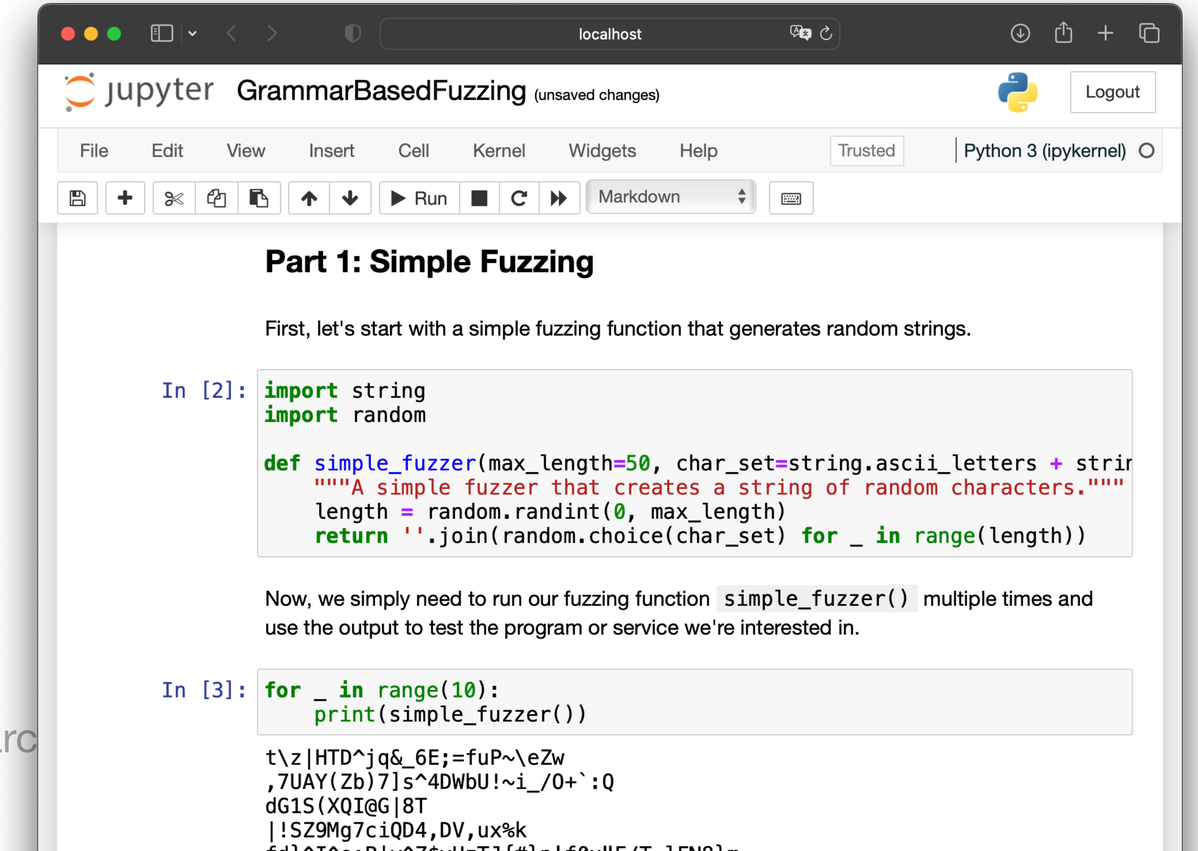
- **Fuzzing can be guided by properties**
  Most common strategies: code coverage, diverse inputs, …

- **Simple Fuzzers can easily be written by yourself!**
  Jupyter-Notebook

Basic Fuzzing

Generating Program Input 101

- **Simple Fuzzers can easily be written by yourself!**
  Jupyter-Notebook



Martin Eberlein, Doctoral Researc

Grammar-Based Fuzzing

Like *Fuzzing* with extra steps

- Most applications require structured input

Web applications, interpreters, and parser require syntactically valid input

- Guided by Grammars

Input generation can be guided by context free grammars to produce syntactically valid inputs

- More Efficient! 

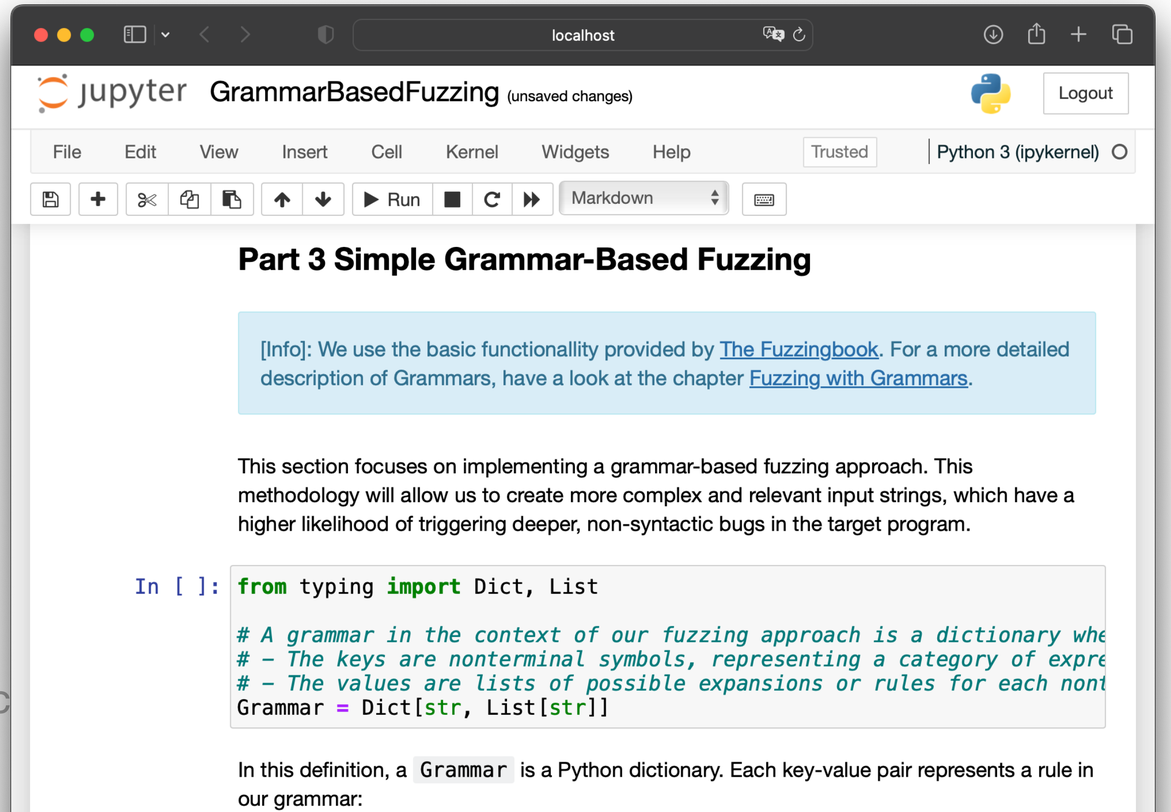
Input generation can be guided by context free grammars to produce syntactically valid inputs

Grammar-Based Fuzzing

Like *Fuzzing* with extra steps

- **More Efficient!**

Input generation can be guided by context free grammars to produce syntactically valid inputs



Martin Eberlein, Doctoral Researc

Probabilistic Grammar-Based Fuzzing

Guided by Probabilities

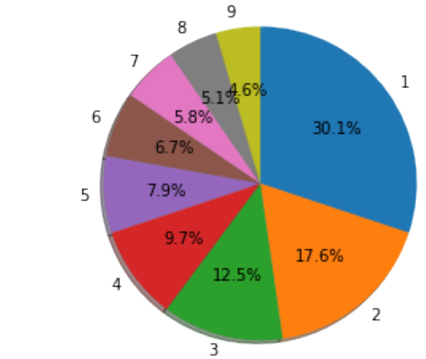
- **More Efficient!**

  Input generation can be guided by context free grammars to produce syntactically valid inputs

- **Observation:**

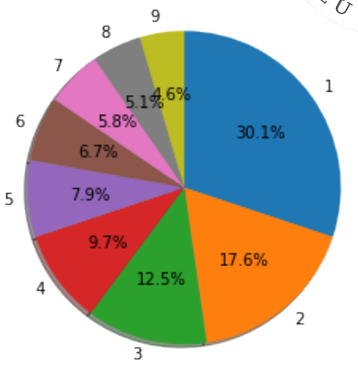
  Some Features are more important than others

- **Example: Law of the Leading Digits**
  - In real-world data sets of numerical data, the leading digit is likely to by small
  - [Newcomb 1881, Benford 1938]

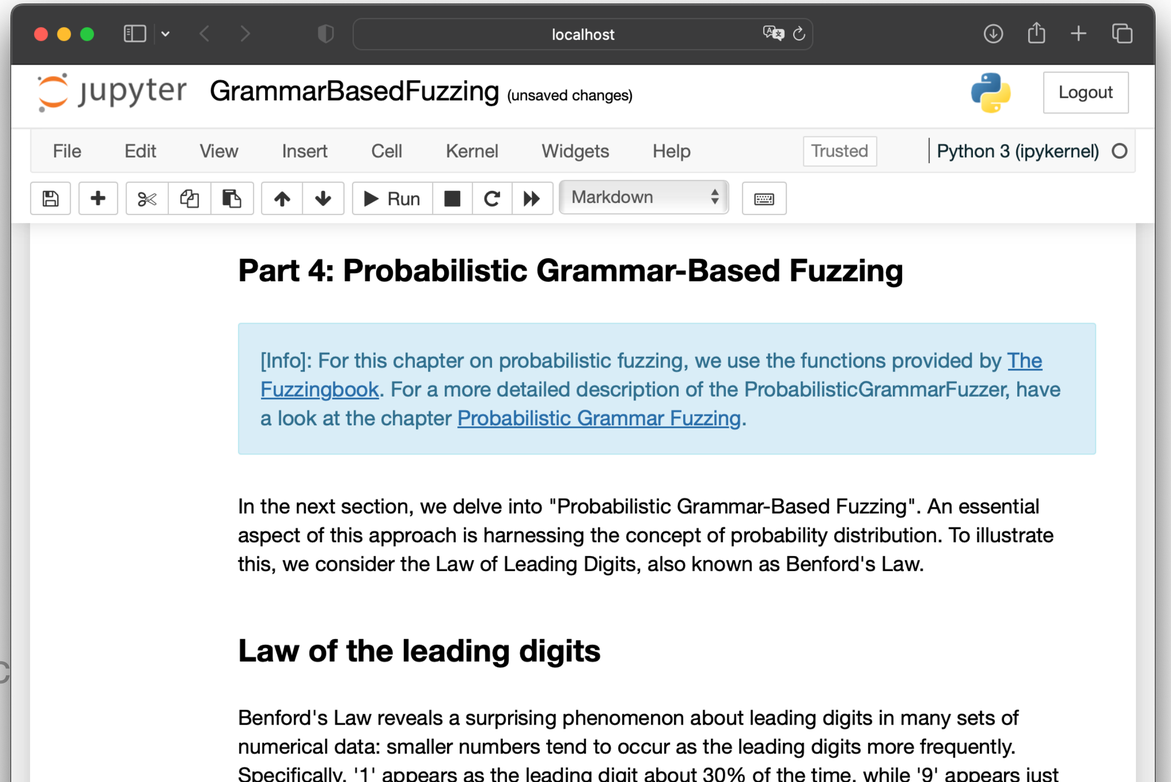


Probabilistic Grammar-Based Fuzzing

Guided by Probabilities

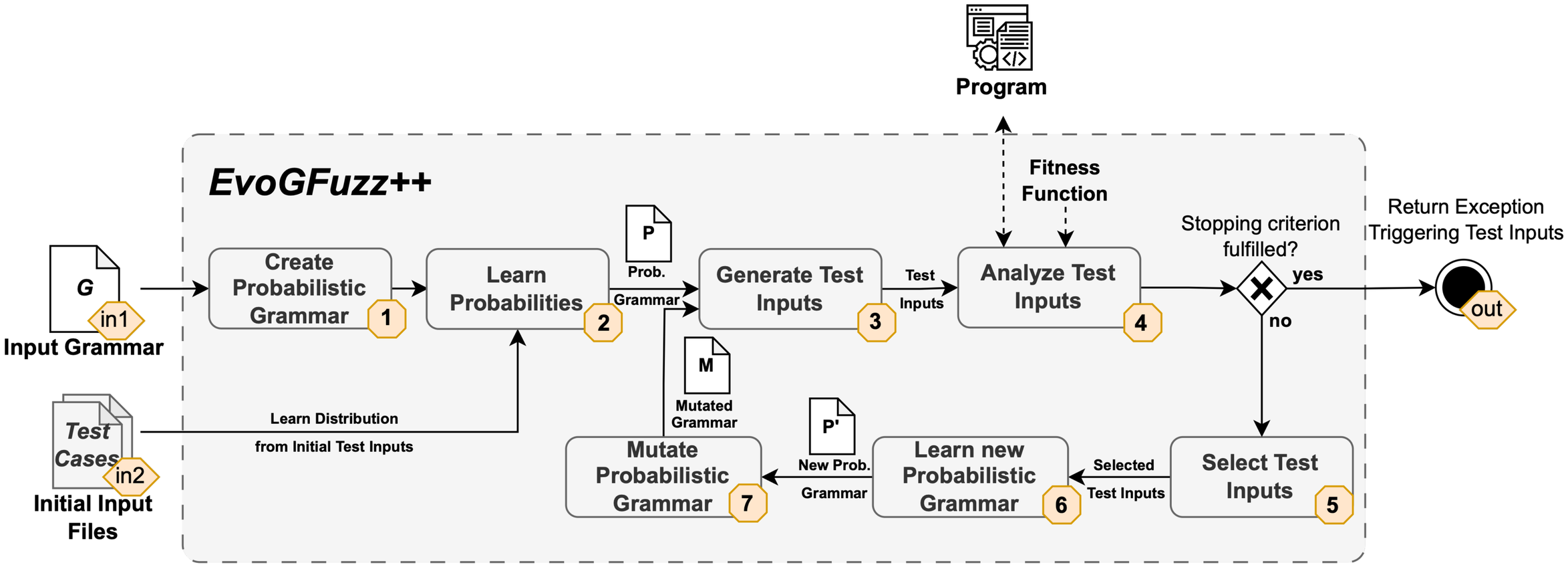


- **Example: Law of the Leading Digits**
  - In real-world data sets of numerical data, the leading digit is likely to by small
  - [Newcomb 1881, Benford 1938]

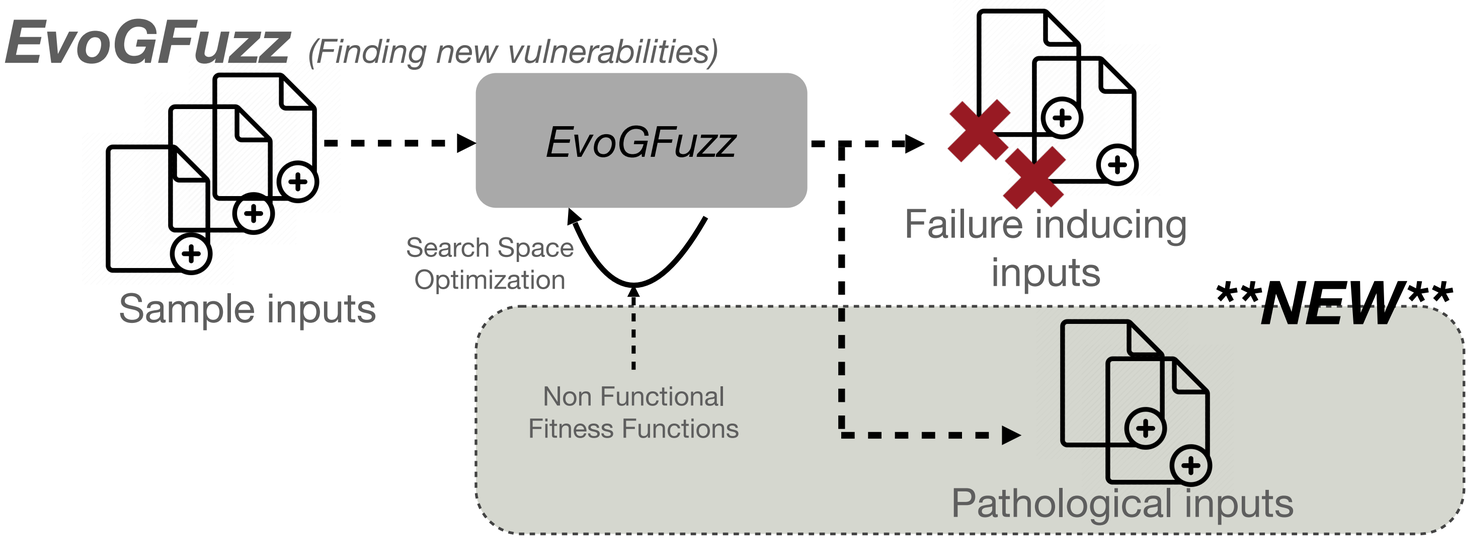


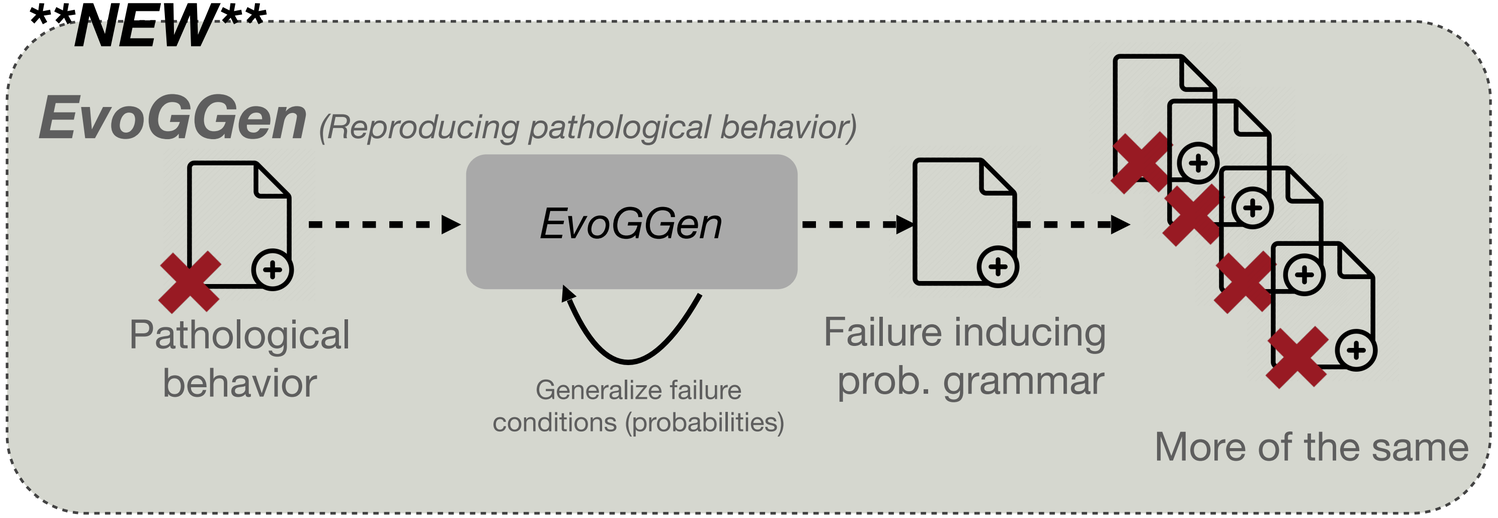
Martin Eberlein, Doctoral Researc

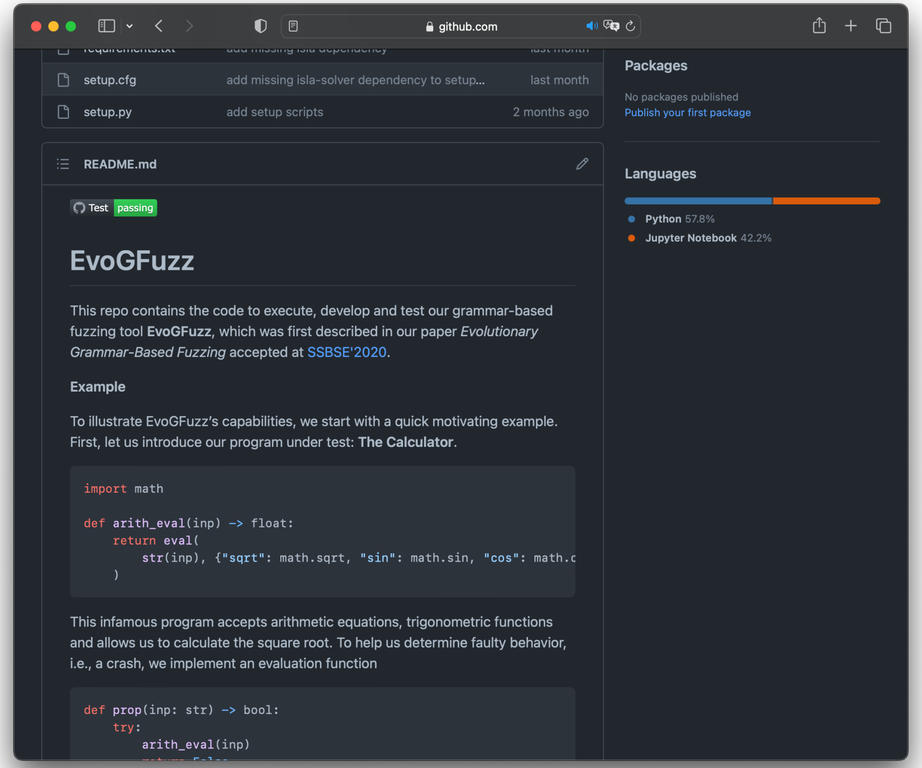
Evolutionary Grammar-Based Fuzzing



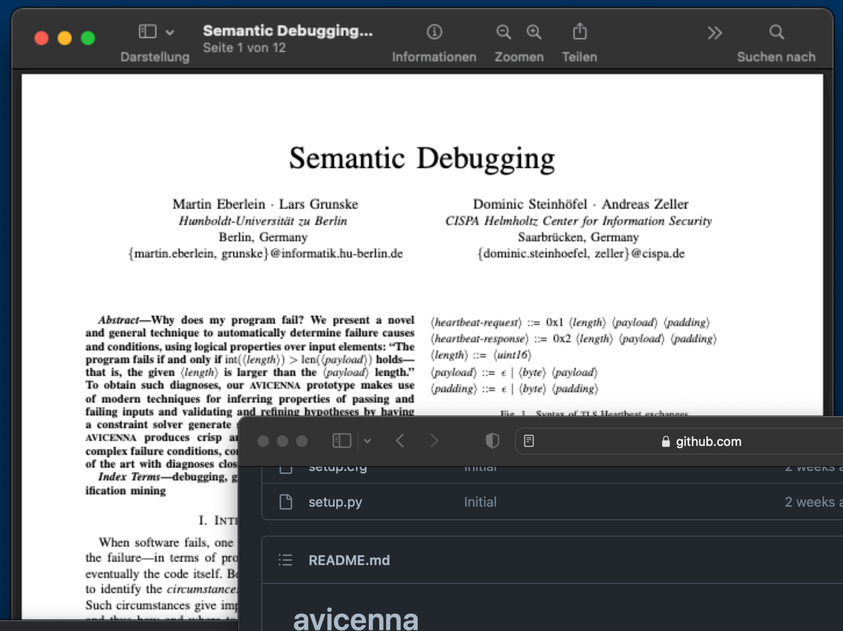
Evolutionary Grammar-Based Fuzzing

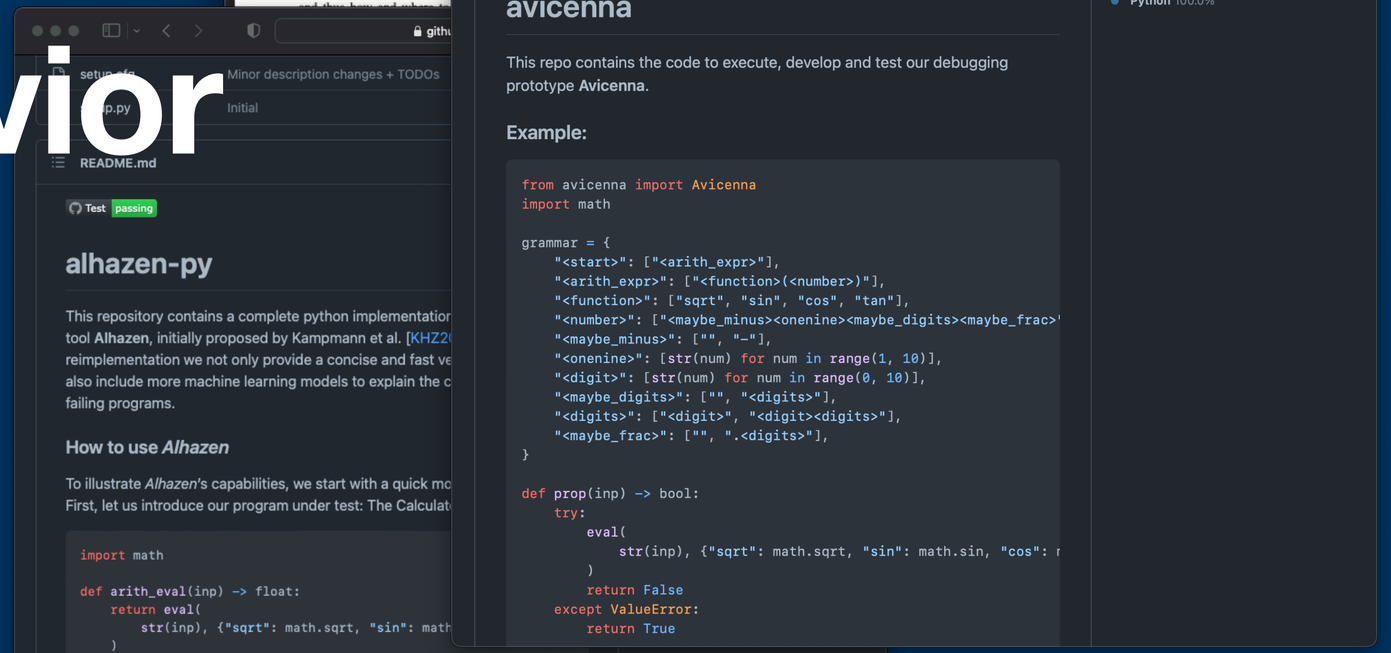




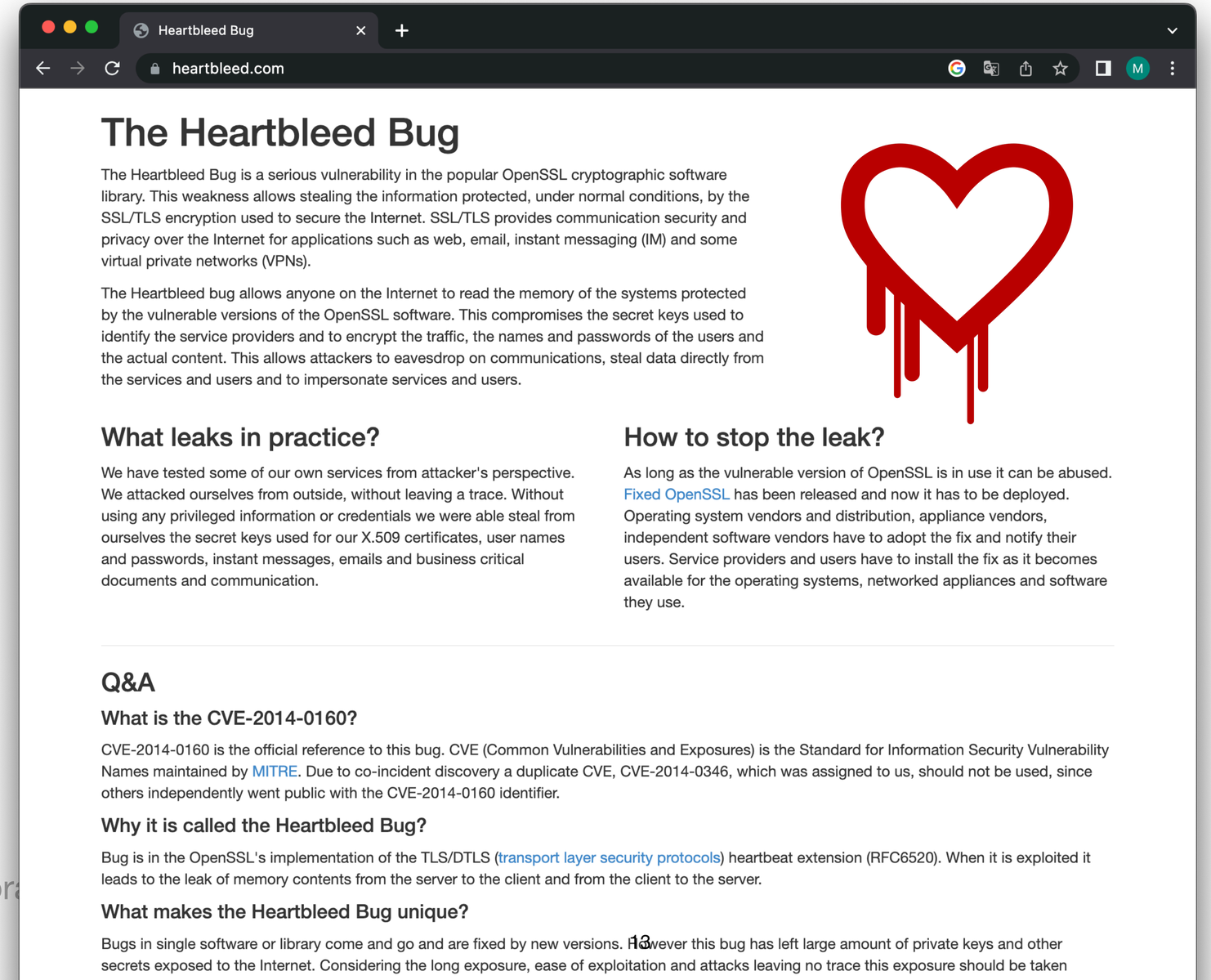


Explaining Pathological Program Behavior





Heartbeat Protocol



Heartbeat Protocol

Client

Server

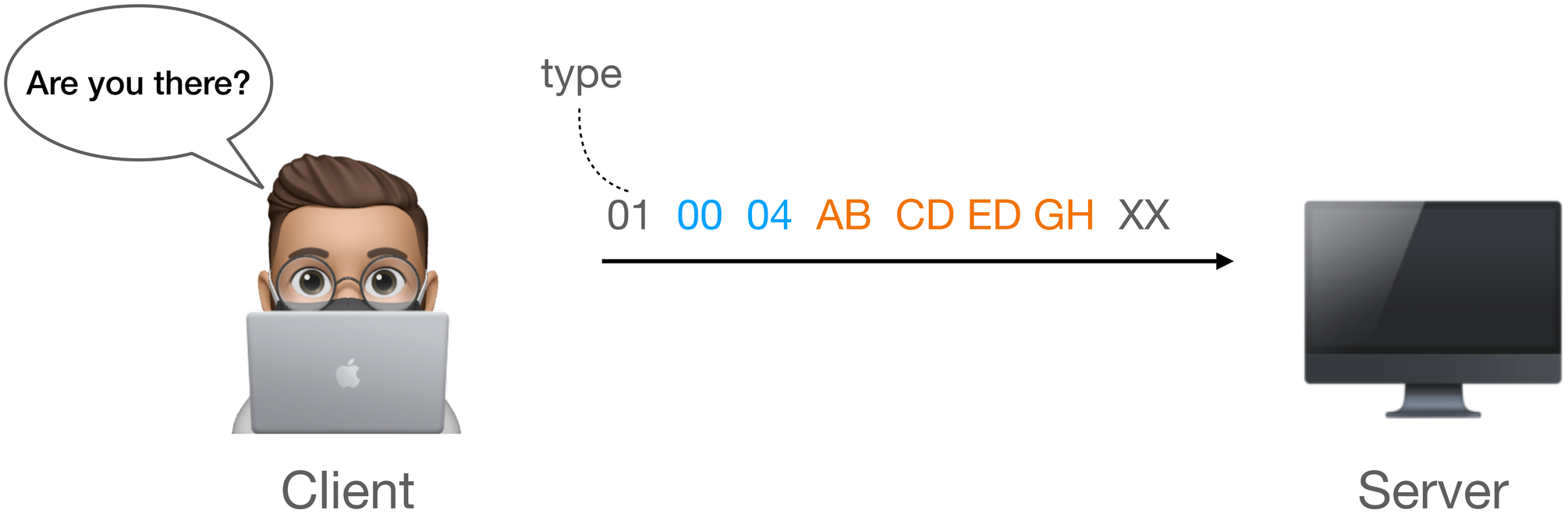
Heartbeat Protocol

Client

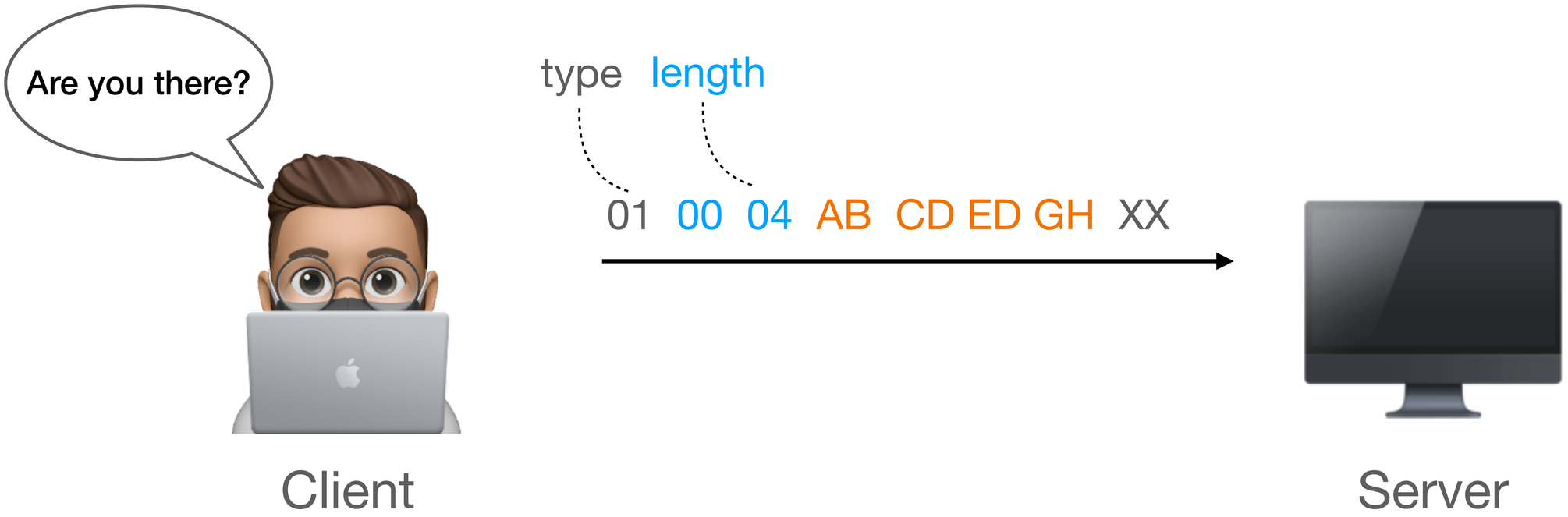
01 00 04 AB CD ED GH XX

Server

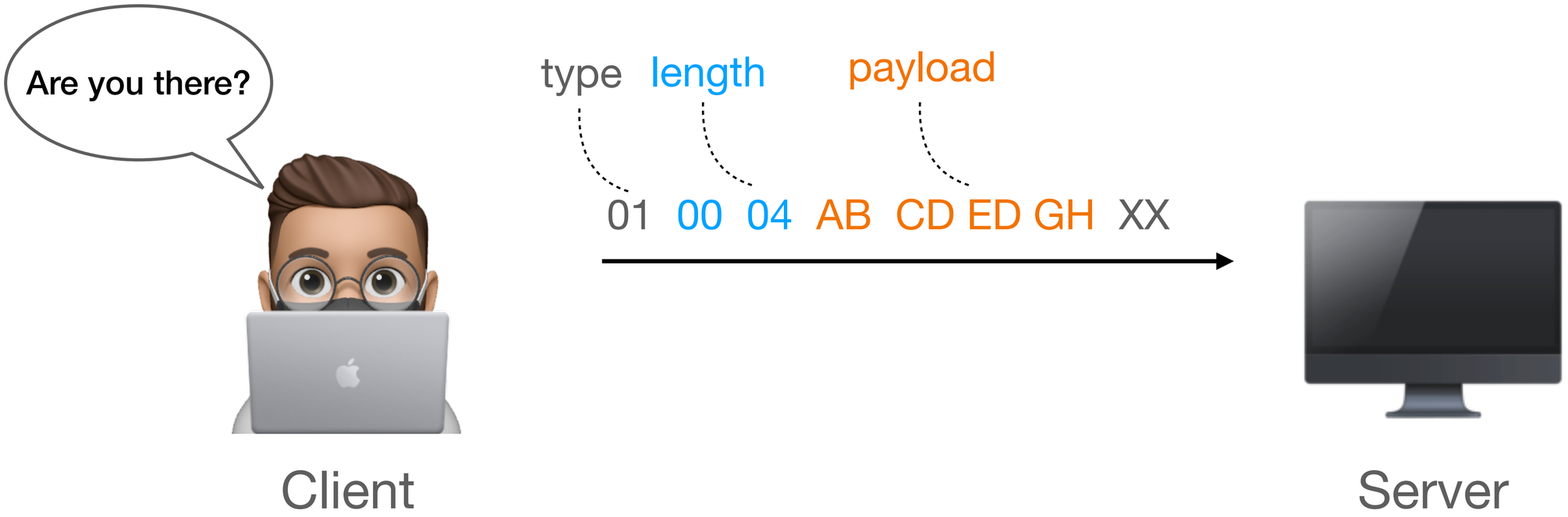
Heartbeat Protocol



Heartbeat Protocol



Heartbeat Protocol



Heartbeat Protocol

Client

type length payload padding

01 00 04 AB CD ED GH XX

Server

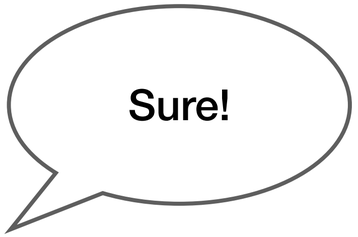
Heartbeat Protocol

Client

type length payload padding

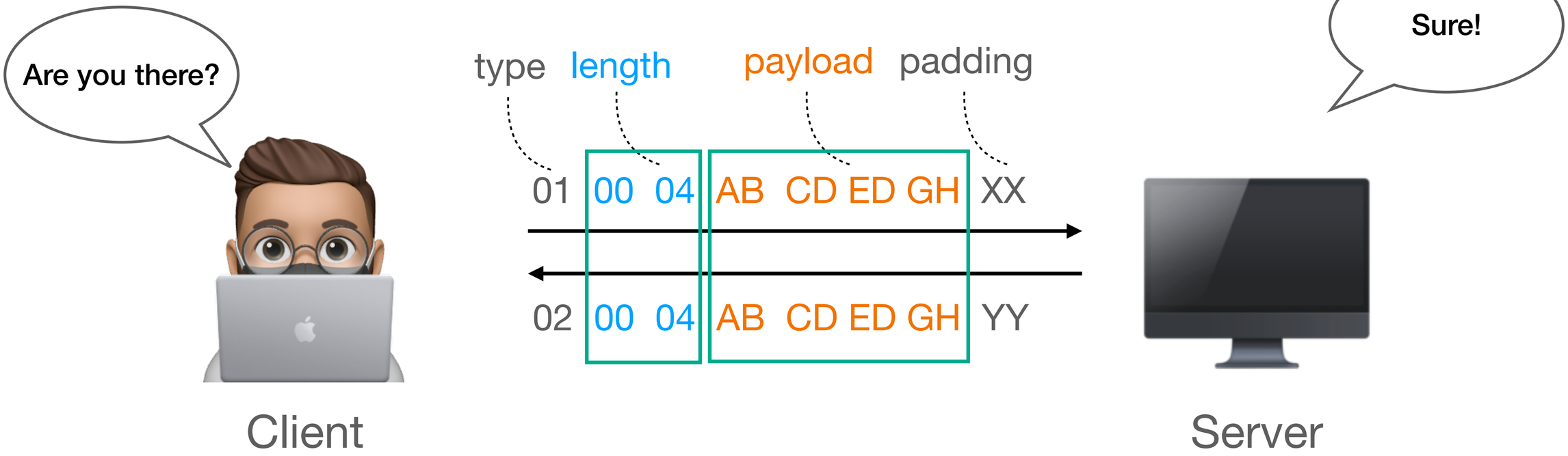
01 00 04 AB CD ED GH XX

02 00 04 AB CD ED GH YY



Server

Heartbeat Protocol



Heartbeat Protocol

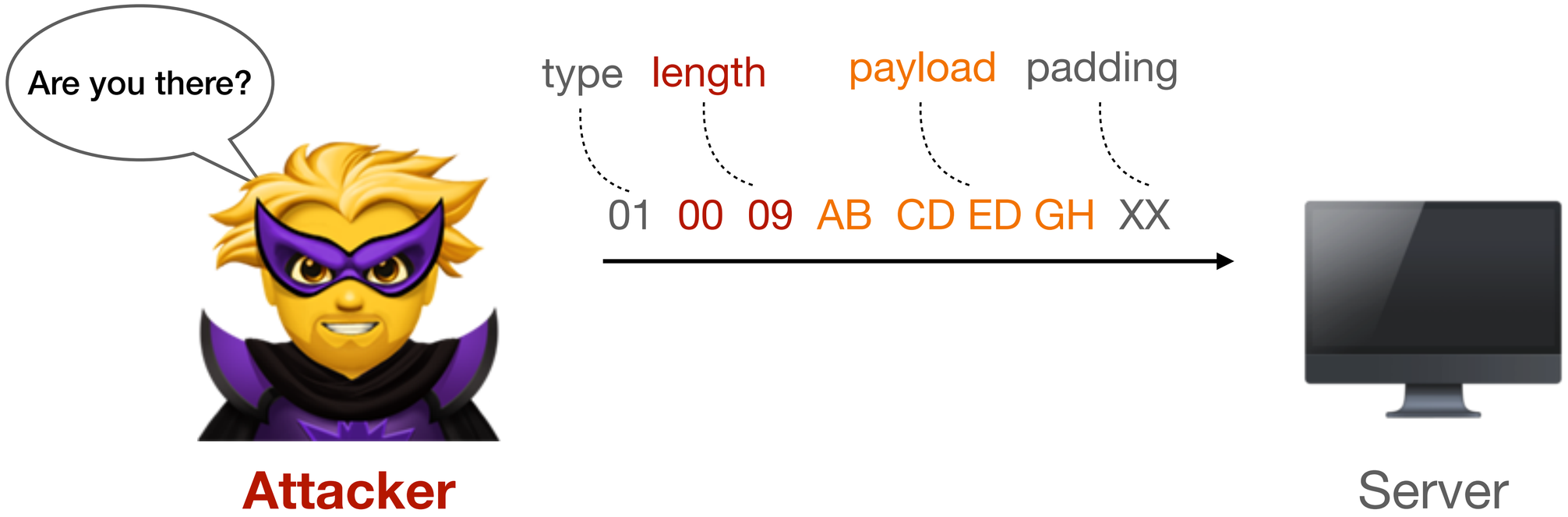
Exploit

Attacker

Server

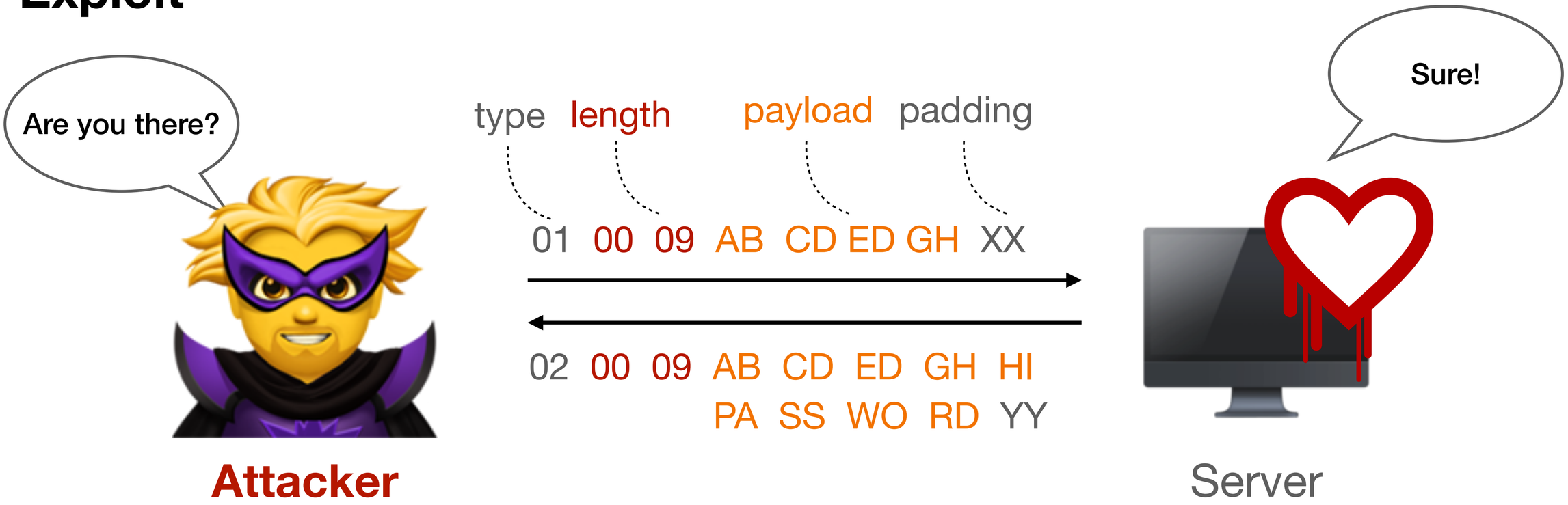
Heartbeat Protocol

Exploit



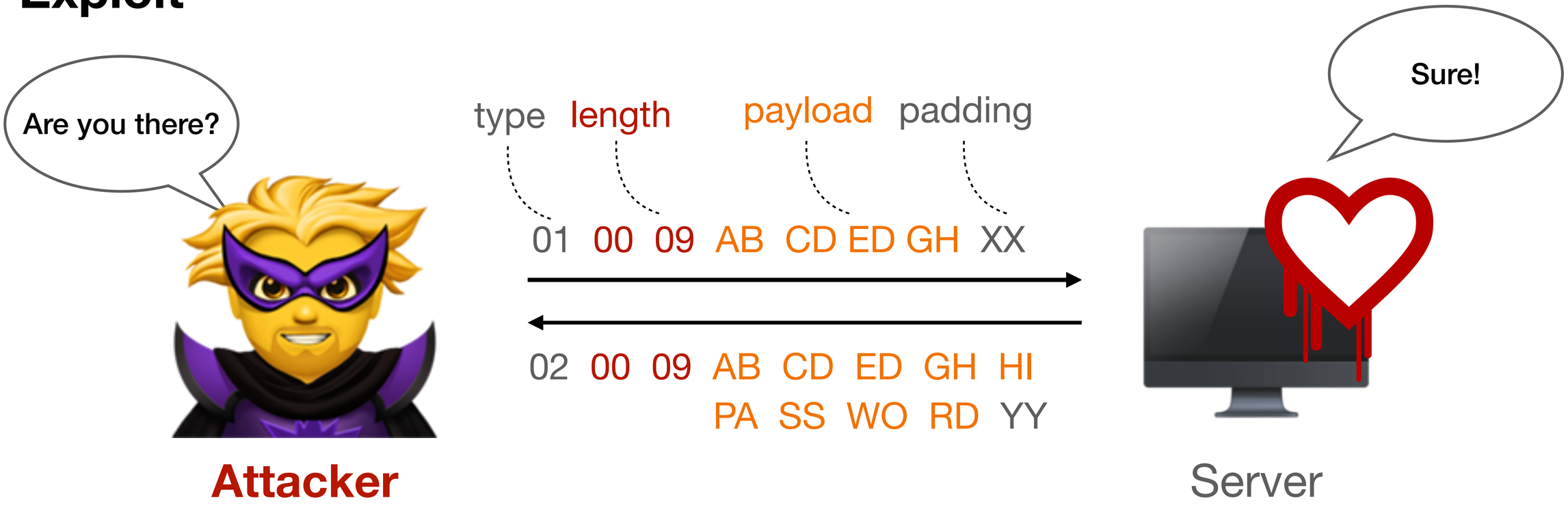
Heartbeat Protocol

Exploit



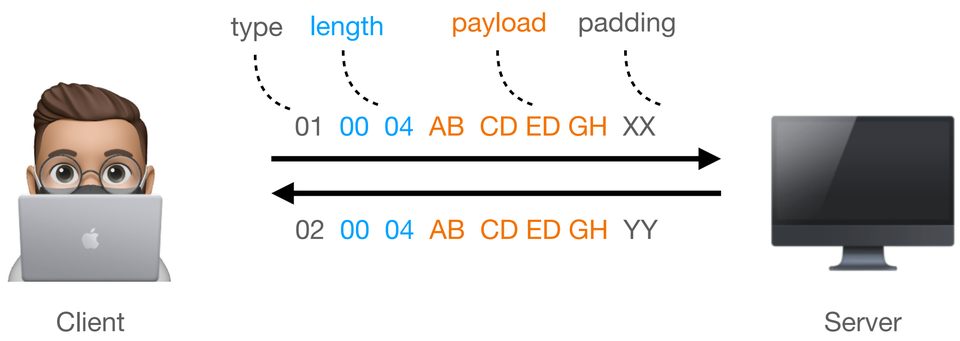
Heartbeat Protocol

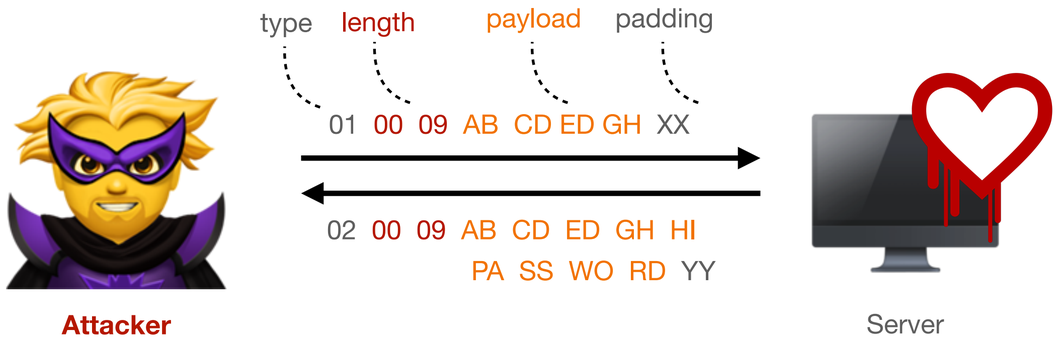
Exploit



The *heartbleed bug* occurs if
int(*length*) > ***length***(*payload*)

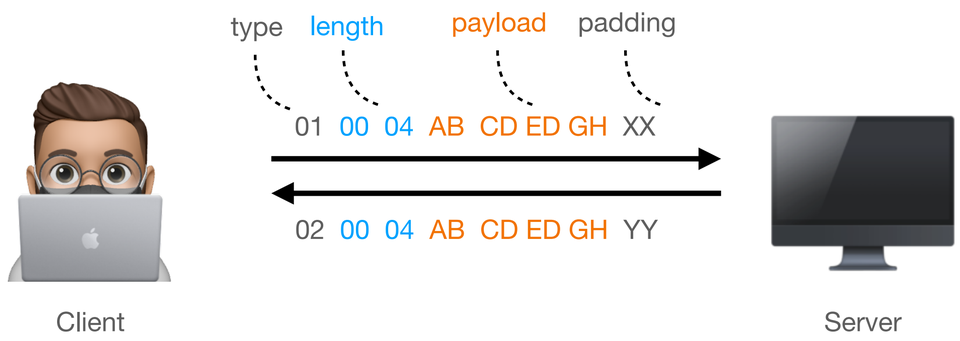
Explaining Program Behavior

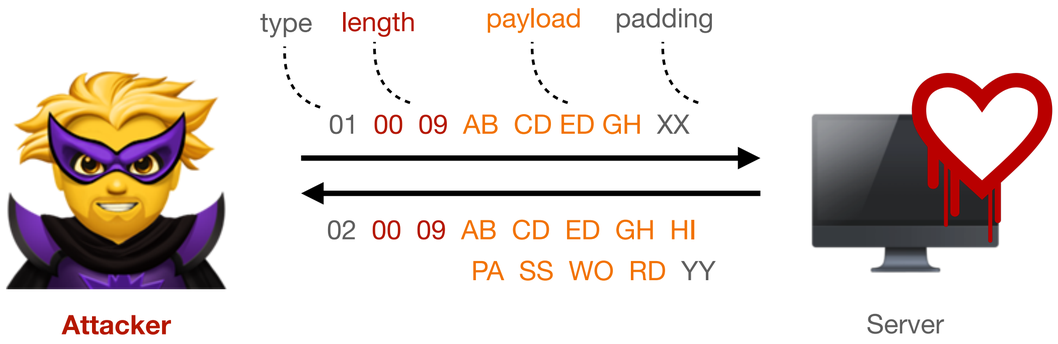




Can we **automatically** generate a **theory** explaining why a given input triggers the bug?

Explaining Program Behavior



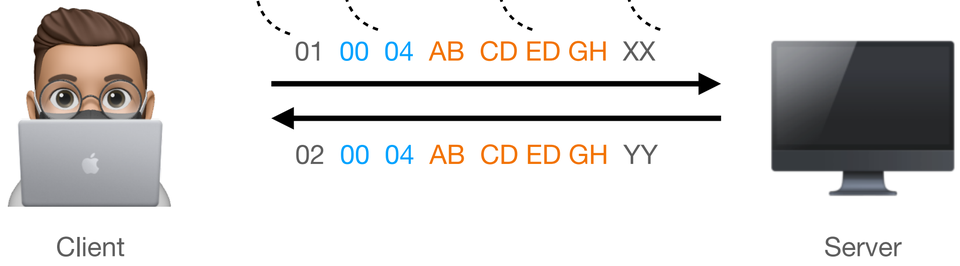


Can we **automatically** generate a **theory** explaining why a given input triggers the bug?

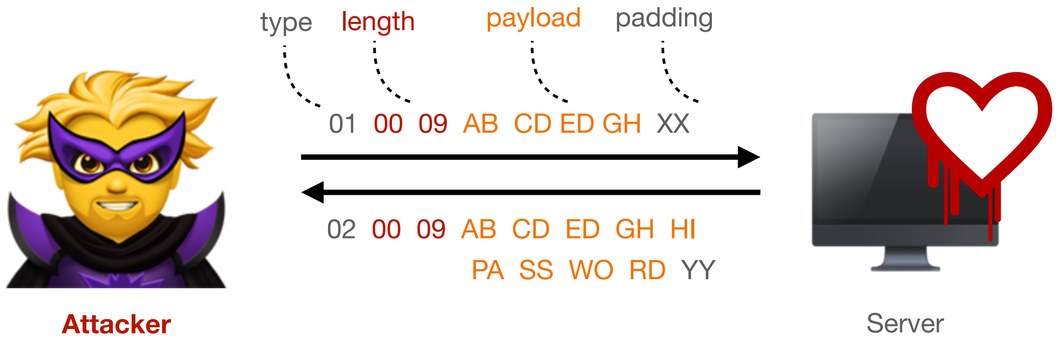
Possible Theories:

```
payload >= 9

length > 7

length > payload
```

Explaining Program Behavior



Can we **automatically** generate a **theory** explaining why a given input triggers the bug?

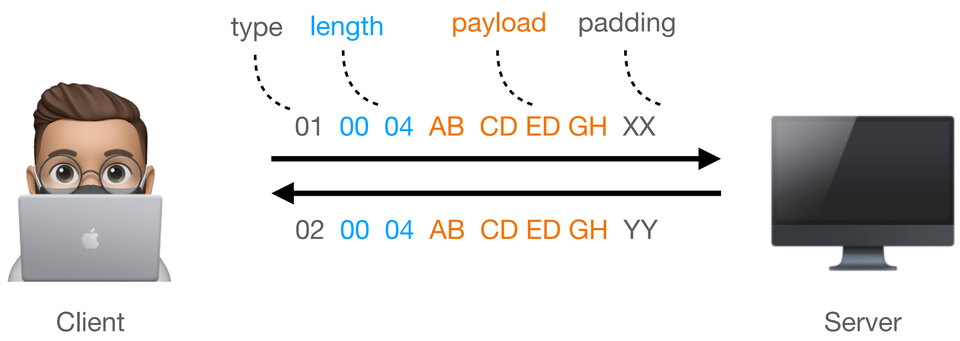


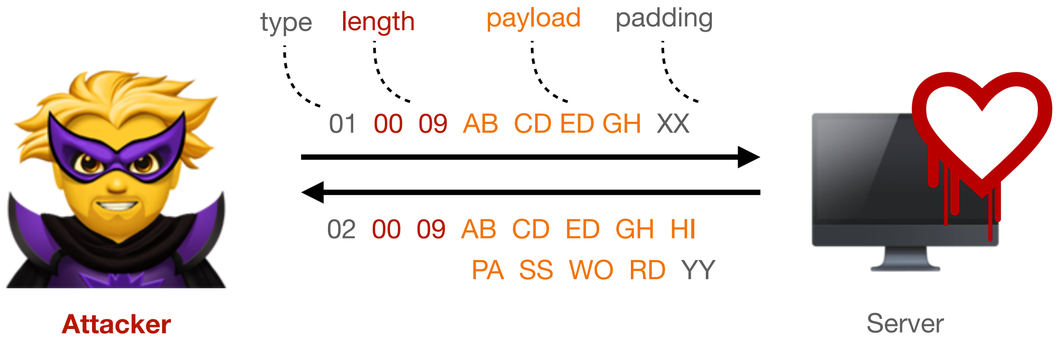
Observe Interactions

Input Generator

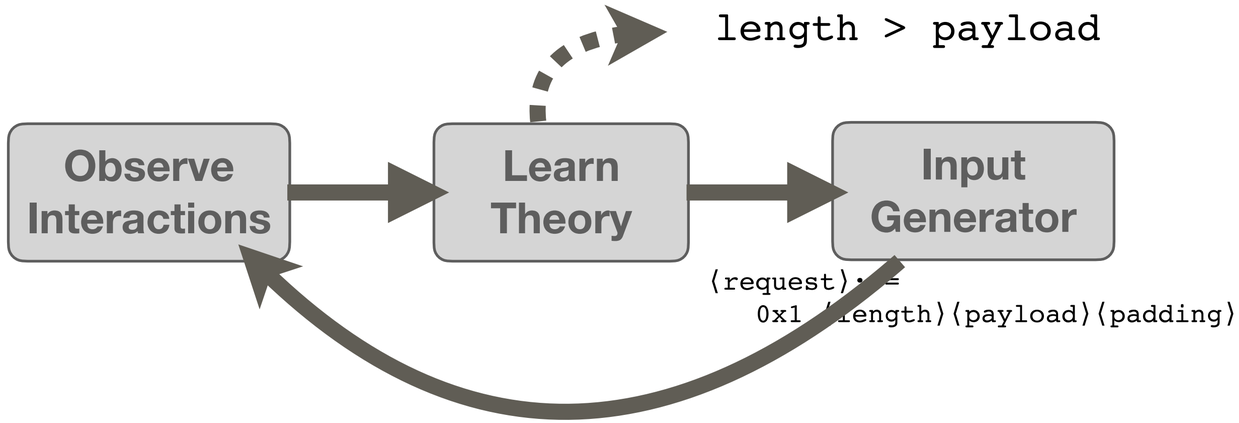
```
⟨request⟩::=
   0x1 ⟨length⟩⟨payload⟩⟨padding⟩
```

Explaining Program Behavior



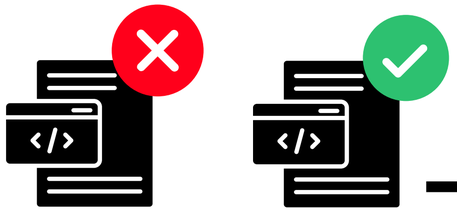


Can we **automatically** generate a **theory** explaining why a given input triggers the bug?



Semantic Debugging

Learning Failure Circumstances



test inputs

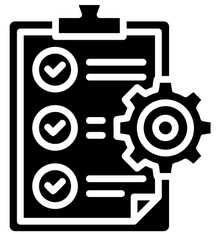
passing + failing

```
⟨request⟩::= 0x1 0x2000 „hello“
             0x0 …
```

Input specification

```
⟨request⟩::=
   0x1 ⟨length⟩⟨payload⟩⟨padding⟩
```

AVICENNA

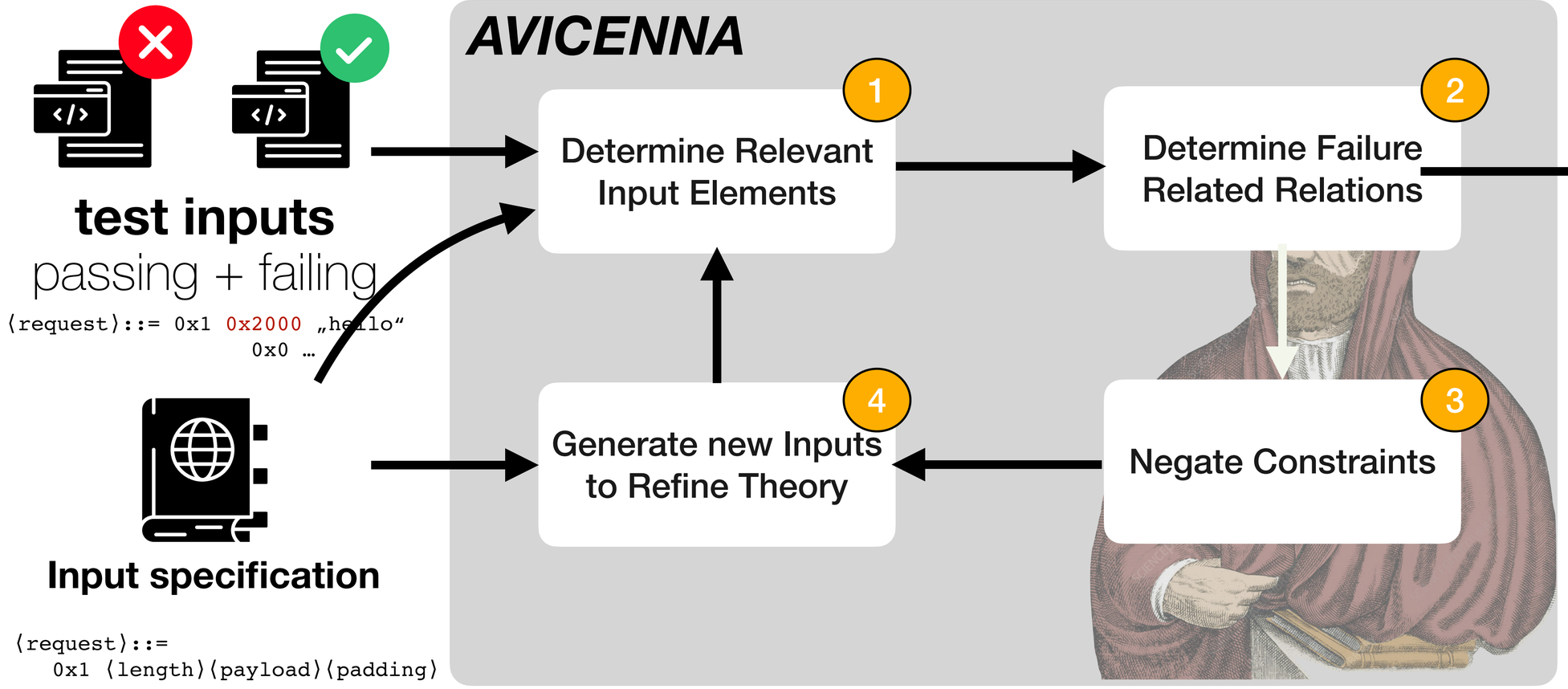


Failure Circumstances

```
int(⟨length⟩)
   > len(⟨payload⟩)
```

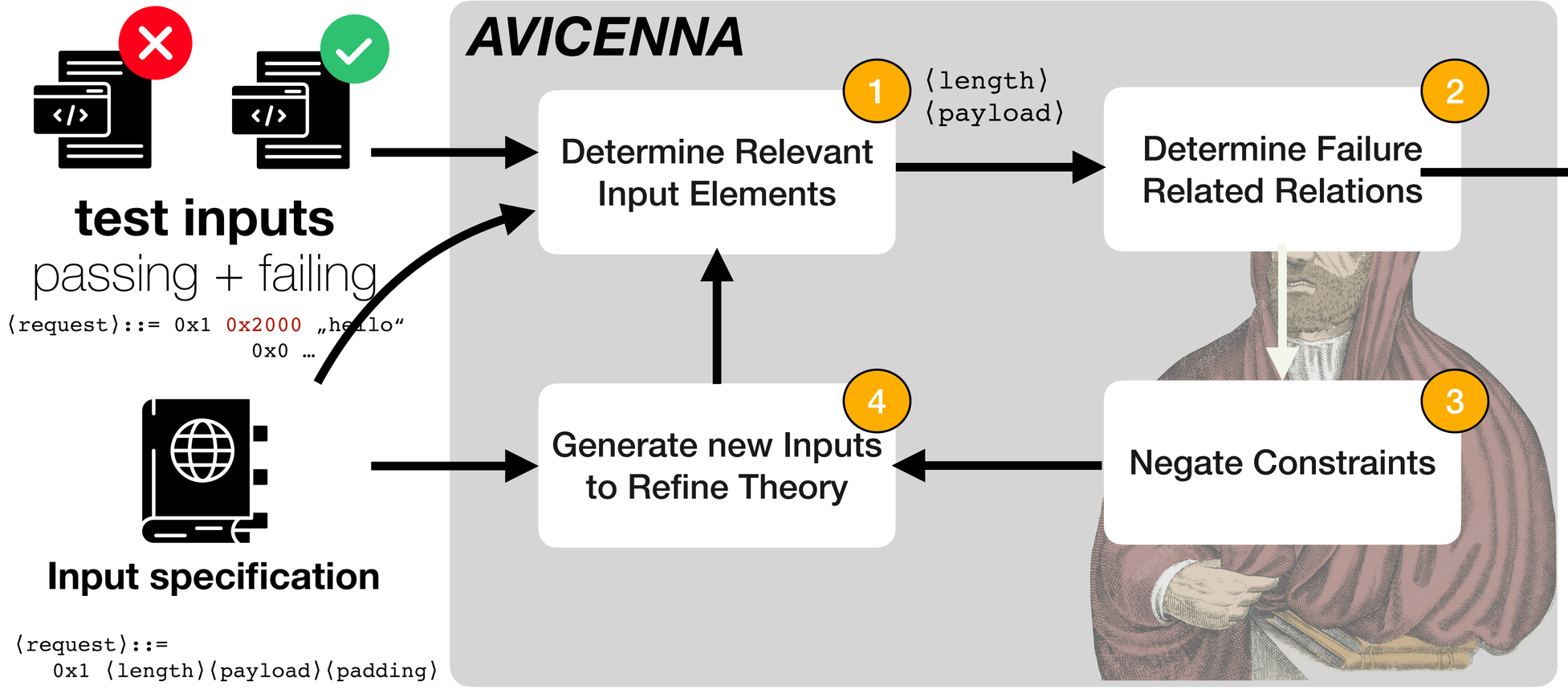
Semantic Debugging

Learning Failure Circumstances



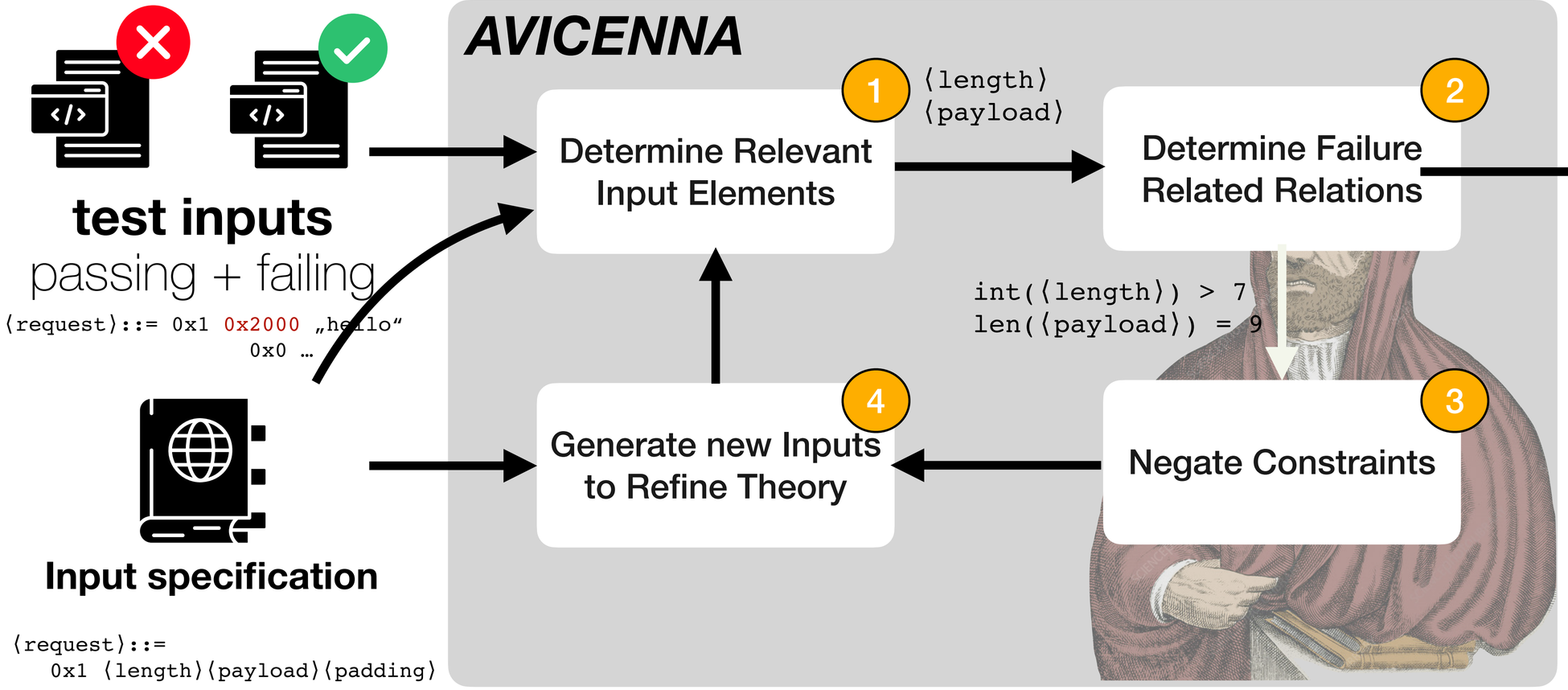
Semantic Debugging

Learning Failure Circumstances



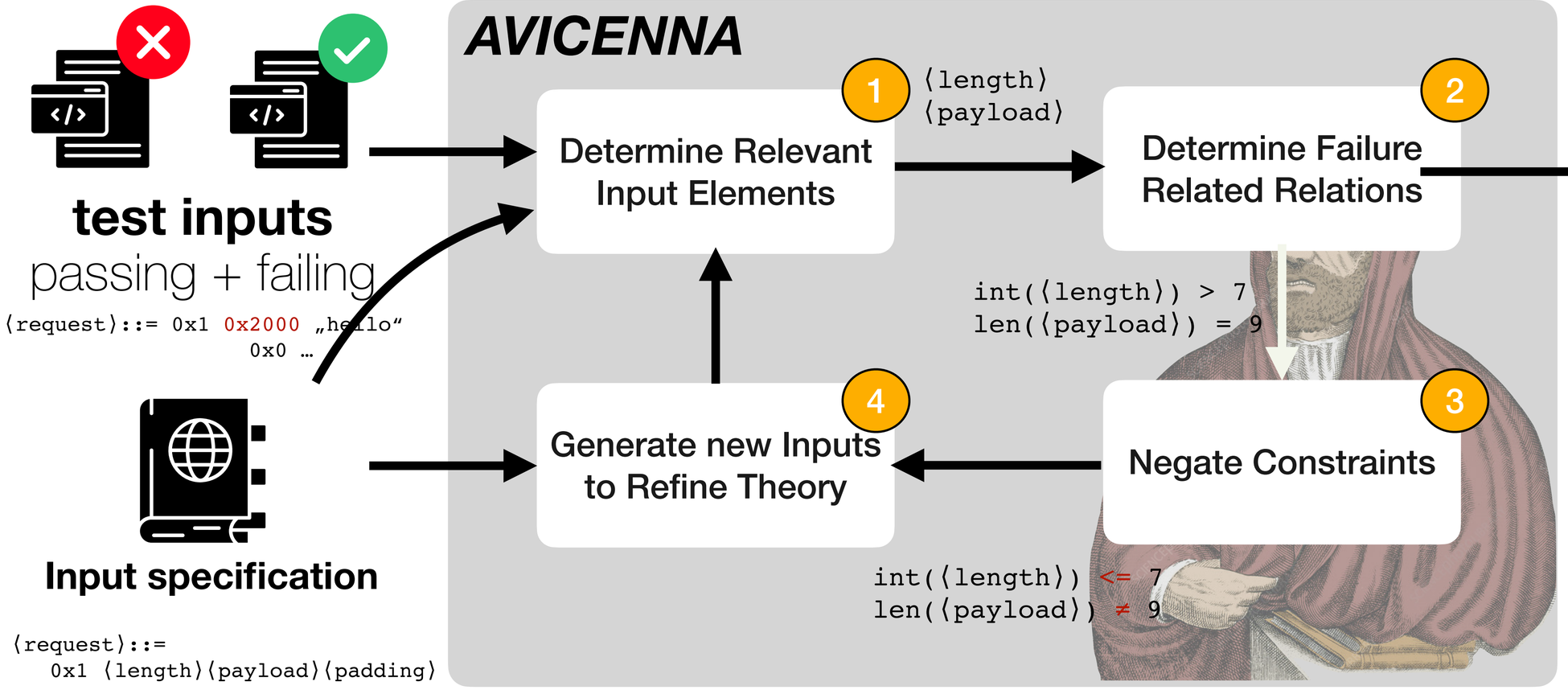
Semantic Debugging

Learning Failure Circumstances



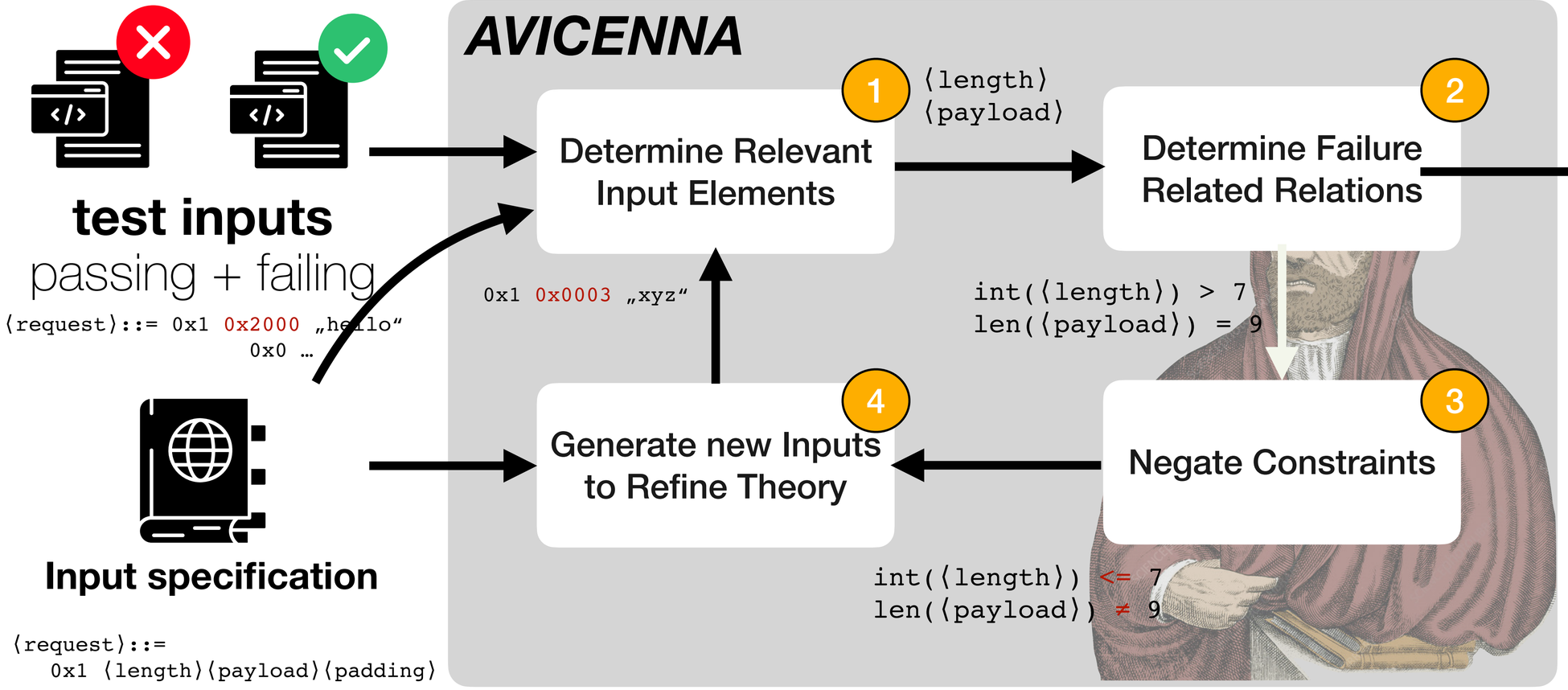
Semantic Debugging

Learning Failure Circumstances



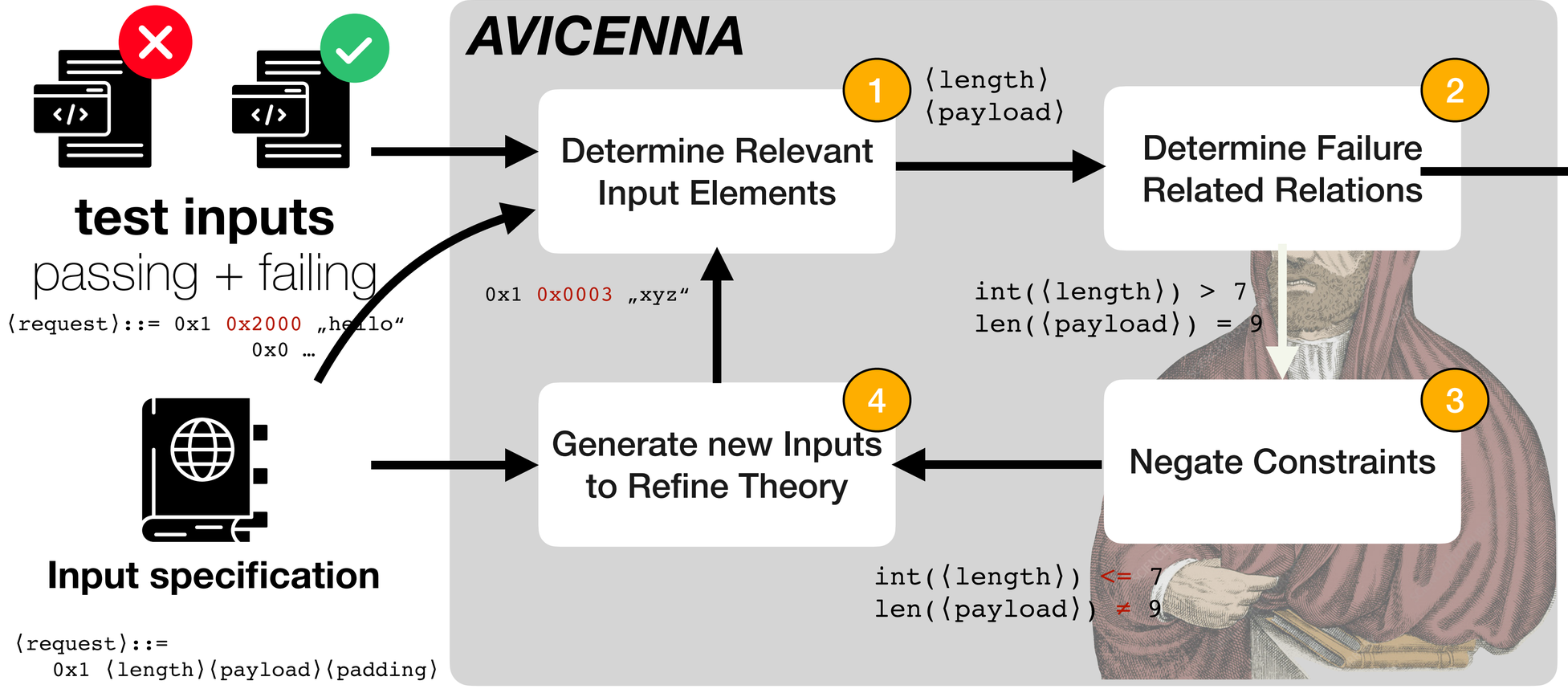
Semantic Debugging

Learning Failure Circumstances



Semantic Debugging

Learning Failure Circumstances

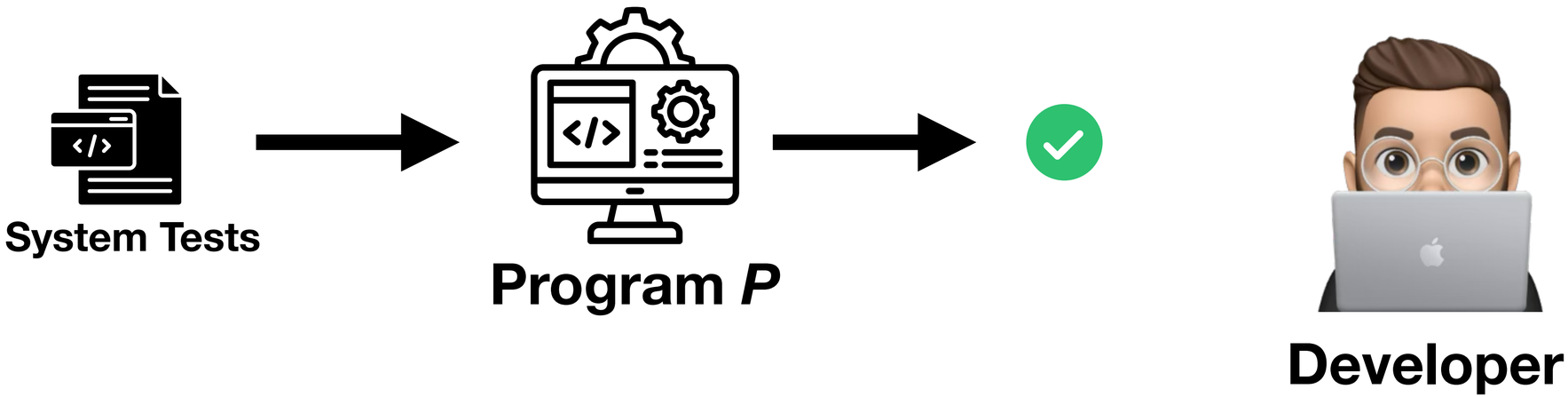


Which Inputs Trigger my Patch?

Generating explanations to describe the input properties affected by a patch

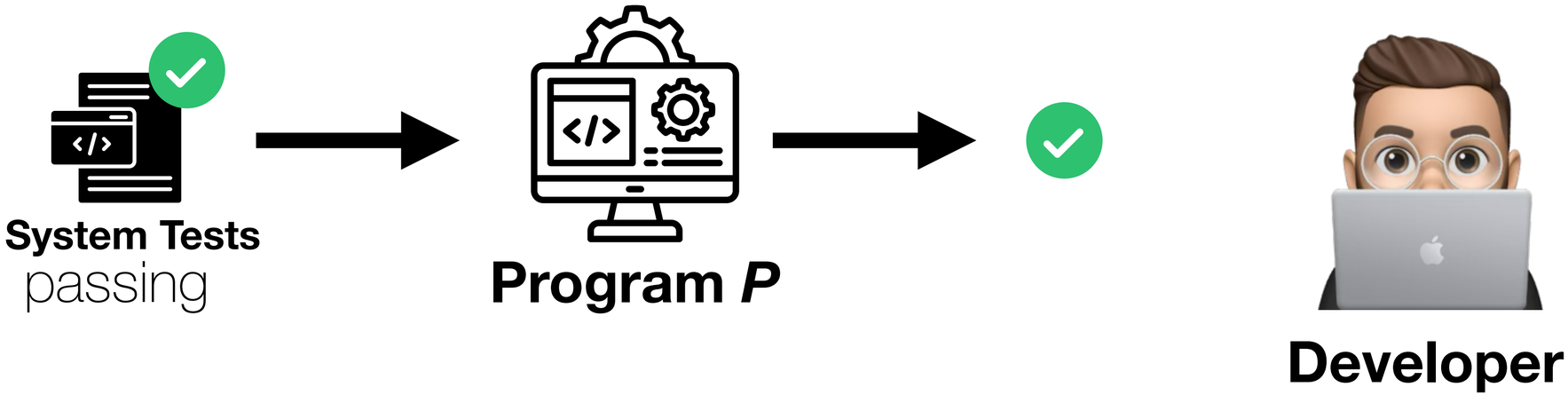
Trust Issues?

Would you trust a patch generated by ChatGPT to fix your critical code?



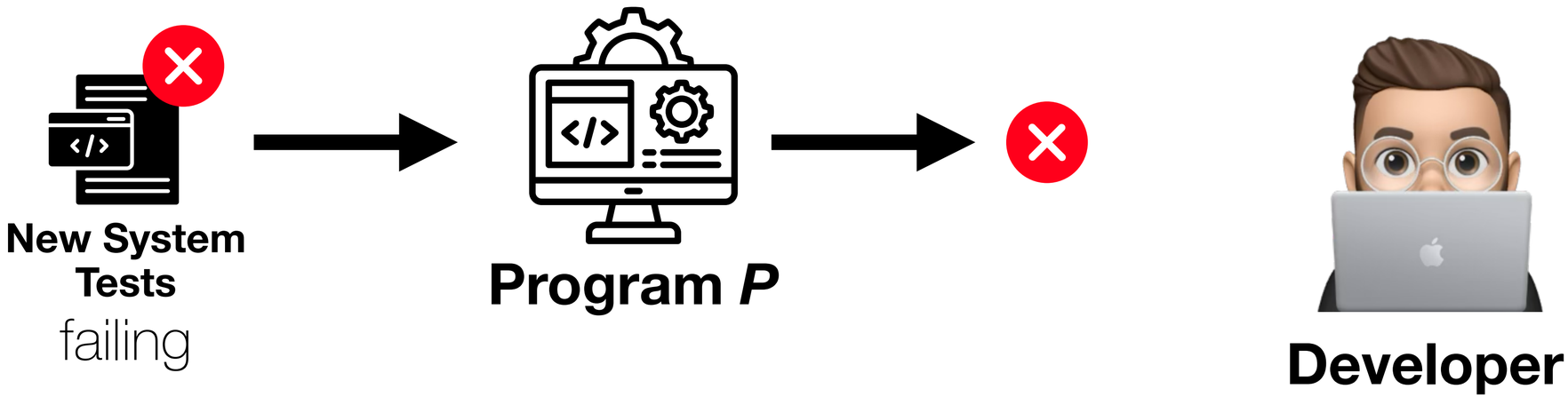
Trust Issues?

Would you trust a patch generated by ChatGPT to fix your critical code?



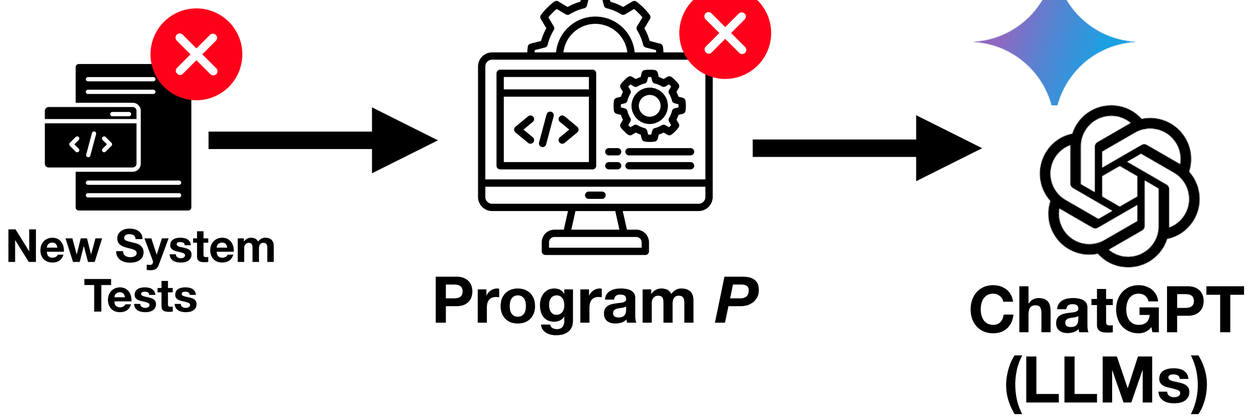
Trust Issues?

Would you trust a patch generated by ChatGPT to fix your critical code?



Trust Issues?

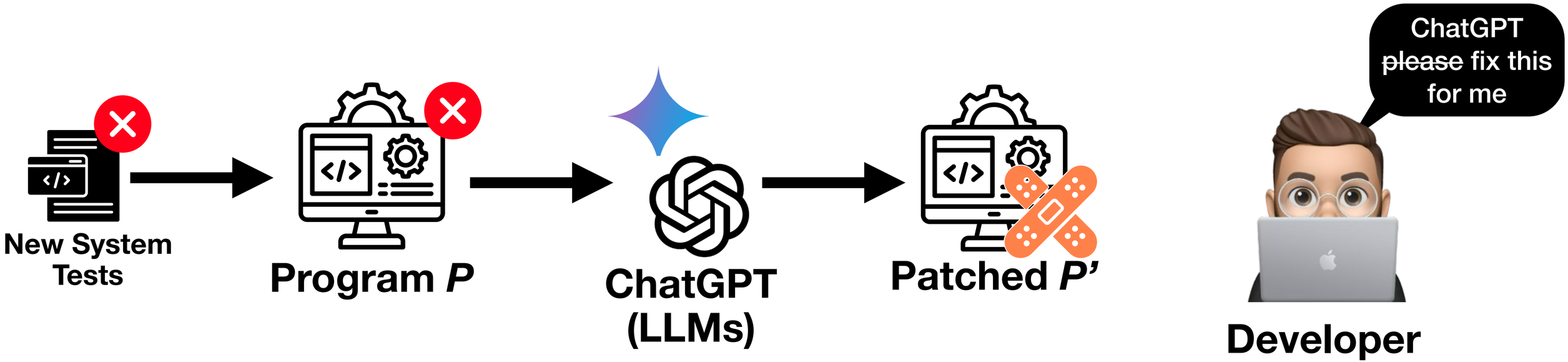
Would you trust a patch generated by ChatGPT to fix your critical code?



Developer

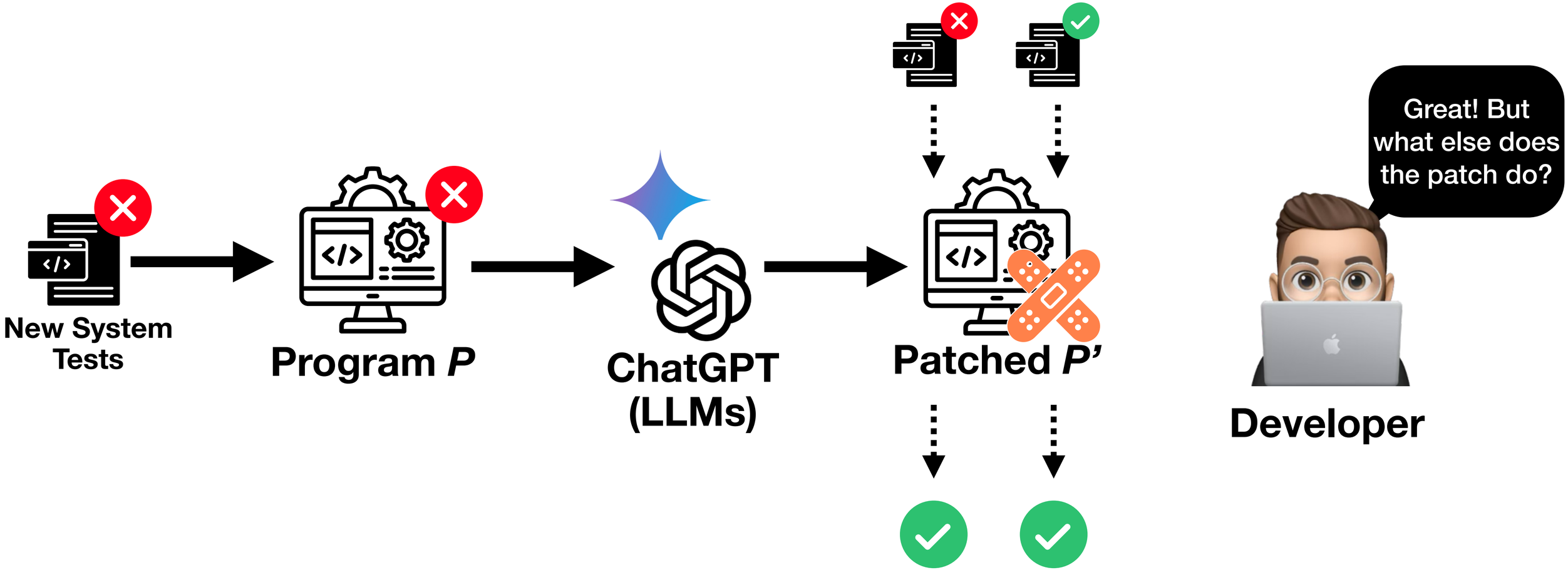
Trust Issues?

Would you trust a patch generated by ChatGPT to fix your critical code?



Trust Issues?

Would you trust a patch generated by ChatGPT to fix your critical code?



Trust Issues?

The Core Issue

- APR tools—including those powered by LLMs—are powerful but often opaque.
- It's unclear exactly which inputs are affected by these automatically generated patches.
- This lack of clarity creates significant trust issues for developers.

Developer

Trust Issues?

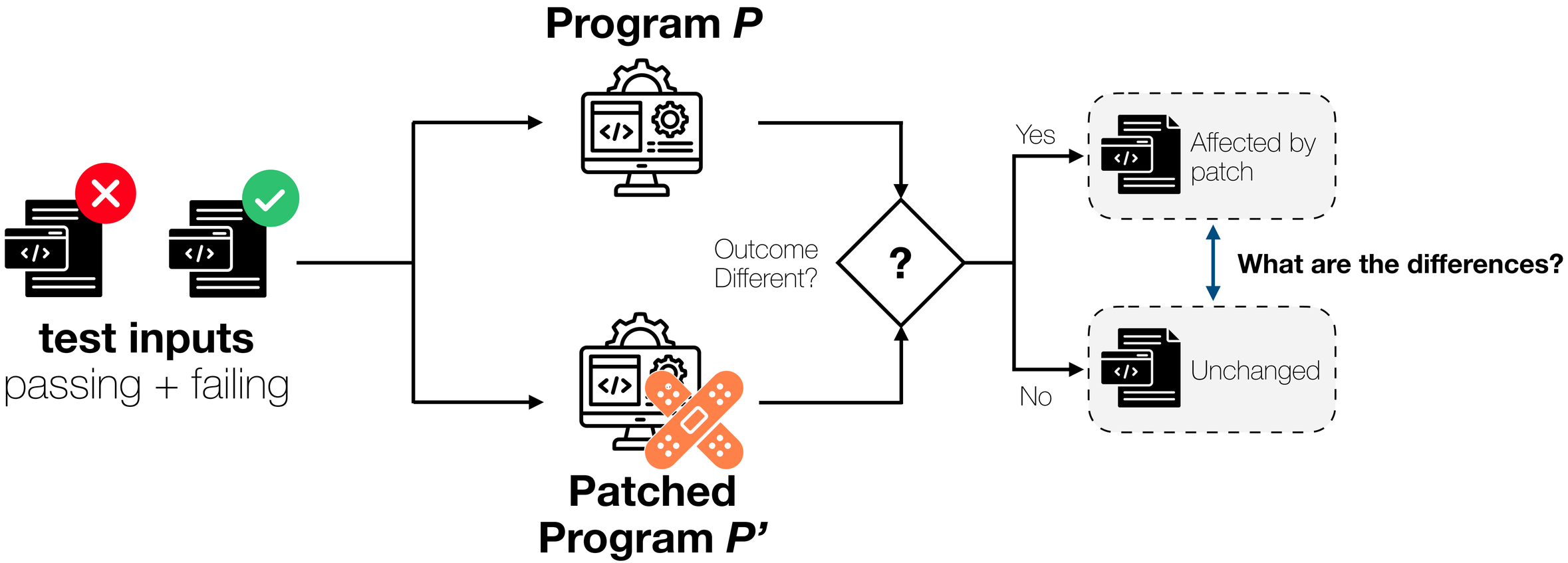
Why is this crucial?

- Developers must be certain about patch boundaries to ensure reliability.
- Blind trust in APR tools or LLM-generated fixes could lead to unforeseen consequences and software failures.

Developer

Automatic Explanations

Differential Behavior



Example

```
1  import math
2
3  def logarithm(x: float):
4      return math.log(x)
```

Fig. 2. A simple function computing the natural logarithm of a number x. While straightforward, it does not handle edge cases such as negative values, leading to errors.

output

P


logarithm(10)


logarithm(-34)


logarithm(-1521)

Example

```python
import math

def logarithm(x: float):
    return math.log(x)
```

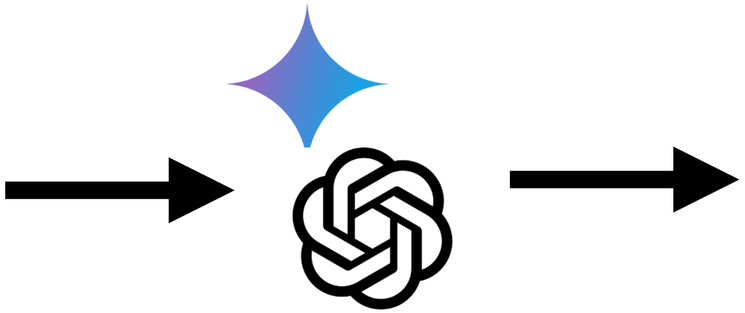
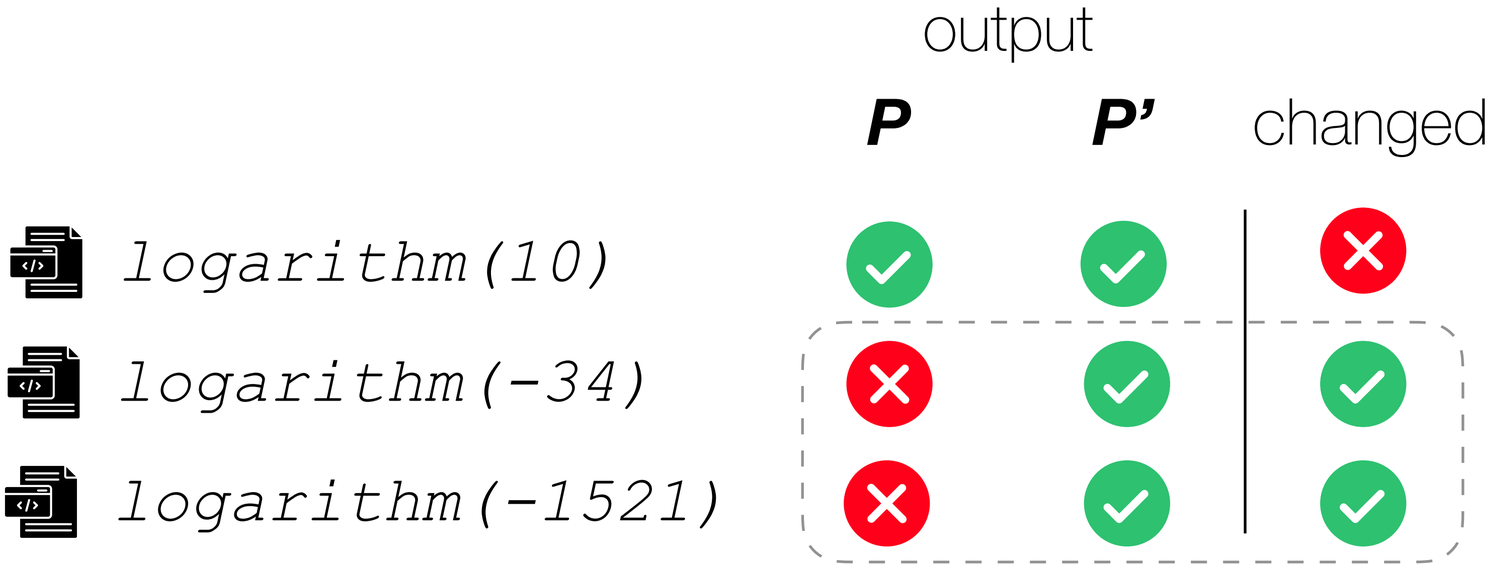
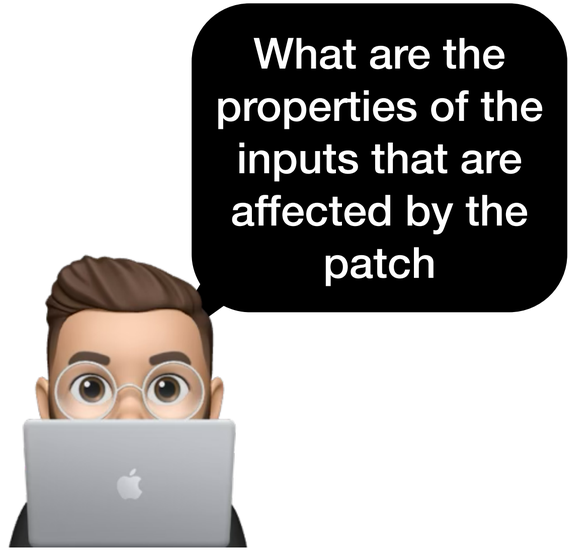


Fig. 2. A simple function computing the natural logarithm of a number x. While straightforward, it does not handle edge cases such as negative values, leading to errors.

```python
import math

def logarithm(x: float):
    # Patch applied to cover edge cases
    if x == 0:
        return float("-inf")
    elif x > 0:
        return math.log(x)
    else:
        raise TypeError("Input must be a" \
                        "positive value")
```

Fig. 3. Patched logarithmic function. It explicitly checks for zero and negative numbers. These changes ensure robust handling of edge cases.



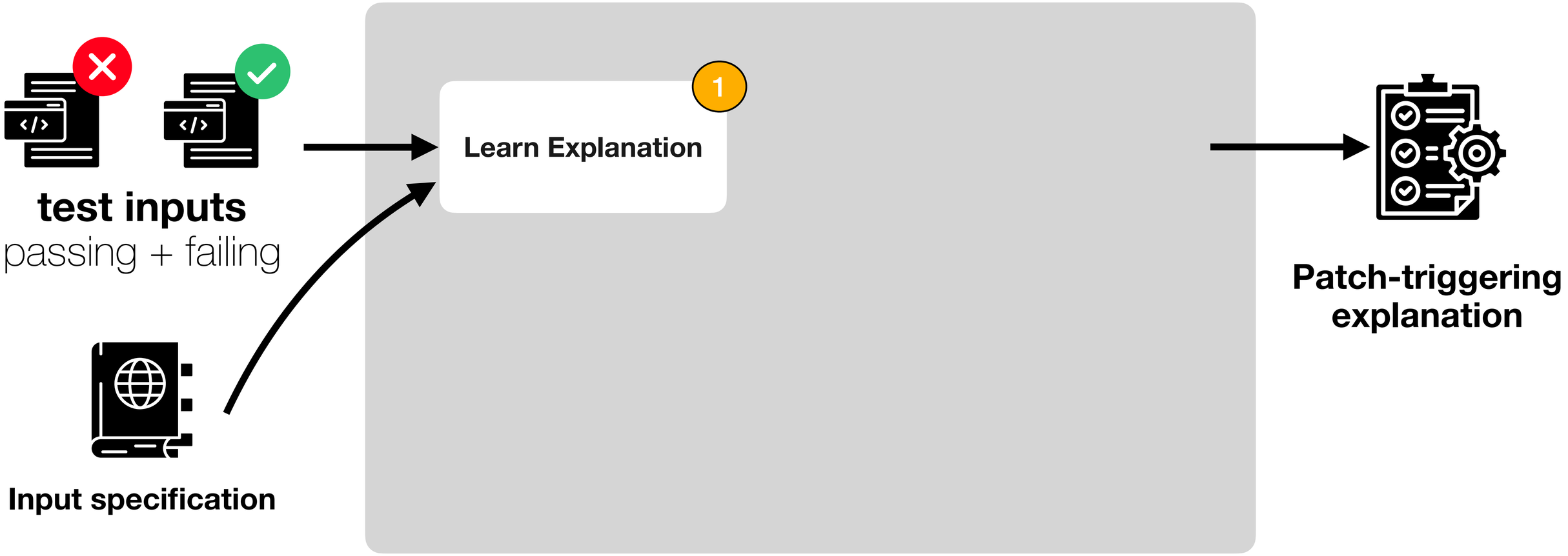


Automatic Explanations

Generating explanations to describe the input properties affected by a patch

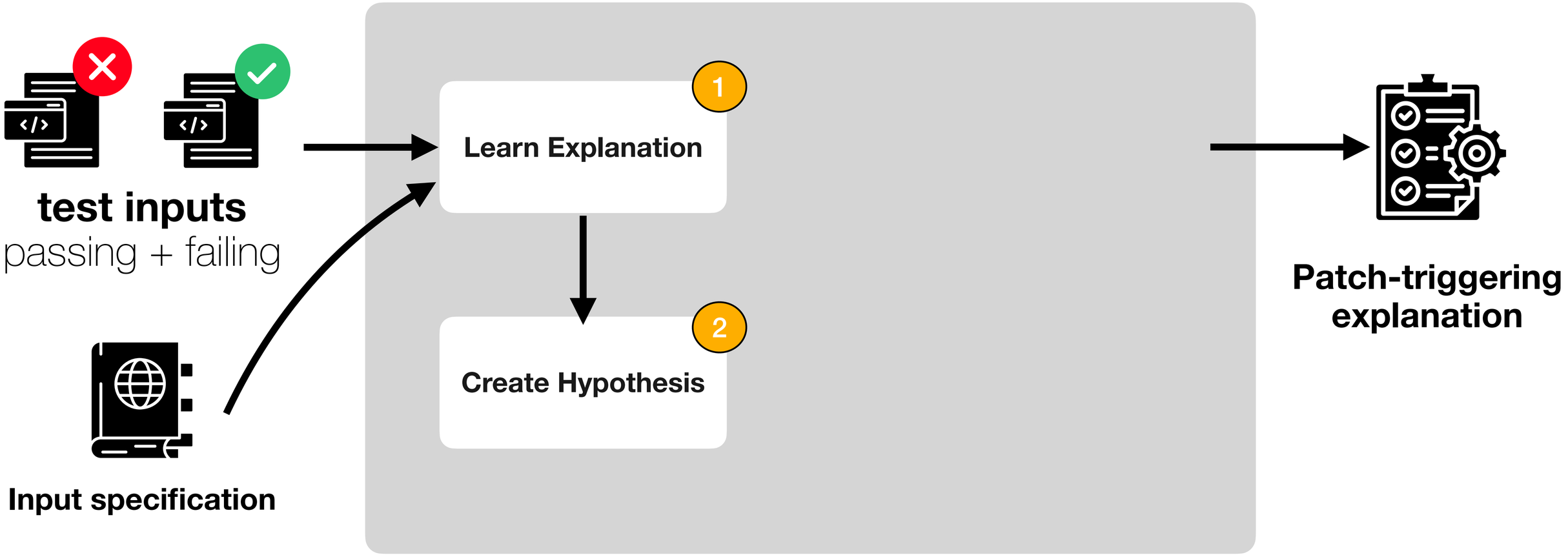
Automatic Explanations

Generating explanations to describe the input properties affected by a patch



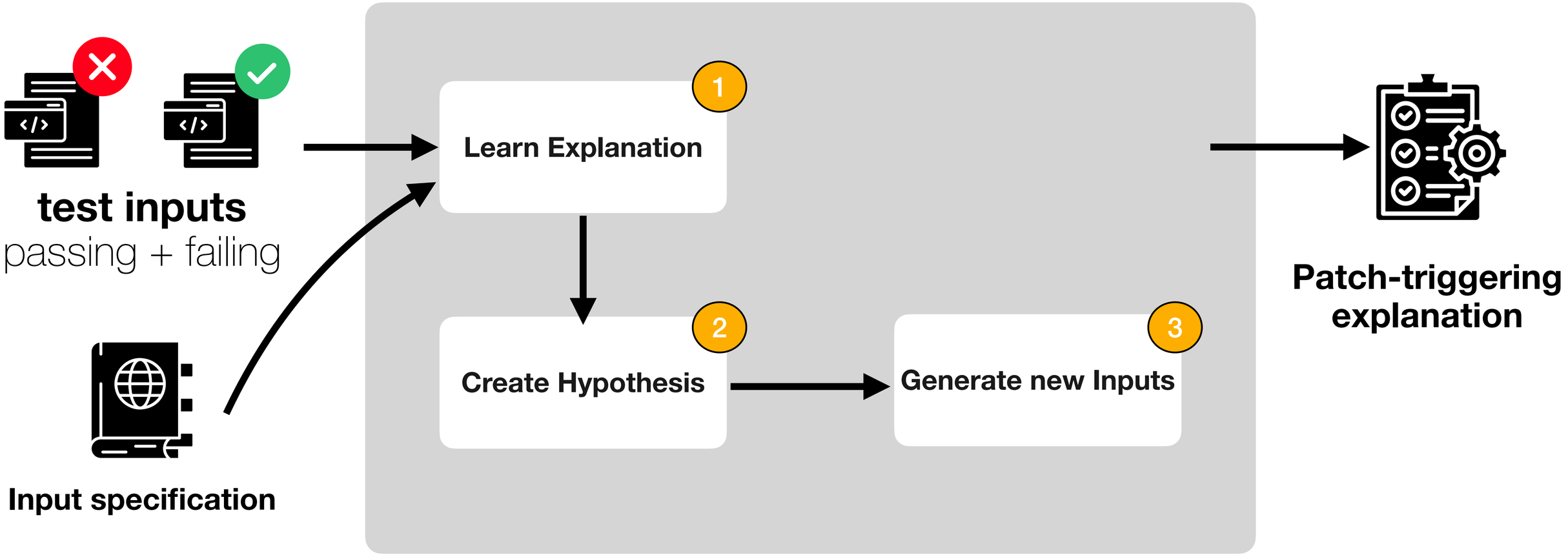
Automatic Explanations

Generating explanations to describe the input properties affected by a patch



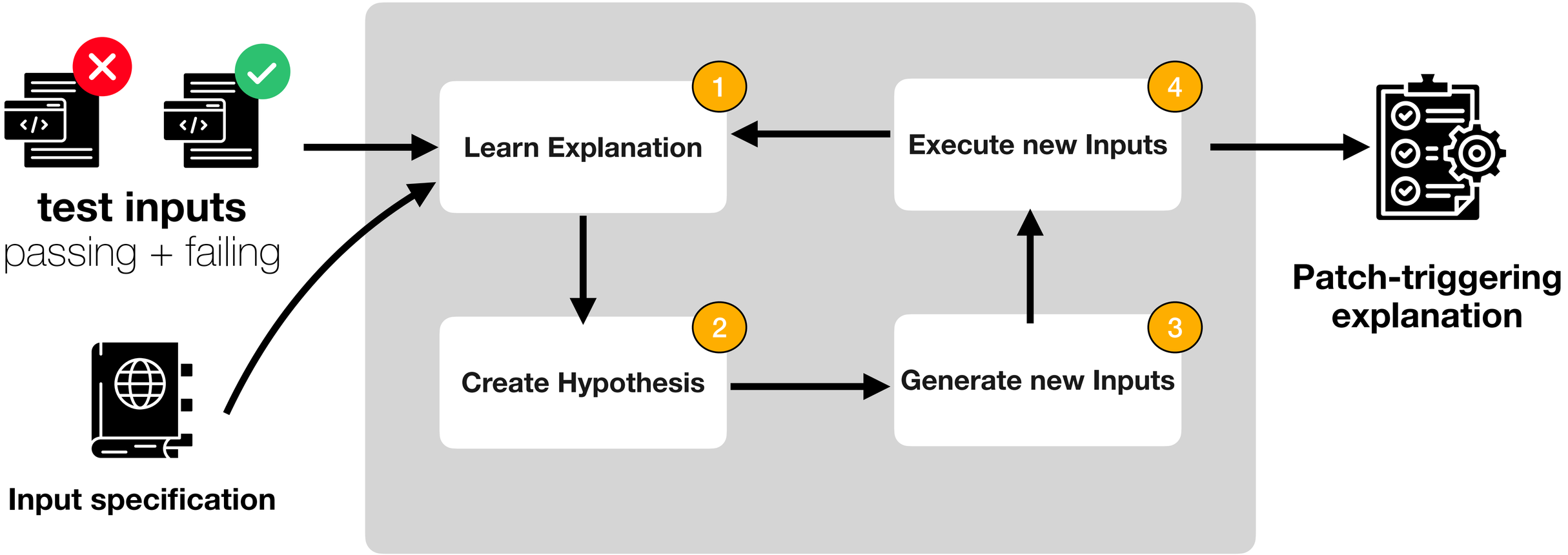
Automatic Explanations

Generating explanations to describe the input properties affected by a patch



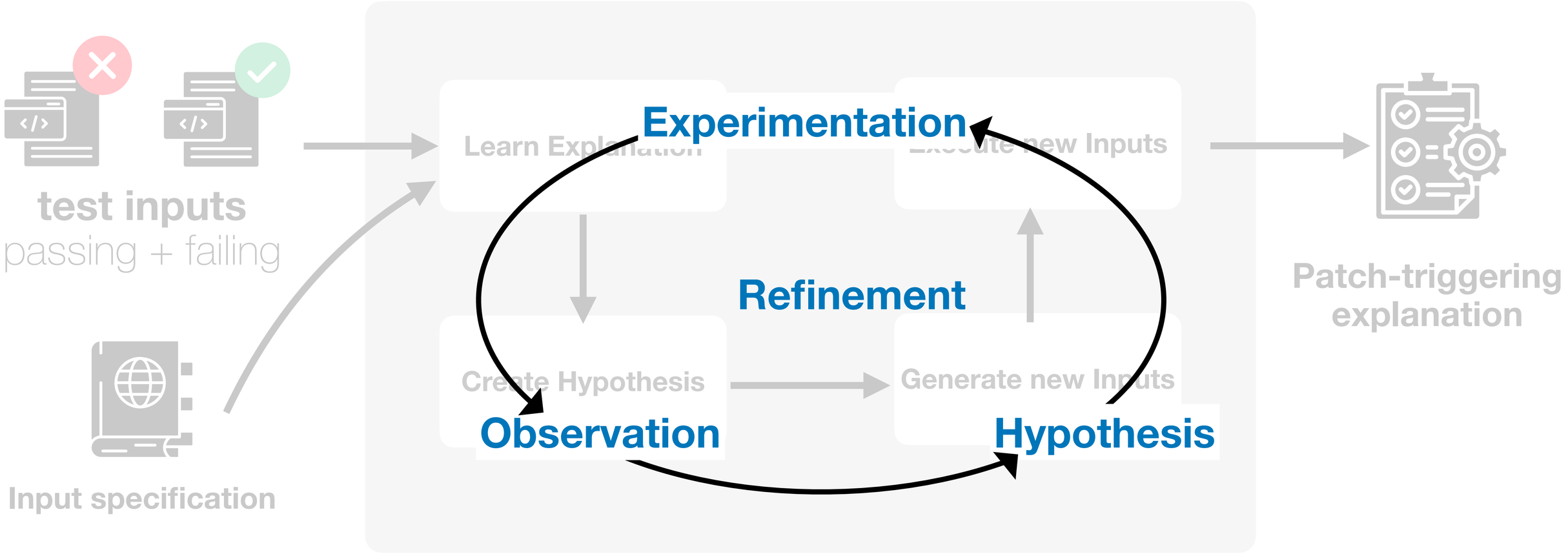
Automatic Explanations

Generating explanations to describe the input properties affected by a patch



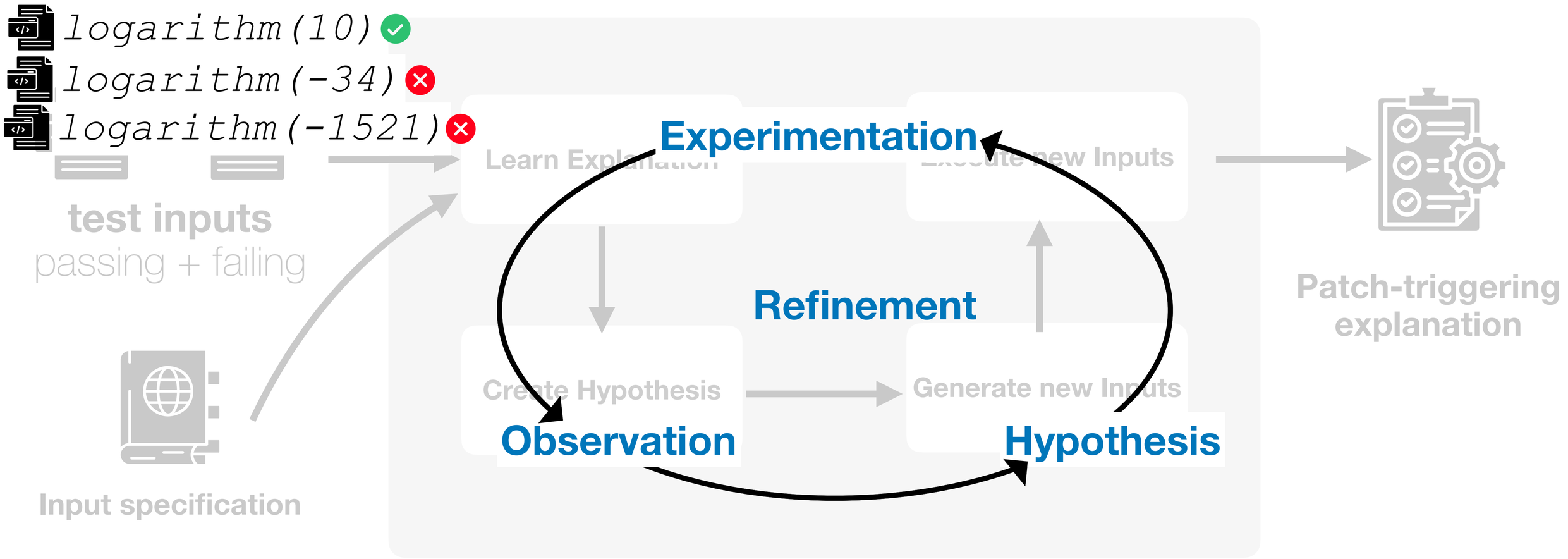
Automatic Explanations

Scientific Debugging



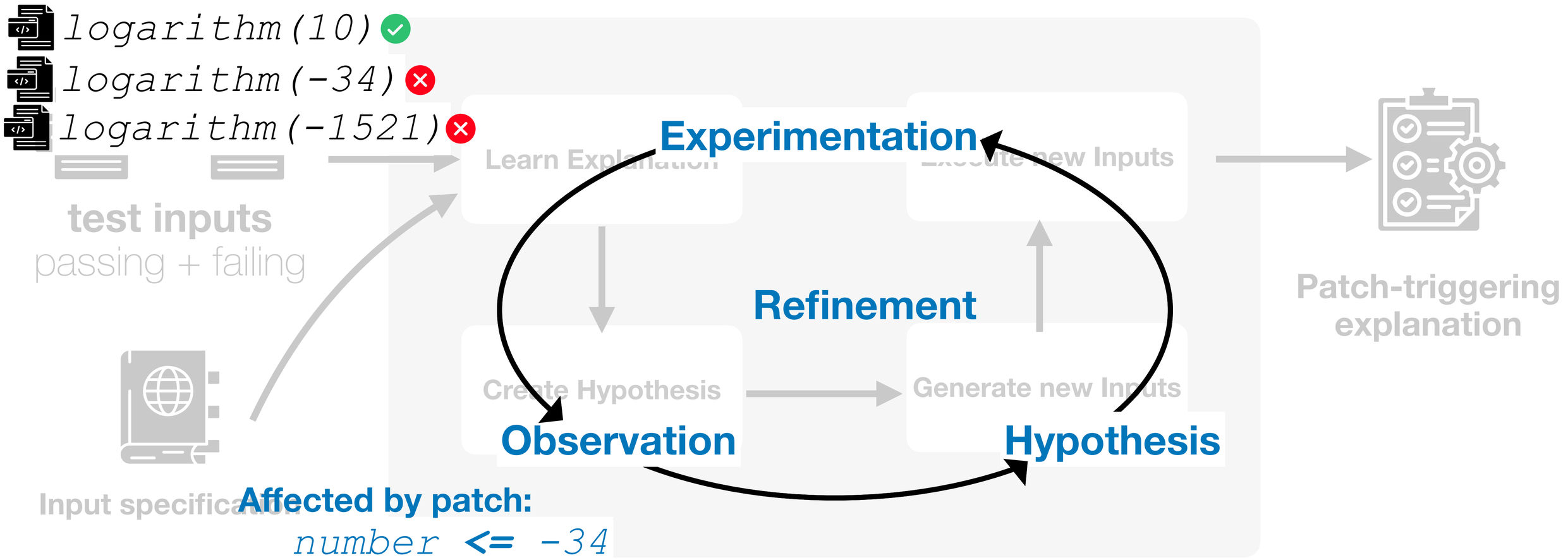
Automatic Explanations

Scientific Debugging



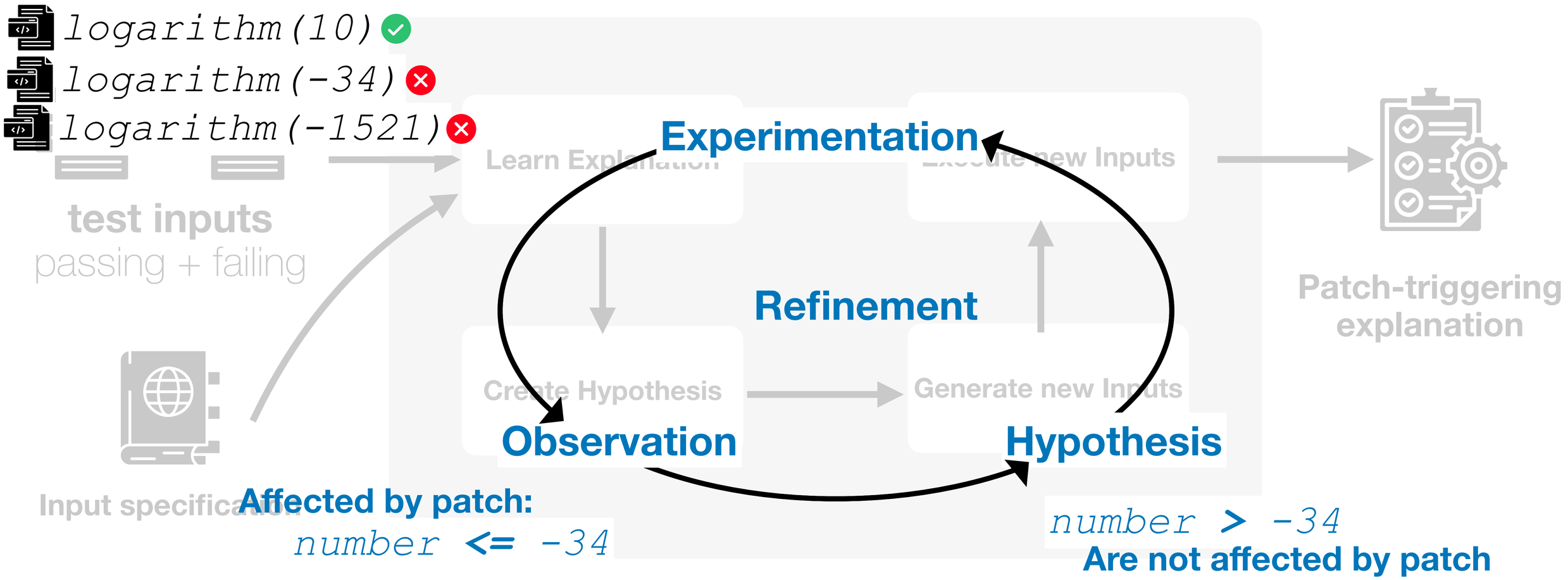
Automatic Explanations

Scientific Debugging



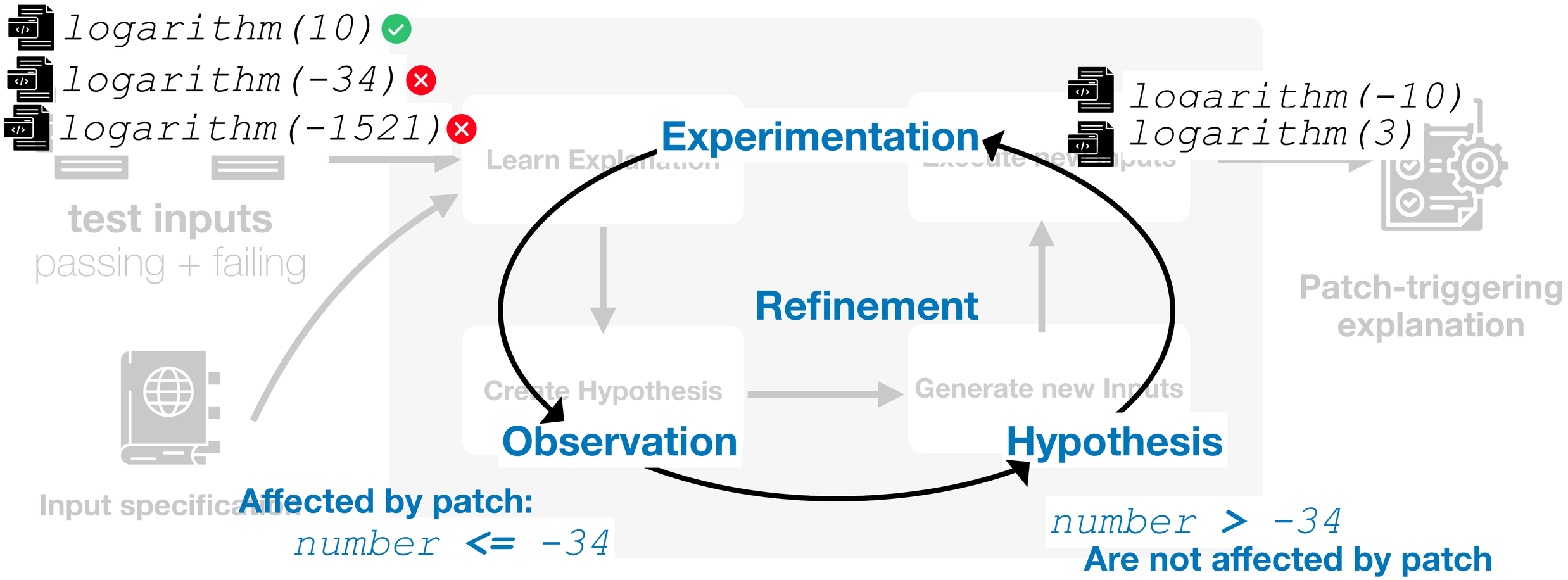
Automatic Explanations

Scientific Debugging



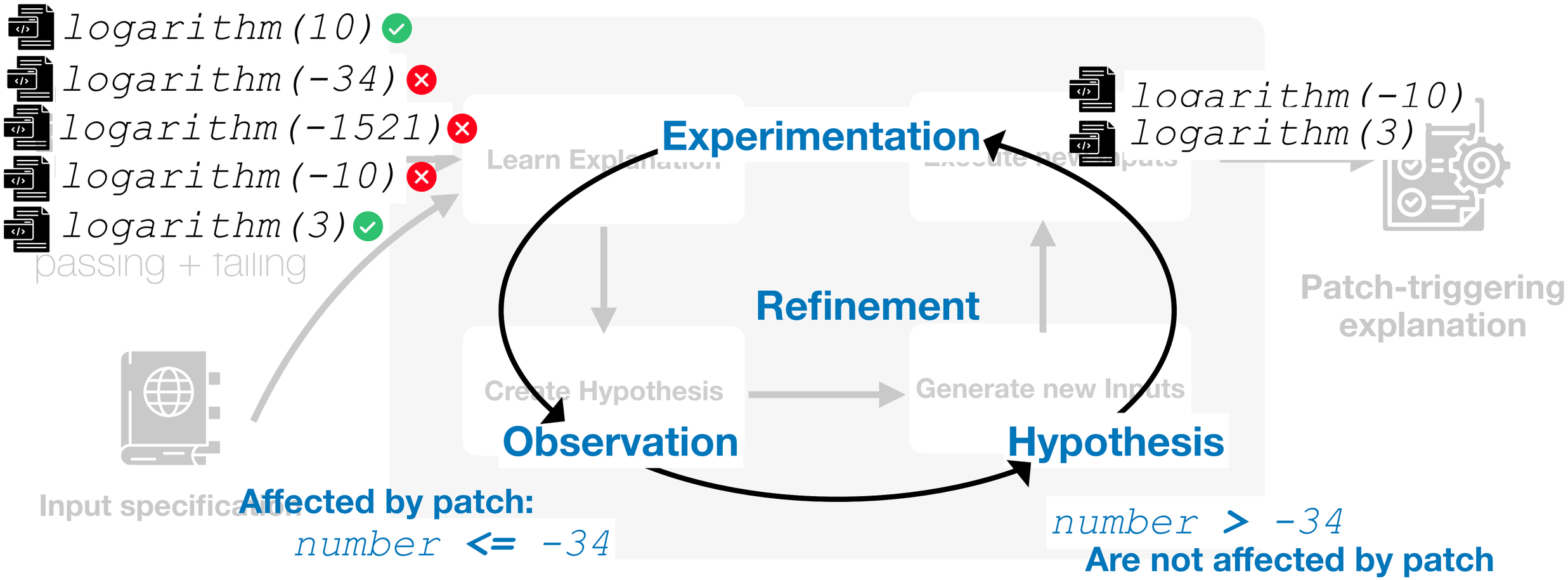
Automatic Explanations

Scientific Debugging



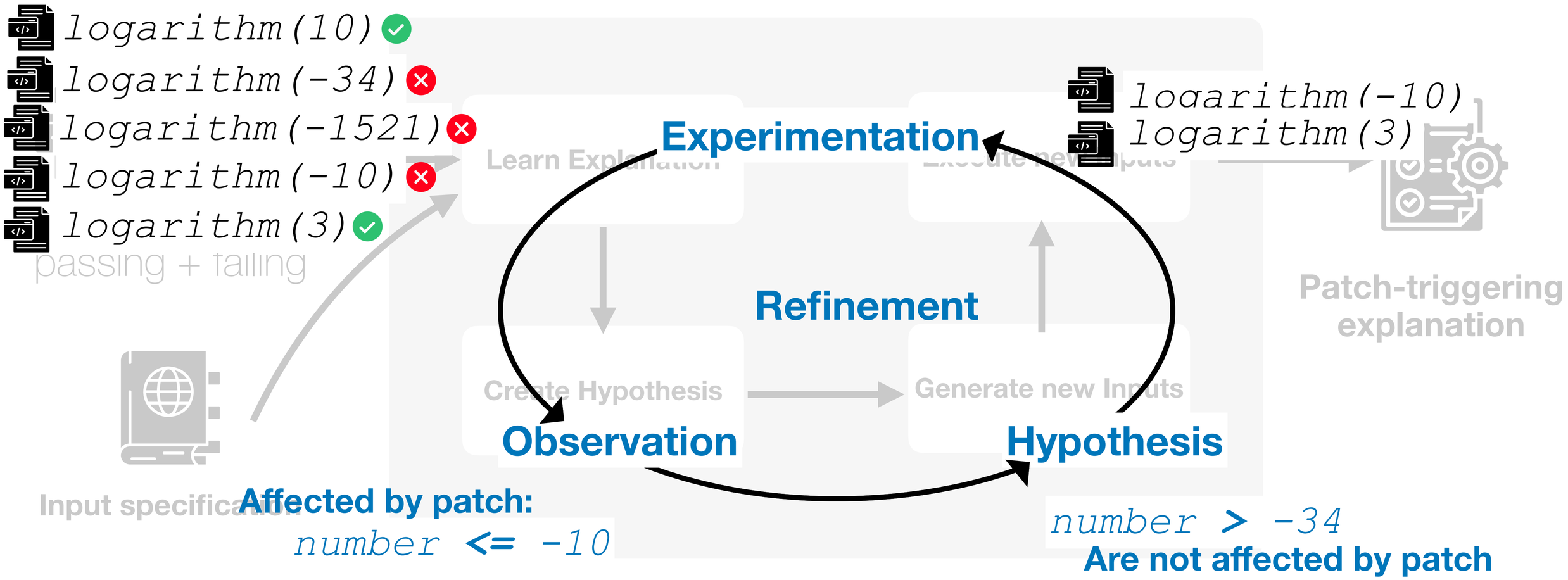
Automatic Explanations

Scientific Debugging



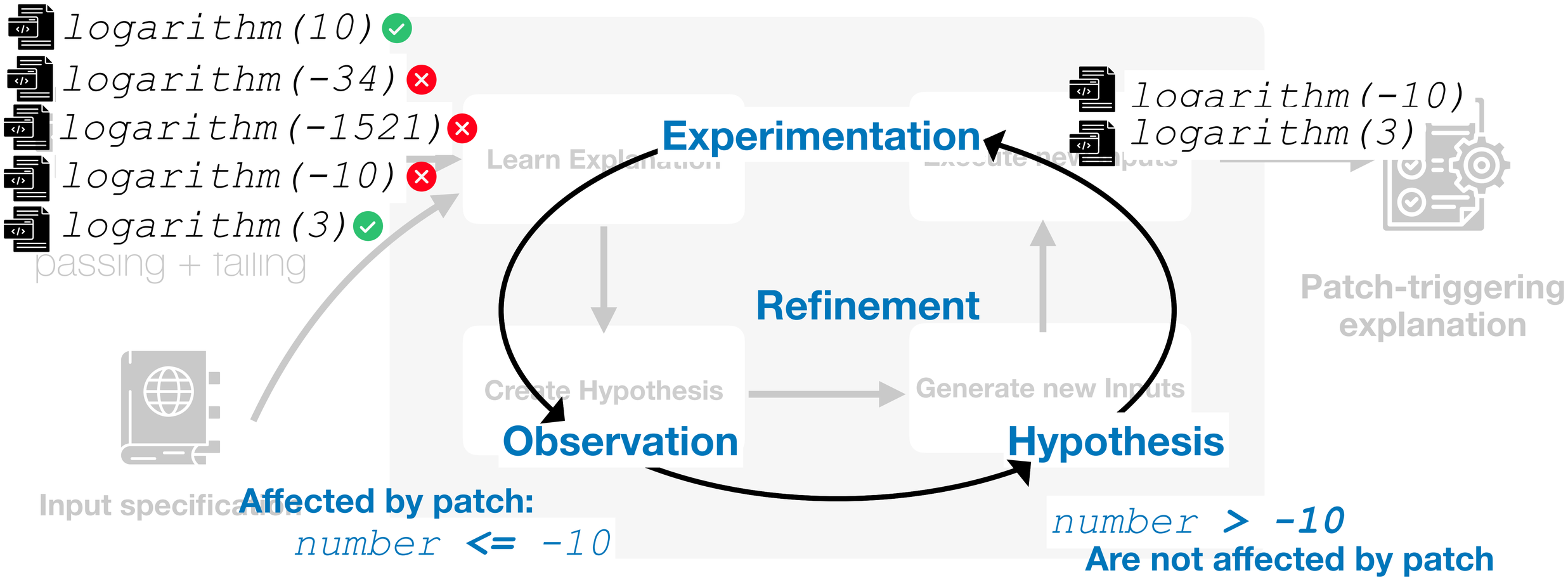
Automatic Explanations

Scientific Debugging



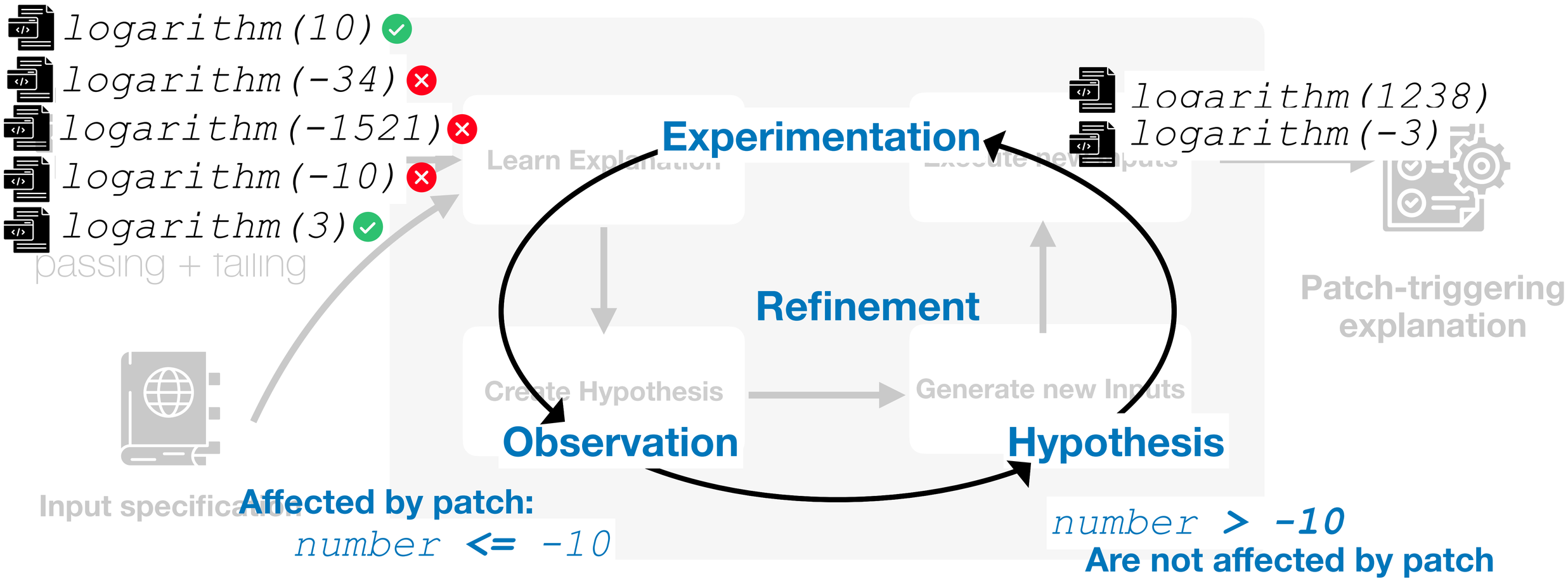
Automatic Explanations

Scientific Debugging



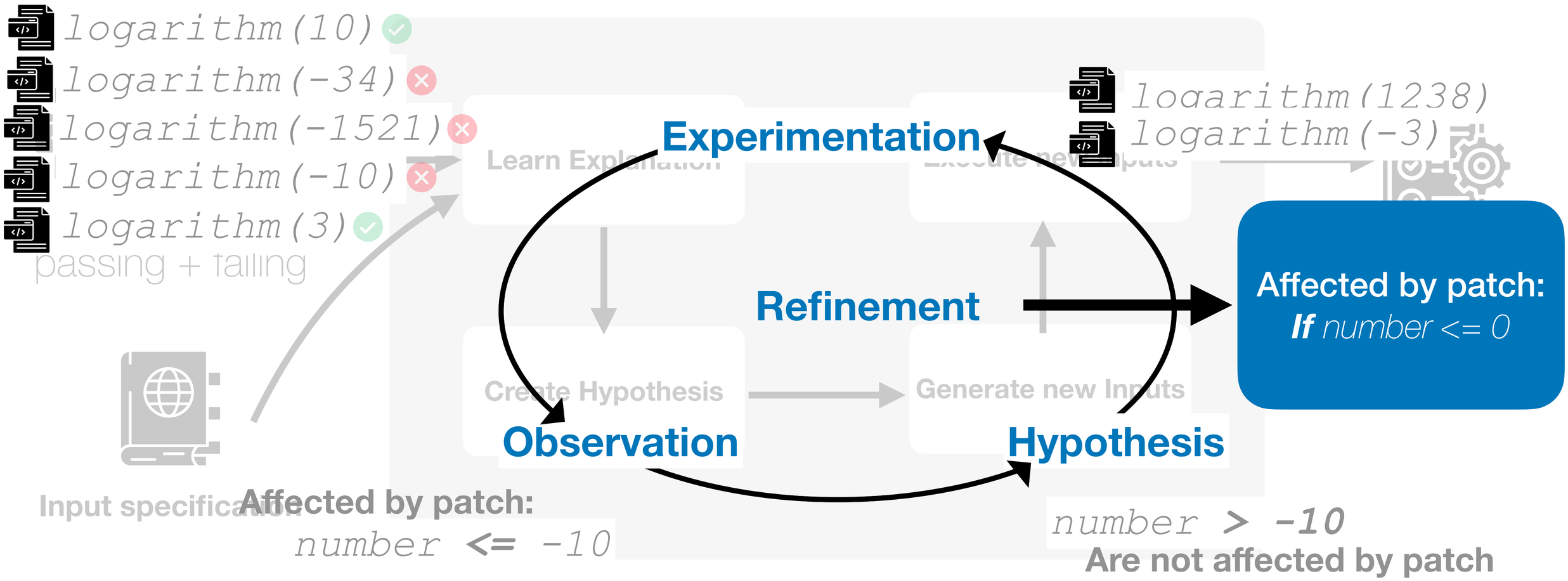
Automatic Explanations

Scientific Debugging



Automatic Explanations

Scientific Debugging



Benchmark Results

RQ1: Predictor

How accurately can the explanations predict whether an input belongs to the input space affected by the patch?

RQ2: Producer

How effectively can our explanations generate new inputs that belong to the input space affected by the patch?

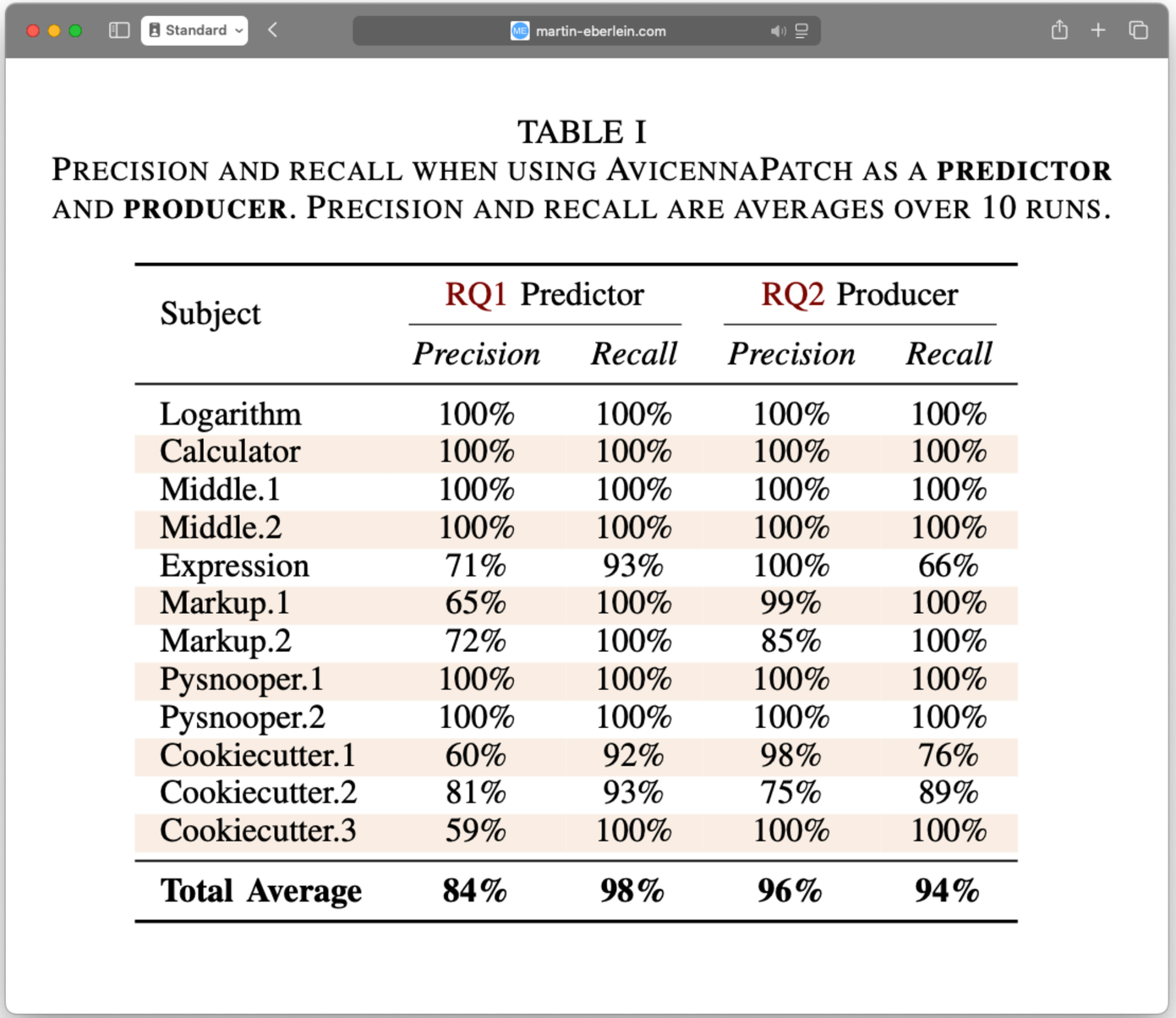


TABLE I

PRECISION AND RECALL WHEN USING AVICENNAPATCH AS A **PREDICTOR** AND **PRODUCER**. PRECISION AND RECALL ARE AVERAGES OVER 10 RUNS.

Subject	RQ1 Predictor		RQ2 Producer	
	Precision	*Recall*	*Precision*	*Recall*
Logarithm	100%	100%	100%	100%
Calculator	100%	100%	100%	100%
Middle.1	100%	100%	100%	100%
Middle.2	100%	100%	100%	100%
Expression	71%	93%	100%	66%
Markup.1	65%	100%	99%	100%
Markup.2	72%	100%	85%	100%
Pysnooper.1	100%	100%	100%	100%
Pysnooper.2	100%	100%	100%	100%
Cookiecutter.1	60%	92%	98%	76%
Cookiecutter.2	81%	93%	75%	89%
Cookiecutter.3	59%	100%	100%	100%
Total Average	**84%**	**98%**	**96%**	**94%**

Limitations

Vocabulary & Pattern Catalogue

Reliance on an appropriate vocabulary to express the conditions under which a patch is triggered.

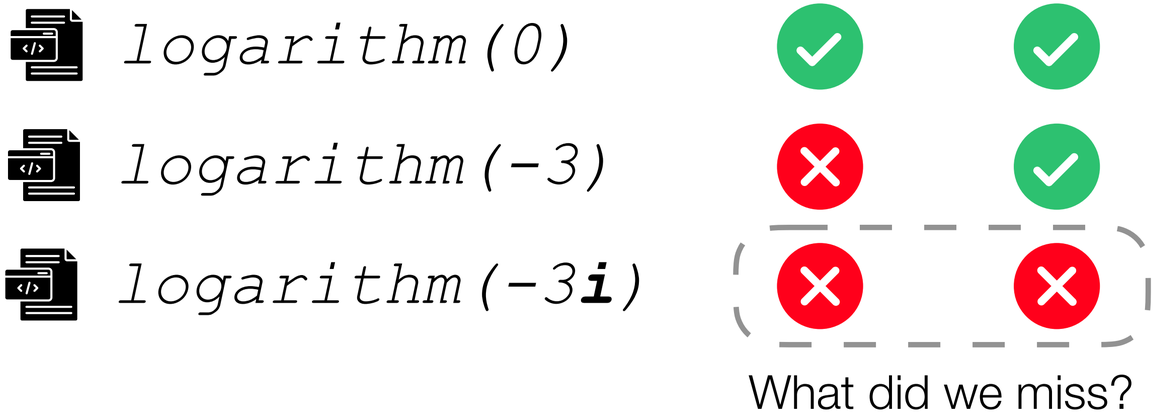
Requires Grammar/Specifaction

Explanations are constructed based on the provided grammar. Grammars may not always be available.

Next Steps?

Partial and Incomplete Patches

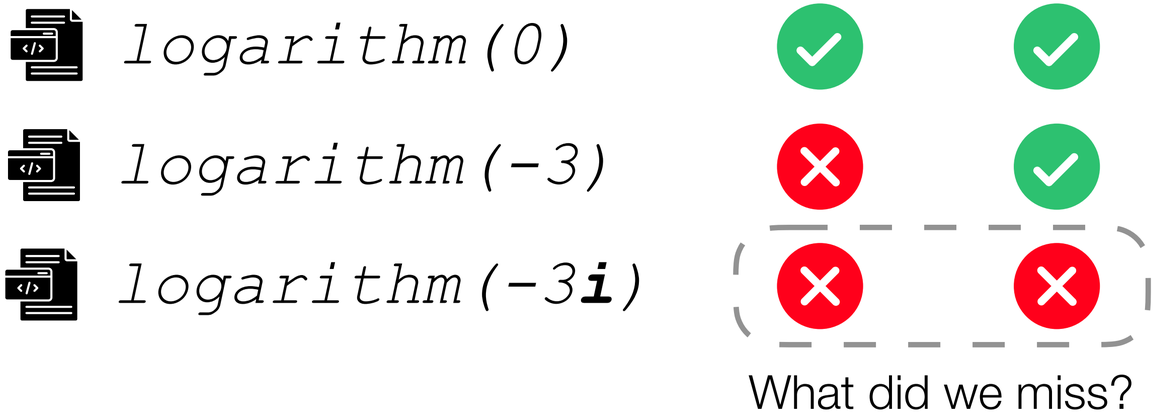
Extended to identify partial or incomplete patches by analyzing the properties of failing inputs unaffected by the patch



Next Steps?

Partial and Incomplete Patches

Extended to identify partial or incomplete patches by analyzing the properties of failing inputs unaffected by the patch



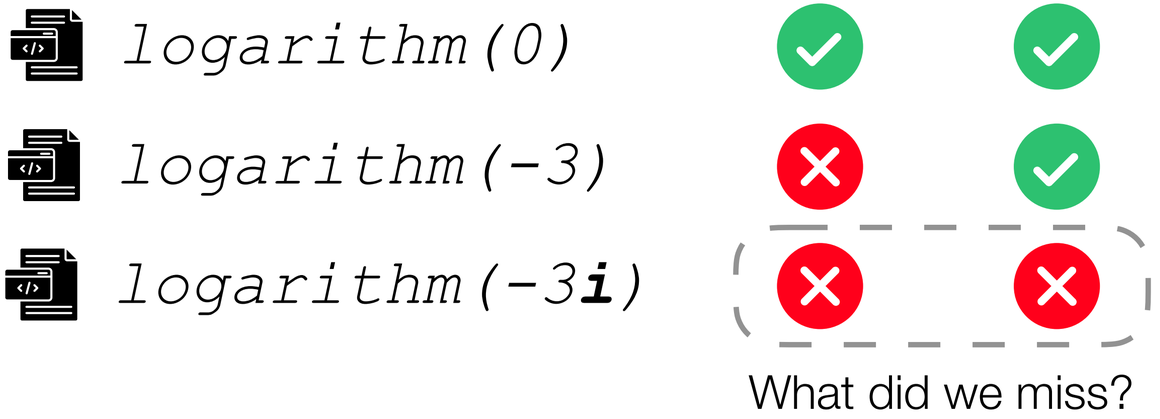
Integration with Symbolic Execution

Can the explanations be used to guide symbolic/concolic execution techniques?

Next Steps?

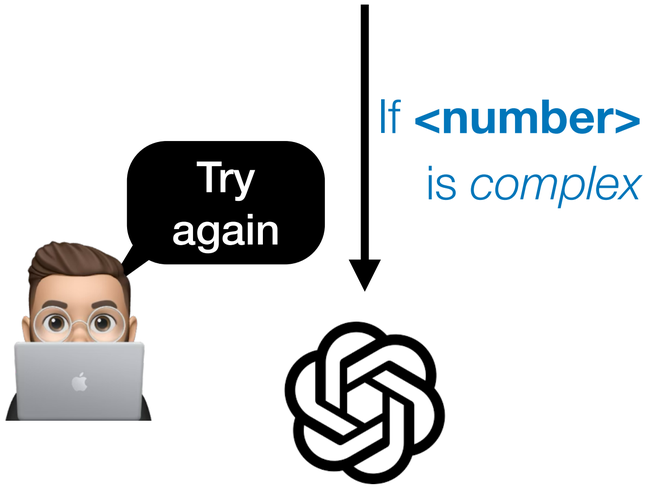
Partial and Incomplete Patches

Extended to identify partial or incomplete patches by analyzing the properties of failing inputs unaffected by the patch



Integration with Symbolic Execution

Can the explanations be used to guide symbolic/concolic execution techniques?

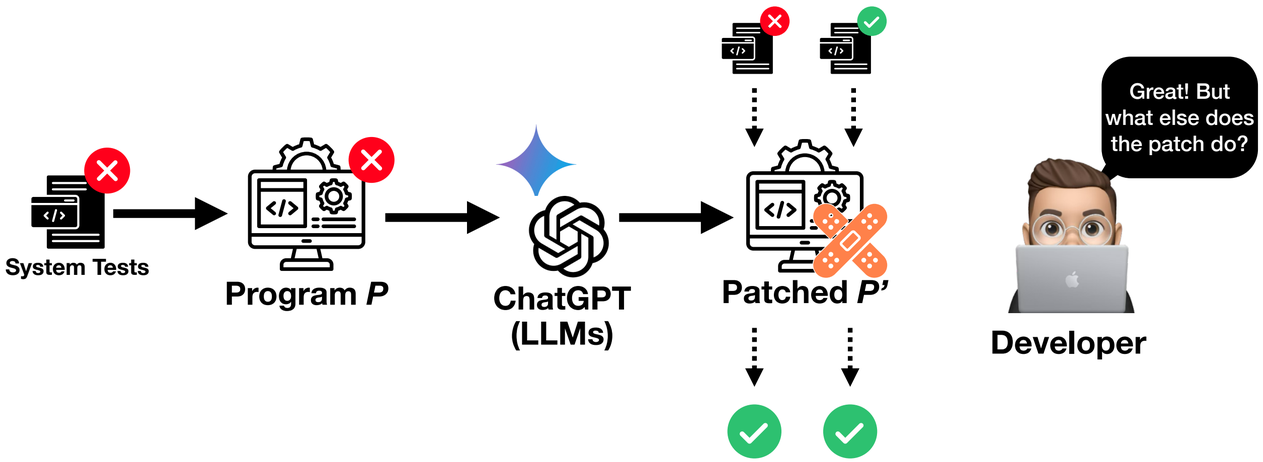


Additional Applications Scenarios

Automated Program Repair & Guide LLMs

Trust Issues?

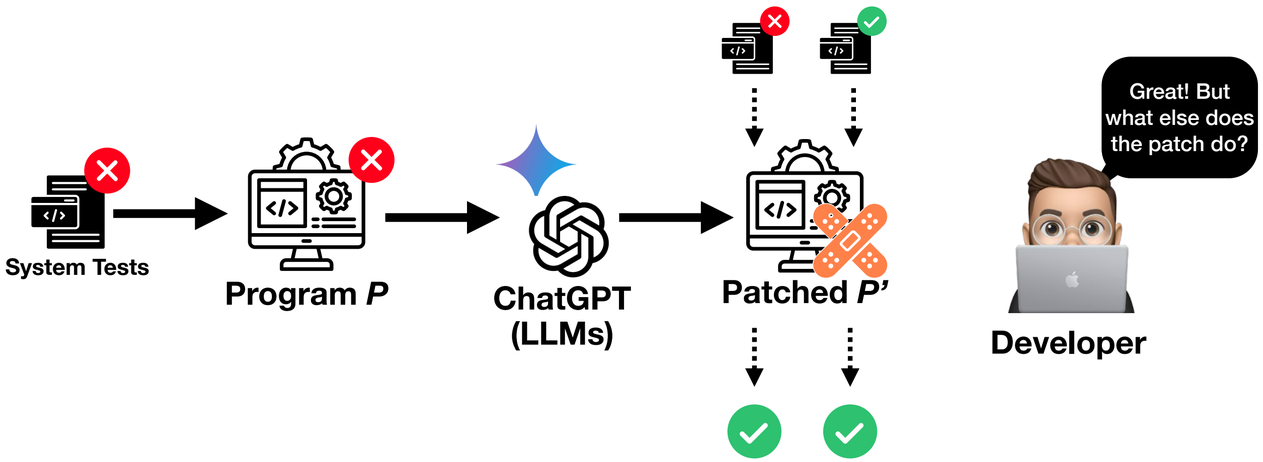
Would you trust a patch generated by ChatGPT to fix your critical code?



3

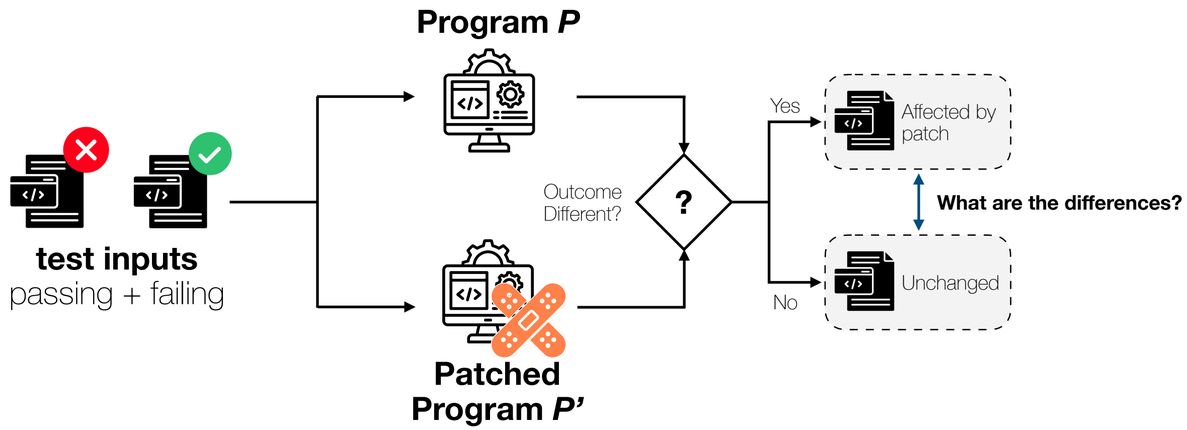
Trust Issues?

Would you trust a patch generated by ChatGPT to fix your critical code?



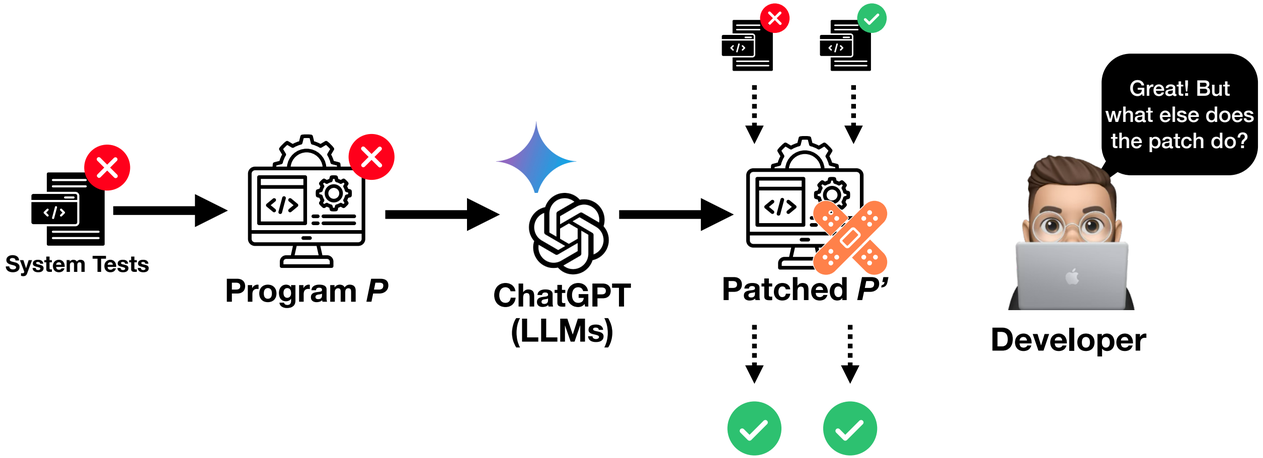
Automatic Explanations

Differential Behavior



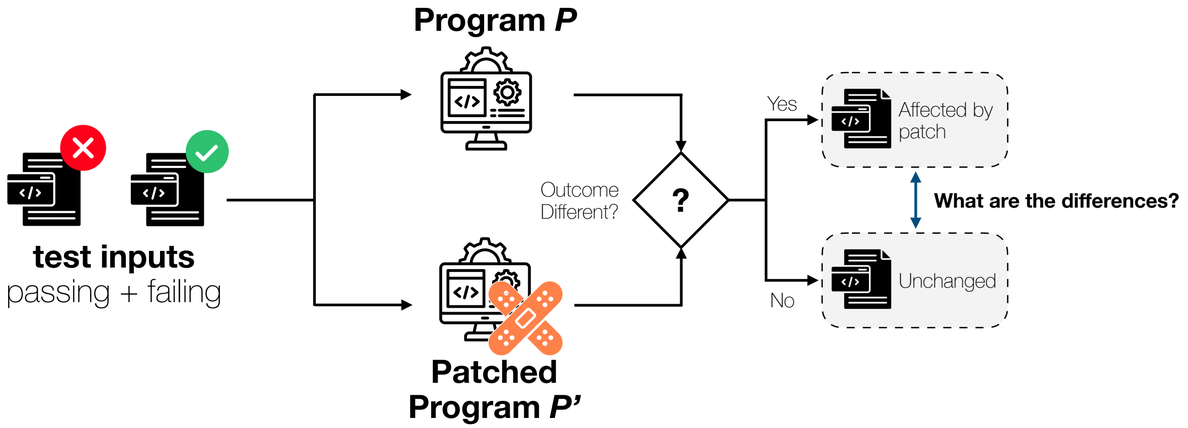
Trust Issues?

Would you trust a patch generated by ChatGPT to fix your critical code?



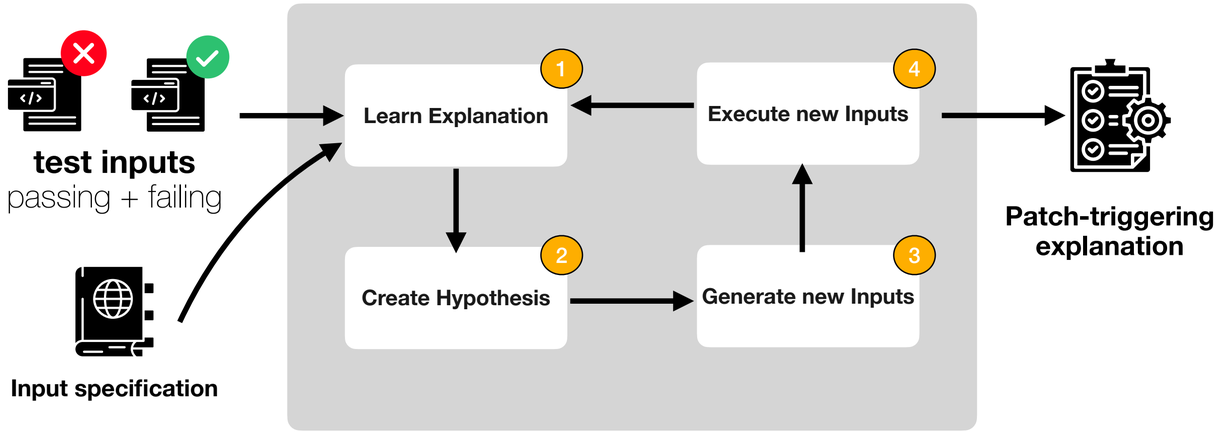
Automatic Explanations

Differential Behavior



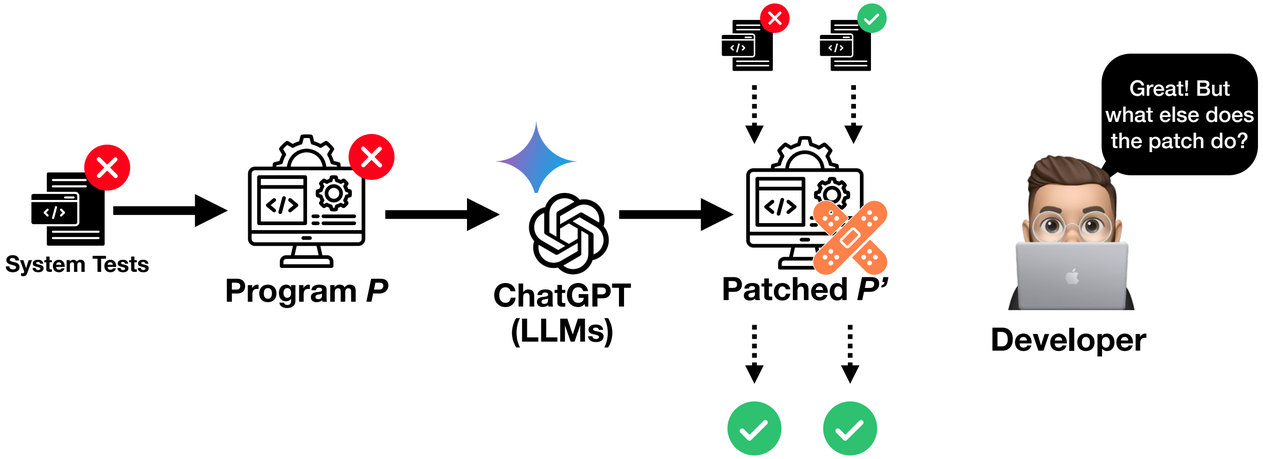
Automatic Explanations

Generating explanations to describe the input properties affected by a patch



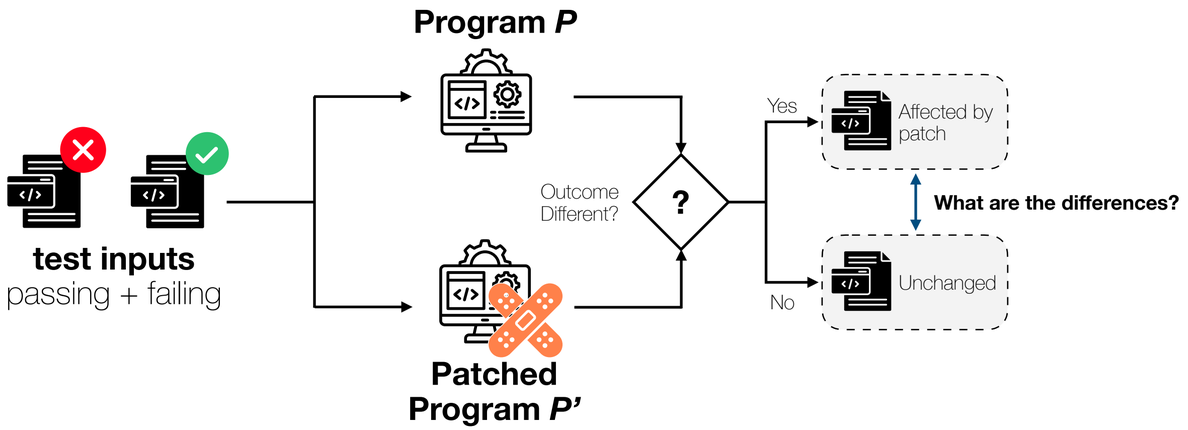
Trust Issues?

Would you trust a patch generated by ChatGPT to fix your critical code?



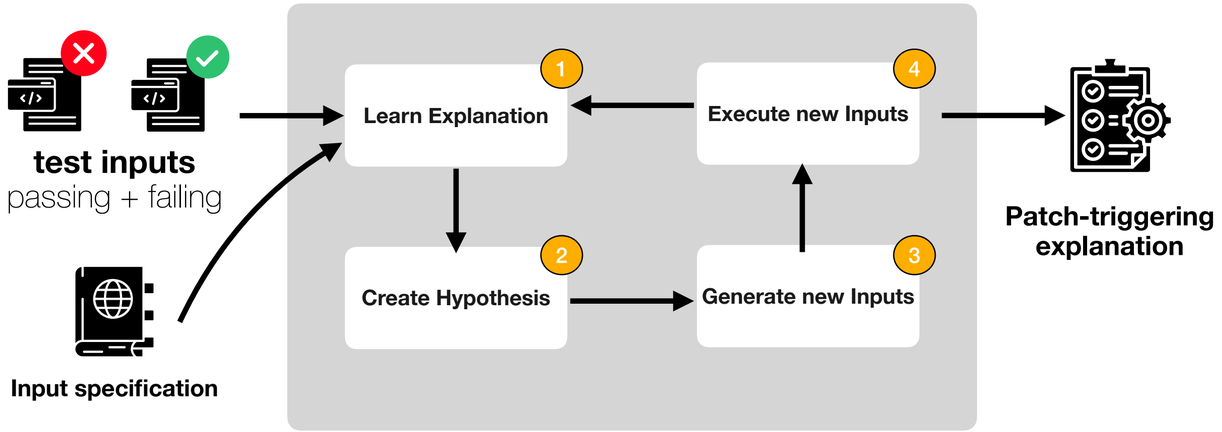
Automatic Explanations

Differential Behavior



Automatic Explanations

Generating explanations to describe the input properties affected by a patch



Benchmark Results

RQ1: Predictor

How accurately can the explanations predict whether an input belongs to the input space affected by the patch?

RQ2: Producer

How effectively can our explanations generate new inputs that belong to the input space affected by the patch?

TABLE I

PRECISION AND RECALL WHEN USING AVICENNAPATCH AS A **PREDICTOR** AND **PRODUCER**. PRECISION AND RECALL ARE AVERAGES OVER 10 RUNS.

Subject	RQ1 Predictor		RQ2 Producer	
	Precision	*Recall*	*Precision*	*Recall*
Logarithm	100%	100%	100%	100%
Calculator	100%	100%	100%	100%
Middle.1	100%	100%	100%	100%
Middle.2	100%	100%	100%	100%
Expression	71%	93%	100%	66%
Markup.1	65%	100%	99%	100%
Markup.2	72%	100%	85%	100%
Pysnooper.1	100%	100%	100%	100%
Pysnooper.2	100%	100%	100%	100%
Cookiecutter.1	60%	92%	98%	76%
Cookiecutter.2	81%	93%	75%	89%
Cookiecutter.3	59%	100%	100%	100%
Total Average	**84%**	**98%**	**96%**	**94%**

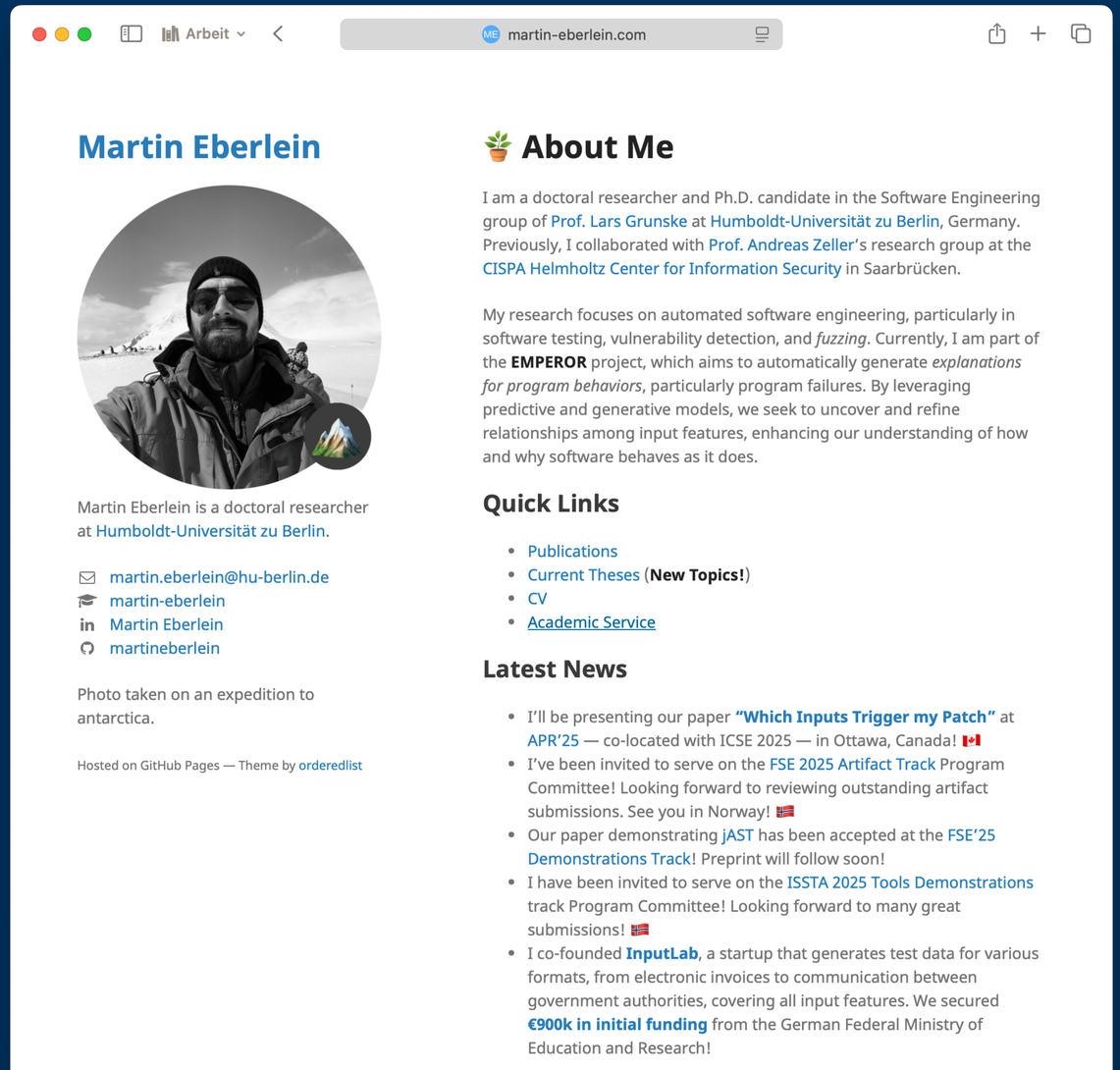
Contact

Website
https://martin-eberlein.com

Mail
martin.eberlein@hu-berlin.de

Github
martineberlein

LinkedIn
Martin Eberlein



Describing Input Properties

Input Specification Language

Input Specification Language

„Input Invariants“: Dominic Steinhöfel & Andreas Zeller, ESEC/FSE 2022

Syntax
Input Grammar

```
⟨heartbeat-request⟩ ::= 0x1 ⟨length⟩ ⟨payload⟩ ⟨padding⟩

⟨heartbeat-response⟩ ::= 0x2 ⟨length⟩ ⟨payload⟩ ⟨padding⟩

⟨length⟩ ::= ⟨uint16⟩

⟨payload⟩ ::= ε | ⟨byte⟩ ⟨payload⟩

⟨padding⟩ ::= ε | ⟨byte⟩ ⟨padding⟩
```

Semantics
ISLa Constraints

```
int(⟨length⟩) <= 7

len(⟨payload⟩) ≠ 9

int(⟨length⟩) > len(⟨payload⟩)
```