Machine Programming

Lecture 15 – LLM for Software Testing

Logistics – Week 9

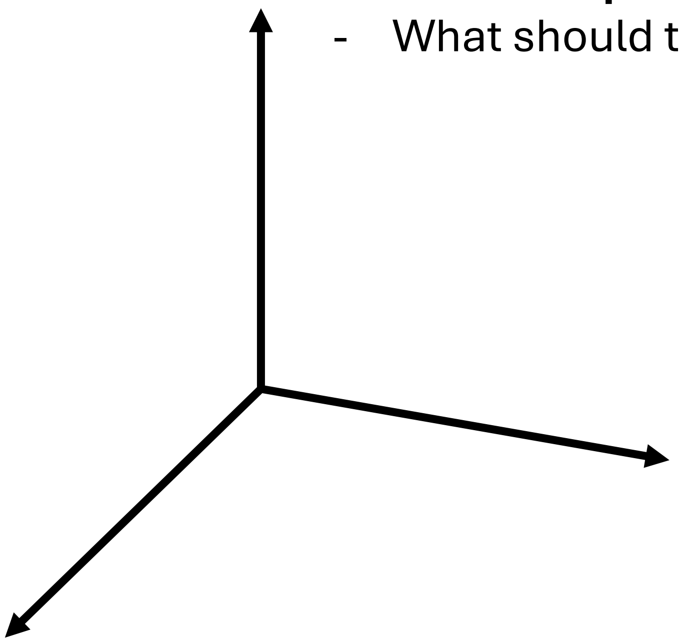
- Assignment 3: Coding LLM Agents
  - https://github.com/machine-programming/assignment-3
  - Fully functional web-app agent. Due: Oct 23 (Thu)
- Forming groups for your final projects!
  - Form a group of 2-3 before This Thursday (~~Oct 19~~ Oct 23)

Review

- What we have learned so far
  - Programming languages and synthesis: syntax, semantics, enumeration
  - Programming with LLMs: language modeling, prompting, agents, training

Dimensions in Program Synthesis

Behavioral Specification
- What should the program do?



Synthesis Strategy
- How do we find such a program?

Structural Specification
- What is the space of the programs?

The Course So Far

Behavioral Specification
- What should the program do?

1. Examples
2. Types
3. Functional Specifications
4. Natural Language

Synthesis Strategy
- How do we find such a program?

Enumeration: Bottom-up and Top-Down

Programming with Large Language Models
- Next token prediction, prompting, controlled decoding
- Iterative refinement, agentic frameworks and tool use
- Pre-training, fine-tuning, reinforcement learning

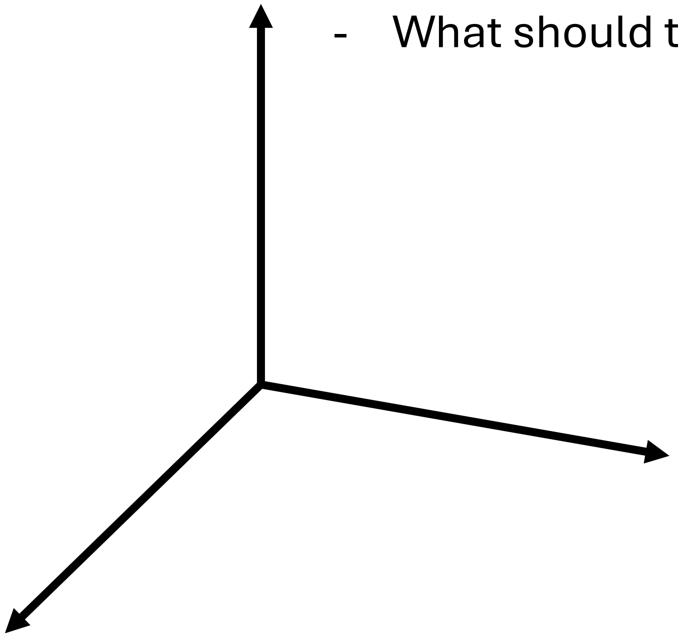
Structural Specification
- What is the space of the programs?

General Purpose Programming Language
Python / Java / C / Rust / …

Custom Domain Specific Languages

Module 3

Behavioral Specification
- What should the program do?



Structural Specification
- What is the space of the programs?

Synthesis Strategy
- How do we find such a program?

Enumeration: Bottom-up and Top-Down

Programming with Large Language Models
- Next token prediction, prompting, controlled decoding
- Iterative refinement, agentic frameworks and tool use
- Pre-training, fine-tuning, reinforcement learning

Fixed

Module 3

Behavioral Specification
- What should the program do?

Synthesis Strategy
- How do we find such a program?

Enumeration: Bottom-up and Top-Down

Programming with Large Language Models
- Next token prediction, prompting, controlled decoding
- Iterative refinement, agentic frameworks and tool use
- Pre-training, fine-tuning, reinforcement learning

Fixed

Structural Specification
- What is the space of the programs?

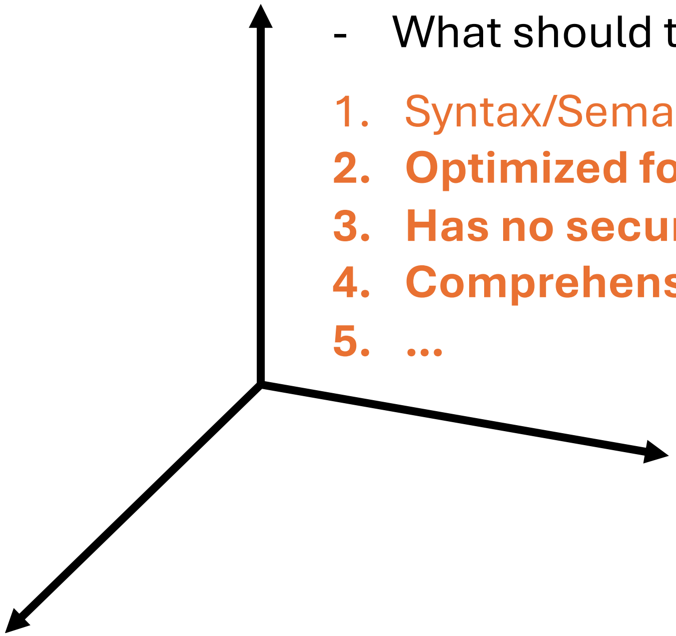
General Purpose Programming Language
Python / Java / C / Rust / …

Domain Specific Languages
SQL / LEAN / ROCQ / DATALOG / PDDL / …

Module 3

Behavioral Specification
- What should the program do?

1. Syntax/Semantics/Functional correctness
2. **Optimized for runtime speed**
3. **Has no security flaw**
4. **Comprehensive test coverage**
5. **...**



Synthesis Strategy
- How do we find such a program?

Enumeration: Bottom-up and Top-Down

Programming with Large Language Models
- Next token prediction, prompting, controlled decoding
- Iterative refinement, agentic frameworks and tool use
- Pre-training, fine-tuning, reinforcement learning

Fixed

Structural Specification
- What is the space of the programs?

General Purpose Programming Language
Python / Java / C / Rust / ...

Domain Specific Languages
SQL / LEAN / ROCQ / DATALOG / PDDL / ...

Module 3: Overview

- What we have learned so far
  - Programming languages and synthesis: syntax, semantics, enumeration
  - Programming with LLMs: language modeling, prompting, agents, training
- Module 3: Applications of Machine Programming
  - Goal: how synthesis can help with diverse applications
  - Application: Software analysis, testing, optimization, security, verification
  - Application: Logic and Formal Methods, Math, Physics, Science
  - Application: Embodied systems, robotics, planning, reward modeling
  - Application: UI, interactive environment, games

Software Analysis

Testing and Dynamic Analysis

```
BH_I16 := Integer_16(BH_F64);
```

```
BH_I16 := Integer_16(BH_F64);
```

Ada: covert 64-bit floating point number to 16-bit signed integer

The Ariane Rocket Disaster

Date: June 4, 1996
Mission: Ariane 5 Flight 501
- European Space Agency

https://www.youtube.com/watch?v=PK_yguLapgA&t=63s

The Ariane Rocket Disaster: Post Mortem

- Outcome:
  - Rocket veered off course after 37 seconds, self-destructed.

The Ariane Rocket Disaster: Post Mortem

- Outcome:
  - Rocket veered off course after 37 seconds, self-destructed.
- What happened:
  - The rocket reused Ariane 4's Inertial Reference System (SRI) software.
  - During ascent, the SRI computed a variable BH (Horizontal Bias), representing horizontal velocity.
  - The program executed:

```
BH_I16 := Integer_16(BH_F64);
```

The Ariane Rocket Disaster: Post Mortem

- Outcome:
  - Rocket veered off course after 37 seconds, self-destructed.
- What happened:
  - The rocket reused Ariane 4's Inertial Reference System (SRI) software.
  - During ascent, the SRI computed a variable BH (Horizontal Bias), representing horizontal velocity.
  - The program executed:

```
BH_I16 := Integer_16(BH_F64);
```

  - On Ariane 4, `BH_F64` never exceeded 32767.
  - On Ariane 5 (more powerful), `BH_F64` ≈ 65535, causing a numeric overflow.
  - In Ada, this raised a `Constraint_Error` exception (like an "Operand Error").

The Ariane Rocket Disaster: Post Mortem

- Outcome:
  - Rocket veered off course after 37 seconds, self-destructed.
  - Cost: \$350M payload + \$150M rocket
- Have we programmers learned from the mistake?

Common Weakness Enumeration

A community-developed list of SW & HW weaknesses that can become vulnerabilities

New to CWE? Start here!

2024 CWE Top 25 Most Dangerous Software Weaknesses

Top 25 Home | Share via: X | View in table format | Key Insights | Methodology

1 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CWE-79 | CVEs in KEV: 3 | Rank Last Year: 2 (up 1) ▲

2 Out-of-bounds Write
CWE-787 | CVEs in KEV: 18 | Rank Last Year: 1 (down 1) ▼

3 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
CWE-89 | CVEs in KEV: 4 | Rank Last Year: 3

...

22 Use of Hard-coded Credentials
CWE-798 | CVEs in KEV: 2 | Rank Last Year: 18 (down 4) ▼

23 Integer Overflow or Wraparound
CWE-190 | CVEs in KEV: 3 | Rank Last Year: 14 (down 9) ▼

24 Uncontrolled Resource Consumption
CWE-400 | CVEs in KEV: 0 | Rank Last Year: 37 (up 13) ▲

https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html

Common Weakness Enumeration

A community-developed list of SW & HW weaknesses that can become vulnerabilities

Top 25 | Top HW CWE | **New to CWE?** Start here!

Home > CWE Top 25 > 2024 | ID Lookup: Go

Home | About ▼ | Learn ▼ | Access Content ▼ | Community ▼ | Search ▼

2024 CWE Top 25 Most Dangerous Software Weaknesses

Top 25 Home | Share via: X | View in table format | Key Insights | Methodology

1 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CWE-79 | CVEs in KEV: 3 | Rank Last Year: 2 (up 1) ▲

2 Out-of-bounds Write
CWE-787 | CVEs in KEV: 18 | Rank Last Year: 1 (down 1) ▼

3 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
CWE-89 | CVEs in KEV: 4 | Rank Last Year: 3

...

22 Use of Hard-coded Credentials
CWE-798 | CVEs in KEV: 2 | Rank Last Year: 18 (down 4) ▼

23 Integer Overflow or Wraparound
CWE-190 | CVEs in KEV: 3 | Rank Last Year: 14 (down 9) ▼

24 Uncontrolled Resource Consumption
CWE-400 | CVEs in KEV: 0 | Rank Last Year: 37 (up 13) ▲

https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html

Goals of Software Analysis

- Question:
  - If Ariane 5's code compiled cleanly, why did it still explode?
- Idea:
  - Software analysis $\neq$ compile without error/warning
  - Search for hidden failure modes before deployment

Goals of Software Analysis

Goal	Example	Early Warning Signal
Detect Bugs	Off-by-one, numeric overflow, null dereference, out-of-bounds	Static range warnings, failed test cases
Detect Vulnerabilities	SQL-injection, buffer overflow, path traversal, incorrect permission	CWE-based pattern matching
Check Correctness	Functional specifications, Algorithmic invariants	Violated assertions
Verification	Know whether a functional specification "always holds"	Verifier error, symbolic execution traces
Meaure Reliability	Coverage (line, block, path), test pass rate	Uncovered lines, failed test cases

Forms of Software Analysis

Static Analysis	Dynamic Analysis

Forms of Software Analysis

Static Analysis

Examine code without concrete execution

Dynamic Analysis

Observes program behavior while executing

Forms of Software Analysis

Static Analysis

Examine code without concrete execution

- Taint analysis
- Reachability analysis
- Abstract interpretation
- Symbolic execution
- ...

Dynamic Analysis

Observes program behavior while executing

- Unit testing
- Fuzzing
- Property-based testing
- Penetration testing
- ...

Forms of Software Analysis

Dynamic Analysis

(This Lesson)

Observes program behavior while executing

- Unit testing
- Fuzzing
- Property-based testing
- Penetration testing

Dynamic Analysis

- Some bugs only manifest when code runs
  - Runtime errors
  - Unhandled exceptions
  - Errors dependent on external environments
- Want to concretely execute code
  - Observe program behavior
  - Analyze potential mistakes

Dynamic Testing

Unit Testing

Human-written tests
in the form of concrete
input-output

```
assert f(2) == 4
```

Dynamic Testing

Unit Testing

Human-written tests in the form of concrete input-output

```
assert f(2) == 4
```

Fuzzing

Randomly generating test inputs

```
for i in 0..10^4:
  x = gen_input()
  f(x)
```

Dynamic Testing

Unit Testing

Human-written tests in the form of concrete input-output

```
assert f(2) == 4
```

Fuzzing

Randomly generating test inputs

```
for i in 0..10^4:
  x = gen_input()
  f(x)
```

Property-based testing

Randomly generating test inputs, but with properties checking outputs

```
for i in 0..10^4:
  x = gen_input()
  y = f(x)
  assert property(x,y)
```

Dynamic Testing

Unit Testing

Human-written tests in the form of concrete input-output

```
assert f(2) == 4
```

Fuzzing

Randomly generating test inputs

```
for i in 0..10^4:
  x = gen_input()
  f(x)
```

Property-based testing

Randomly generating test inputs, but with properties checking outputs

```
for i in 0..10^4:
  x = gen_input()
  y = f(x)
  assert property(x,y)
```

Penetration testing

Crafting adversarial inputs to trigger vulnerabilities rather than getting outputs

```
try:
  f(very_bad_input)
  assert false
except:
  assert no_bad_behavior
```

Unit Testing

Human-written tests in the form of concrete input-output

```
assert f(2) == 4
```

Fuzzing

Randomly generating test inputs

```
for i in 0..10^4:
  x = gen_input()
  f(x)
```

Property-based testing

Randomly generating test inputs, but with properties checking outputs

```
for i in 0..10^4:
  x = gen_input()
  y = f(x)
  assert property(x,y)
```

Penetration testing

Crafting adversarial inputs to trigger vulnerabilities rather than getting outputs

```
try:
  f(very_bad_input)
  assert false
except:
  assert no_bad_behavior
```

Human Effort: figure out input-outputs, write the test cases

Human Effort: write the input generator and fuzzing harness

Unit Testing

Human-written tests in the form of concrete input-output

```
assert f(2) == 4
```

Human Effort: figure out input-outputs, write the test cases

Fuzzing

Randomly generating test inputs

```
for i in 0..10^4:
  x = gen_input()
  f(x)
```

Property-based testing

Randomly generating test inputs, but with properties checking outputs

```
for i in 0..10^4:
  x = gen_input()
  y = f(x)
  assert property(x,y)
```

Penetration testing

Crafting adversarial inputs to trigger vulnerabilities rather than getting outputs

```
try:
  f(very_bad_input)
  assert false
except:
  assert no_bad_behavior
```

Human Effort: write the input generator and fuzzing harness

Unit Testing

Human-written tests in the form of concrete input-output

```
assert f(2) == 4
```

Fuzzing

Randomly generating test inputs

```
for i in 0..10^4:
  x = gen_input()
  f(x)
```

Human Effort: figure out input-outputs, write the test cases

Property-based testing

Randomly generating test inputs, but with properties checking outputs

```
for i in 0..10^4:
  x = gen_input()
  y = f(x)
  assert property(x,y)
```

Penetration testing

Crafting adversarial inputs to trigger vulnerabilities rather than getting outputs

```
try:
  f(very_bad_input)
  assert false
except:
  assert no_bad_behavior
```

Human Effort: write the input generator, harness and the property

Human Effort: write the input generator and fuzzing harness

Human Effort: figure out the very bad input, describe the bad behavior

Unit Testing

Human-written tests in the form of concrete input-output

```
assert f(2) == 4
```

Human Effort: figure out input-outputs, write the test cases

Fuzzing

Randomly generating test inputs

```
for i in 0..10^4:
  x = gen_input()
  f(x)
```

Property-based testing

Randomly generating test inputs, but with properties checking outputs

```
for i in 0..10^4:
  x = gen_input()
  y = f(x)
  assert property(x,y)
```

Human Effort: write the input generator, harness and the property

Penetration testing

Crafting adversarial inputs to trigger vulnerabilities rather than getting outputs

```
try:
  f(very_bad_input)
  assert false
except:
  assert no_bad_behavior
```

Unit Testing: Precision Shots

- Human effort
  - Figure out expected input/output pairs (multiple)
  - Write a test case for each input/output pairs (multiple)
- Goal
  - Maximize confidence via coverage (approx.: line coverage)
  - Execute precise, human-chosen inputs that exercise specific code paths
  - Error handling is a path too!

Program – Line coverage: 0/5 (0%)

```python
def calculate_interest(balance: float, months: int) -> float:
  rate = 0.02 # 2% per month
  if months > 12:
    rate = 0.03 # higher rate for long-term deposits
  return balance * rate / months
```

Program – Line coverage: 0/5 (0%)

```python
def calculate_interest(balance: float, months: int) -> float:
  rate = 0.02 # 2% per month
  if months > 12:
    rate = 0.03 # higher rate for long-term deposits
  return balance * rate / months
```

Test 1

```python
def test_short_term_interest():
  assert calculate_interest(1000, 6) == pytest.approx(3.33)
```

Program – Line coverage: 3/5 (60%)

```python
def calculate_interest(balance: float, months: int) -> float:
  rate = 0.02 # 2% per month
  if months > 12:
    rate = 0.03 # higher rate for long-term deposits
  return balance * rate / months
```

Test 1

```python
def test_short_term_interest():
  assert calculate_interest(1000, 6) == pytest.approx(3.33)
```

Program – Line coverage: 3/5 (60%)

```python
def calculate_interest(balance: float, months: int) -> float:
  rate = 0.02 # 2% per month
  if months > 12:
    rate = 0.03 # higher rate for long-term deposits
  return balance * rate / months
```

Test 1

```python
def test_short_term_interest():
  assert calculate_interest(1000, 6) == pytest.approx(3.33)
```

Test 2

```python
def test_long_term_interest():
  assert calculate_interest(1000, 24) == pytest.approx(1000 * 0.03 / 24)
```

Program – Line coverage: 5/5 (100%)

```python
def calculate_interest(balance: float, months: int) -> float:
  rate = 0.02 # 2% per month
  if months > 12:
    rate = 0.03 # higher rate for long-term deposits
  return balance * rate / months
```

Test 1

```python
def test_short_term_interest():
  assert calculate_interest(1000, 6) == pytest.approx(3.33)
```

Test 2

```python
def test_long_term_interest():
  assert calculate_interest(1000, 24) == pytest.approx(1000 * 0.03 / 24)
```

Are we done covering everything?

Program – Line coverage: 5/5 (100%)

```
def calculate_interest(balance: float, months: int) -> float:
  rate = 0.02 # 2% per month
  if months > 12:
    rate = 0.03 # higher rate for long-term deposits
  return balance * rate / months
```

Test 1

```
def test_short_term_interest():
  assert calculate_interest(1000, 6) == pytest.approx(3.33)
```

Test 2

```
def test_long_term_interest():
  assert calculate_interest(1000, 24) == pytest.approx(1000 * 0.03 / 24)
```

Are we done covering everything?

Program – Line coverage: 5/5 (100%)

```
def calculate_interest(balance: float, months: int) -> float:
  rate = 0.02 # 2% per month
  if months > 12:
    rate = 0.03 # higher rate for long-term deposits
  return balance * rate / months
```

No!

Test 1

```
def test_short_term_interest():
  assert calculate_interest(1000, 6) == pytest.approx(3.33)
```

Test 2

```
def test_long_term_interest():
  assert calculate_interest(1000, 24) == pytest.approx(1000 * 0.03 / 24)
```

Are we done covering everything?

Program – Line coverage: 5/5 (100%)

```
def calculate_interest(balance: float, months: int) -> float:
  rate = 0.02 # 2% per month
  if months > 12:
    rate = 0.03 # higher rate for long-term deposits
  return balance * rate / months
```

No!

Test 1

```
def test_short_term_interest():
  assert calculate_interest(100
```

CWE-369: Divide By Zero

Weakness ID: 369
Vulnerability Mapping: ALLOWED
Abstraction: Base

Test 2

```
def test_long_term_interest():
  assert calculate_interest(1000, 24) == pytest.approx(1000 * 0.03 / 24)
```

Are we done covering everything?

Program – Line coverage: 5/5 (100%)

```
def calculate_interest(balance: float, months: int) -> float:
  rate = 0.02 # 2% per month
  if months > 12:
    rate = 0.03 # higher rate for long-term deposits
  return balance * rate / months
```

No!

Test 1

```
def test_short_term_interest():
  assert calculate_interest(1000, 6) == pytest.approx(3.33)
```

Test 2

```
def test_long_term_interest():
  assert calculate_interest(1000, 24) == pytest.approx(1000 * 0.03 / 24)
```

Test 3

```
def test_zero_months_divide_by_zero(): # covers implicit error path
  with pytest.raises(ZeroDivisionError):
    calculate_interest(1000, 0)
```

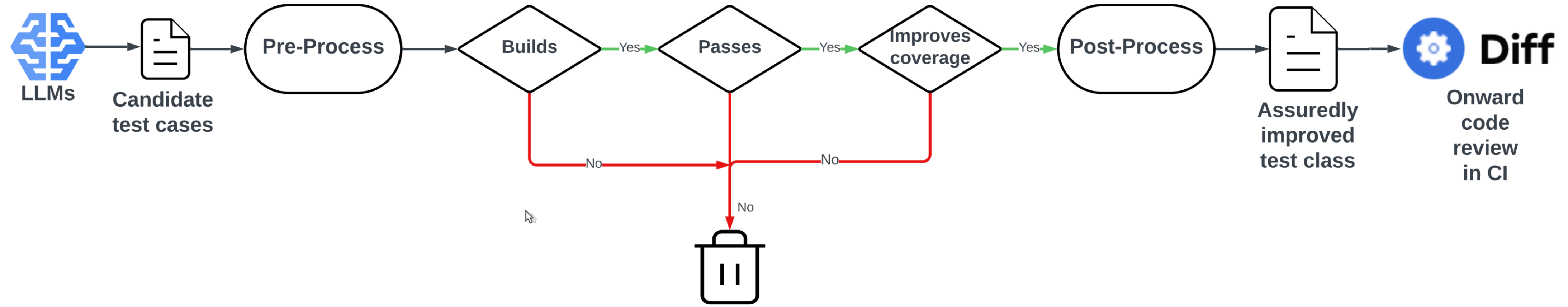
Automated Unit Test Improvement using Large Language Models at Meta

Nadia Alshahwan*
Jubin Chheda
Anastasia Finegenova
Beliz Gokkaya
Mark Harman
Inna Harper
Alexandru Marginean
Shubho Sengupta
Eddy Wang
Meta Platforms Inc.,
Menlo Park, California, USA

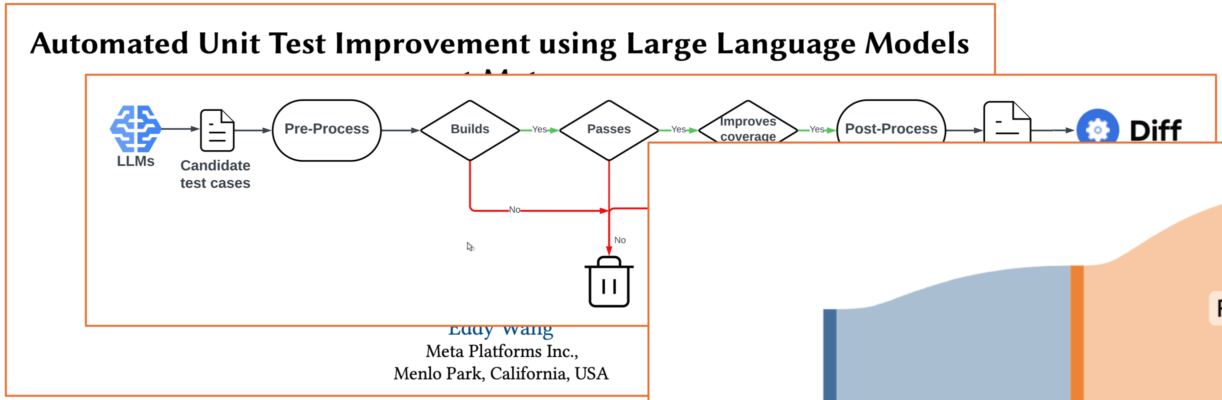
Automated Unit Test Improvement using Large Language Models at Meta, Alshahwan et. al., FSE 2024

Automated Unit Test Improvement using Large Language Models at Meta

Nadia Alshahwan*
Jubin Chheda
Anastasia Finegenova
Beliz Gokkaya
Mark Harman
Inna Harper
Alexandru Marginean
Shubho Sengupta
Eddy Wang



[Automated Unit Test Improvement using Large Language Models at Meta, Alshahwan et. al., FSE 2024](#)



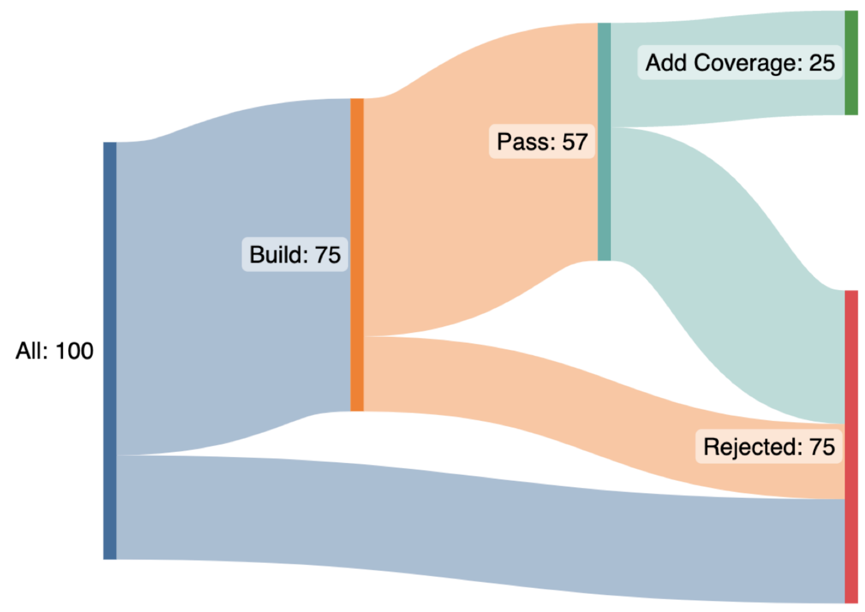
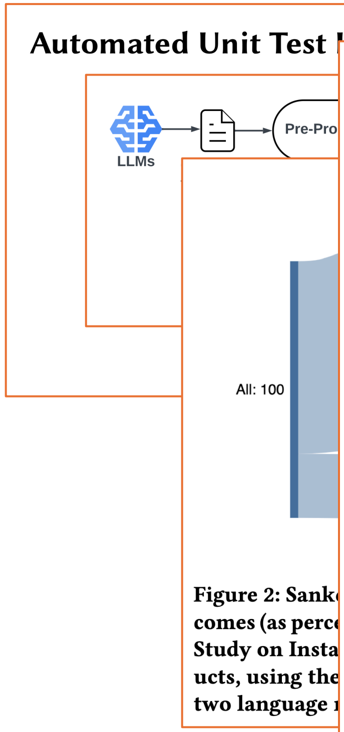


Figure 2: Sankey diagram showing the filtration process outcomes (as percentages of all test cases) from the Experimental Study on Instagram components for Reels and Stories products, using the four prompt strategies from Table 2 and the two language models, LLM1 and LLM2.

Automated Unit Test Improvement using Large Language Models at Meta, Alshahwan et. al., FSE 2024



3.4.1 Deployment at the Facebook App Test-a-thon December 2023.

In this deployment, we had sufficient confidence to automatically submit recommendations from TestGen-LLM to engineers. There was no engineer pre-training process, no specific test-a-thon expectations, and no additional context provided to the engineers. This gave us a realistic assessment of the engineers' response to LLM-generated test recommendations provided 'out of the box'. Overall, over 50% of the diffs submitted were accepted by developers, a figure which rises to almost 70% of those which received a review by developers. Specifically, of the 280 diffs generated by TestGen-LLM:

- 144 were accepted by the engineer reviewing them.
- 64 were rejected and/or abandoned.
- 61 did not receive a review.
- 11 were withdrawn.

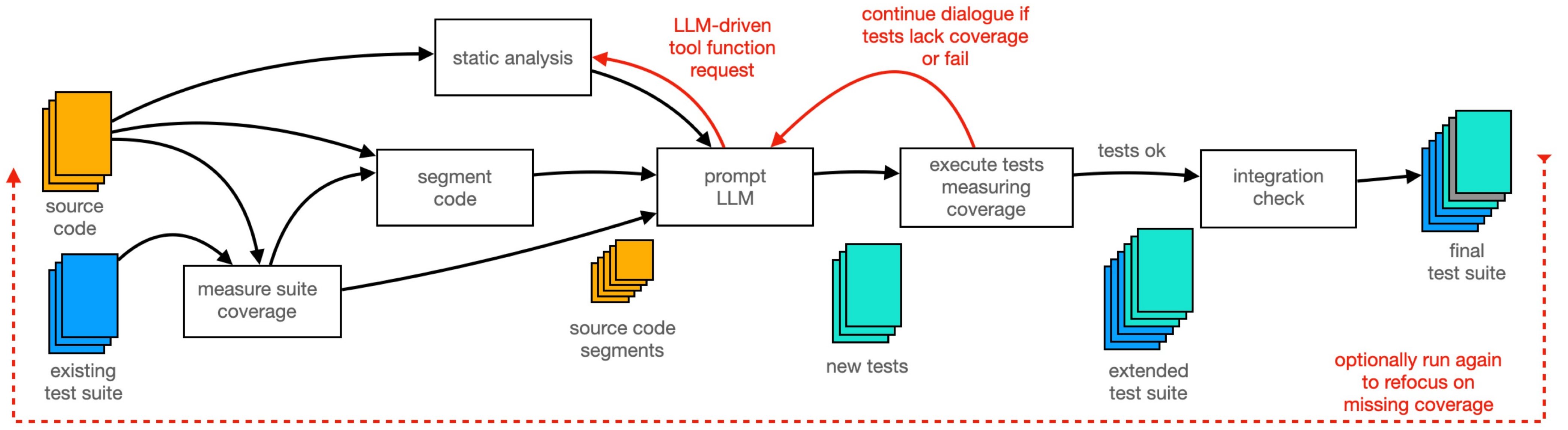
Automated Unit Test Improvement using Large Language Models at Meta, Alshahwan et. al., FSE 2024

CoverUp: Coverage-Guided LLM-Based Test Generation

JUAN ALTMAYER PIZZORNO, University of Massachusetts Amherst, United States
EMERY D. BERGER*, University of Massachusetts Amherst / Amazon, United States

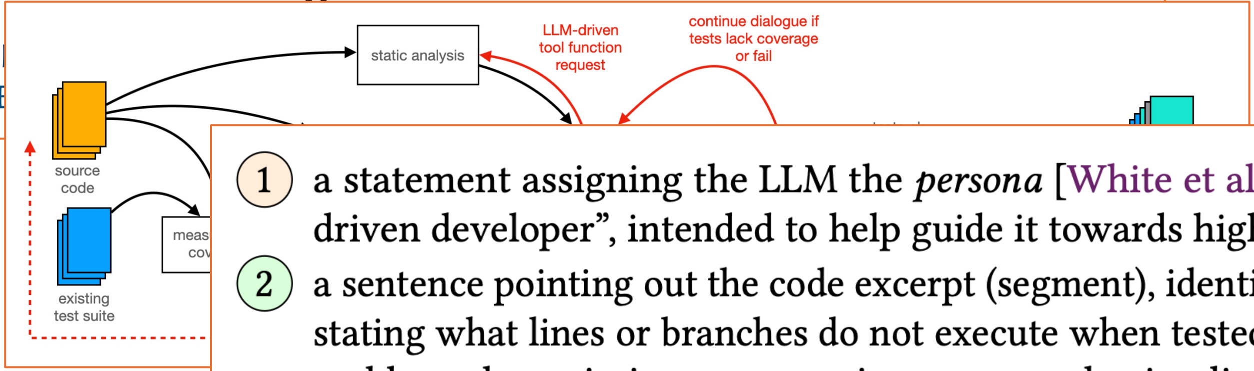
CoverUp: Coverage-Guided LLM-Based Test Generation

JUAN ALTMAYER PIZZORNO, University of Massachusetts Amherst, United States
EMERY D. BERGER*, University of Massachusetts Amherst / Amazon, United States



CoverUp: Coverage-Guided LLM-Based Test Generation

JUA
EME

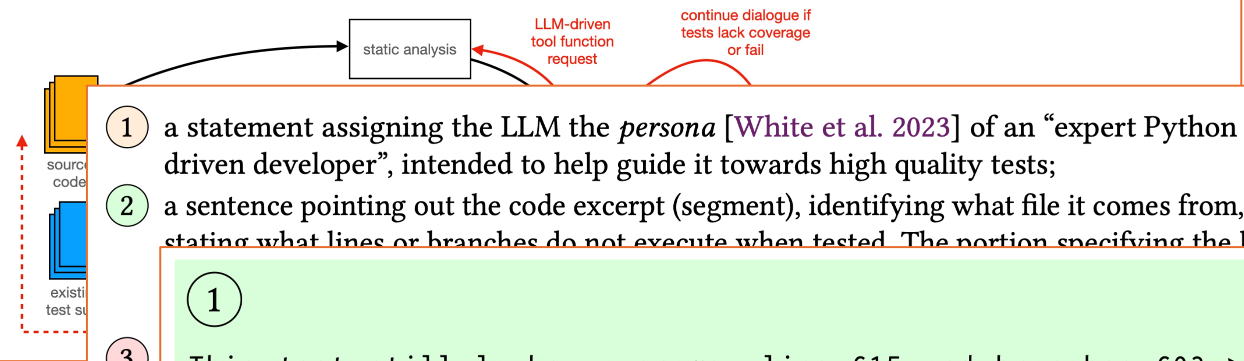


1. a statement assigning the LLM the *persona* [White et al. 2023] of an “expert Python test-driven developer”, intended to help guide it towards high quality tests;
2. a sentence pointing out the code excerpt (segment), identifying what file it comes from, and stating what lines or branches do not execute when tested. The portion specifying the lines and branches missing coverage is compressed using line ranges, simplifying the prompt and reducing token usage.
3. a request for `pytest` test functions and an encouragement for the LLM to use the provided tool function;
4. a series of other requests, such as “include assertions” and “avoid state pollution”, to steer the result towards usable tests;
5. a request that the response only include the new Python tests to facilitate its extraction from the response and to reduce token usage; and
6. the code segment, prefixed by generated `import` statements and tagging the lines lacking (line or branch) coverage with their numbers.

CoverUp: Coverage-Guided LLM-Based Test Generation

JUA

EME



1. a statement assigning the LLM the *persona* [White et al. 2023] of an "expert Python test-driven developer", intended to help guide it towards high quality tests;
2. a sentence pointing out the code excerpt (segment), identifying what file it comes from, and stating what lines or branches do not execute when tested. The portion specifying the lines
3.
4.
5.
6.

(1)

```
This test still lacks coverage: line 615 and branches 603->exit, 610->608, 618->exit
do not execute.
```

(2)

```
Modify it to correct that; respond only with the complete Python code in backticks.
Use the get_info function as necessary.
```

Fig. 6. **Example of a coverage follow-up prompt:** CoverUp indicates to the LLM that a line and some branches still weren't covered (1), asking that it correct the test (2).

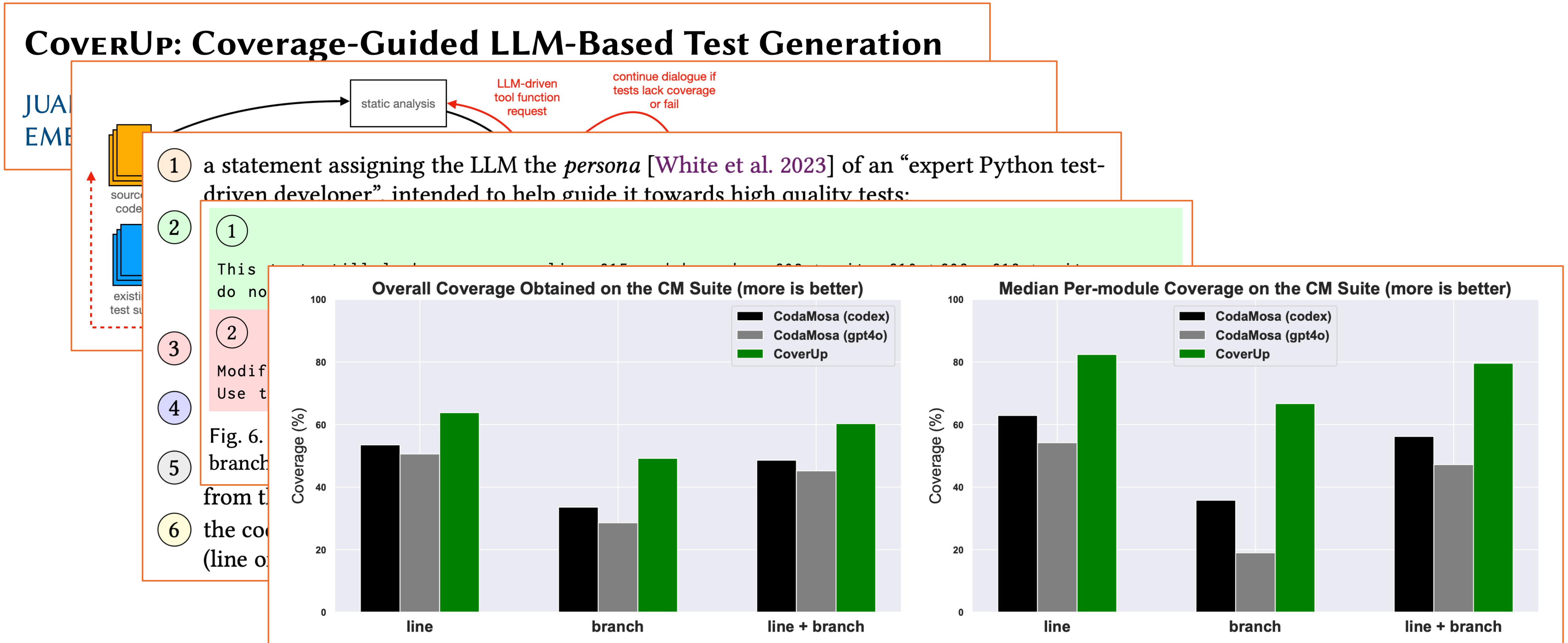


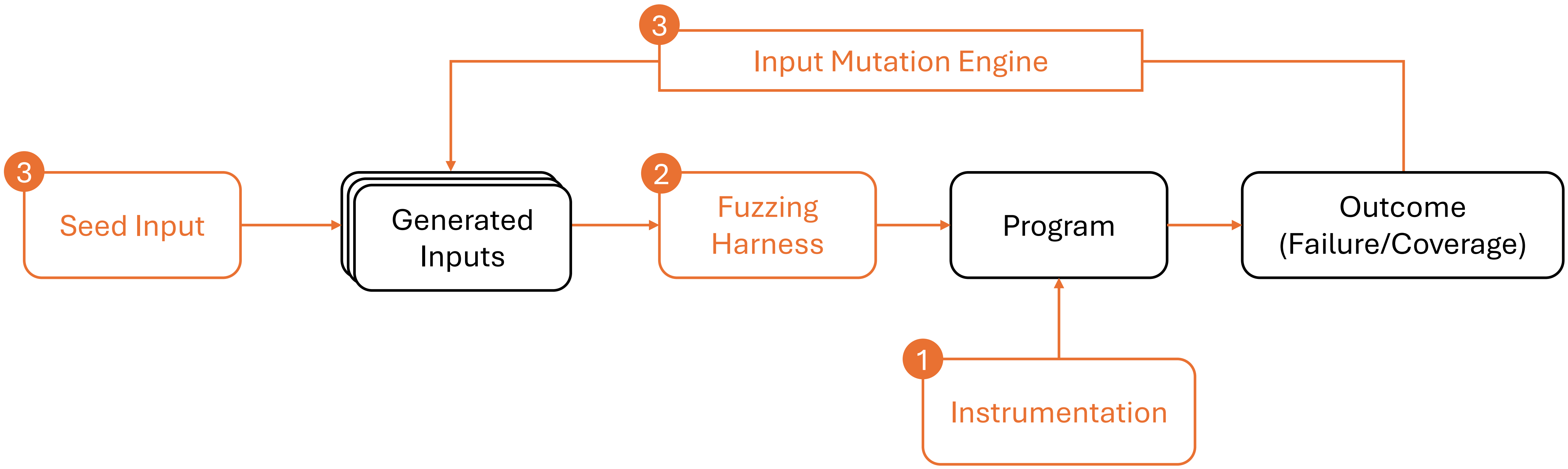
Fig. 12. **[RQ1] CoverUp yields higher overall and median per-module coverage:** Across the board, CoverUp yields higher coverage than CodaMosa, whether measured over the entire suite or on a module-by-module basis.

Fuzzing: Chaos with Purpose

```
for i in 0..10^4:
  x = gen_input()
  f(x)
```

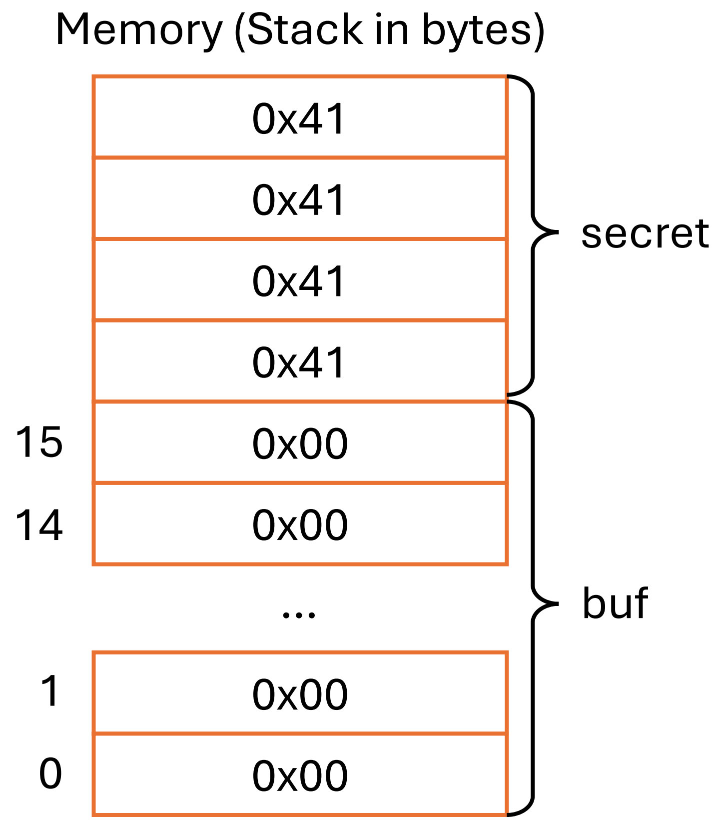
- Goal
  - Explore huge input spaces automatically to trigger failures (crashes, exceptions, timeouts, sanitizer hits)
  - Unlike unit tests, fuzzing does not know the correct output. It detects incorrect behavior by symptoms
- Method
  - Massively and randomly generate inputs to programs

Fuzzing: The Loop



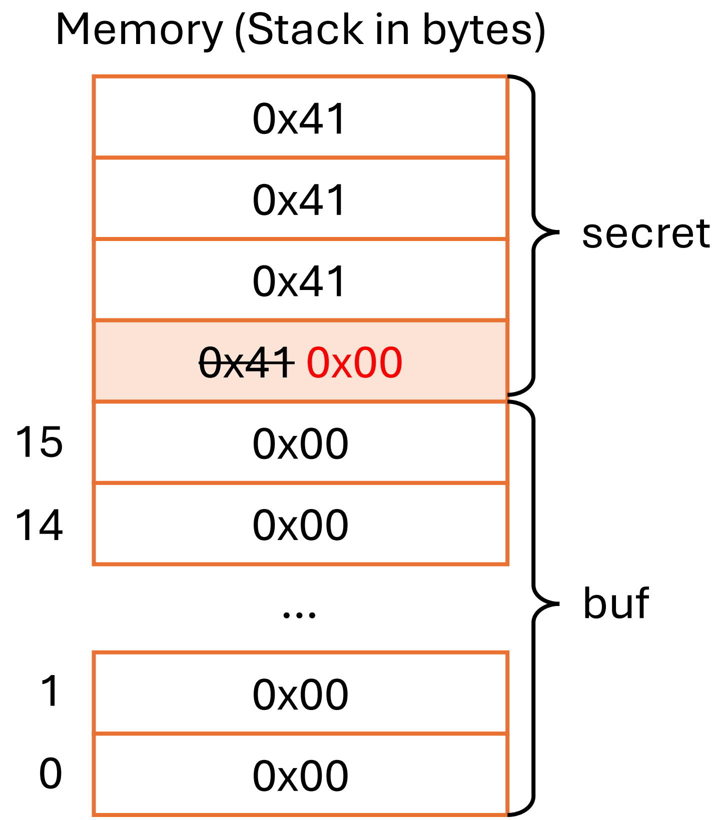
```c
int process_input(const uint8_t *data, size_t size) {
  if (size < 2) return 0; // need at least 2 bytes: cmd + length
  uint8_t cmd = data[0];
  uint8_t n = data[1]; // number of bytes to copy

  char buf[16]; // fixed-size buffer
  int secret = 0x41414141; // adjacent variable

  // If cmd == 0x42 we write to buffer
  if (cmd == 0x42) {
    // Write to buffer
    buf[n] = 0x00;
  }

  // Return secret so program behavior may subtly change if corrupted
  return secret;
}
```

```c
int process_input(const uint8_t *data, size_t size) {
  if (size < 2) return 0; // need at least 2 bytes: cmd + length
  uint8_t cmd = data[0];
  uint8_t n = data[1]; // number of bytes to copy

  char buf[16]; // fixed-size buffer
  int secret = 0x41414141; // adjacent variable

  // If cmd == 0x42 we write to buffer
  if (cmd == 0x42) {
    // Intentionally naive: no bounds check on n relative to buf size
    buf[n] = 0x00;
  }

  // Return secret so program behavior may subtly change if corrupted
  return secret;
}
```

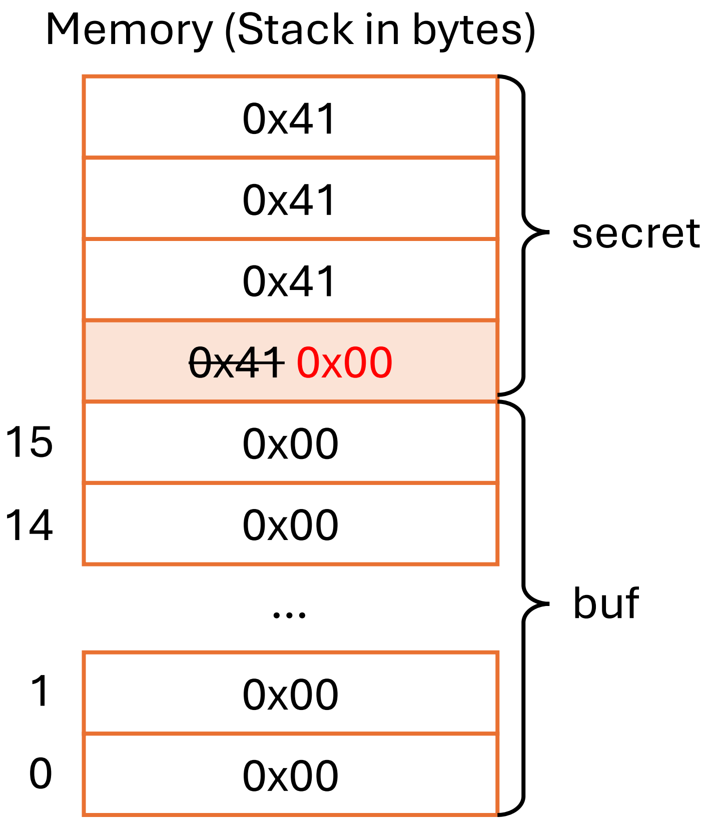


```c
int process_input(const uint8_t *data, size_t size) {
  if (size < 2) return 0; // need at least 2 bytes: cmd + length
  uint8_t cmd = data[0];
  uint8_t n = data[1]; // number of bytes to copy

  char buf[16]; // fixed-size buffer
  int secret = 0x41414141; // adjacent variable

  // If cmd == 0x42 we write to buffer
  if (cmd == 0x42) {
    // Intentionally naive: no bounds check on n relative to buf size
    buf[n] = 0x00;
  }

  // Return secret so program behavior may subtly change if corrupted
  return secret;
}
```

```c
process_input({0x42, 16}, 2)
```



```c
int process_input(const uint8_t *data, size_t size) {
  if (size < 2) return 0; // need at least 2 bytes: cmd + length
  uint8_t cmd = data[0];
  uint8_t n = data[1]; // number of bytes to copy

  char buf[16]; // fixed-size buffer
  int secret = 0x41414141; // adjacent variable

  // If cmd == 0x42 we write to buffer
  if (cmd == 0x42) {
    // Intentionally naive: no bounds check on n relative to buf size
    buf[n] = 0x00;
  }

  // Return secret so program behavior may subtly change if corrupted
  return secret;
}
```



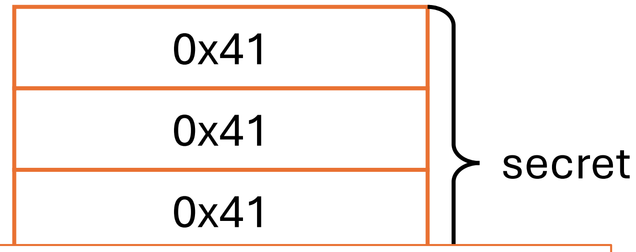

```
process_input({0x42, 16}, 2)
```

Issue:

- There is an out-of-bounds buffer write!
- But the program will not crash!
- How do we still detect the bug?

```c
int process_input(const uint8_t *data, size_t size) {
  if (size < 2) return 0; // need at least 2 bytes: cmd + length
  uint8_t cmd = data[0];
  uint8_t n = data[1]; // number of bytes to copy

  char buf[16]; // fixed-size buffer
```

Memory (Stack in bytes)



2024 CWE Top 25 Most Dangerous Software Weaknesses

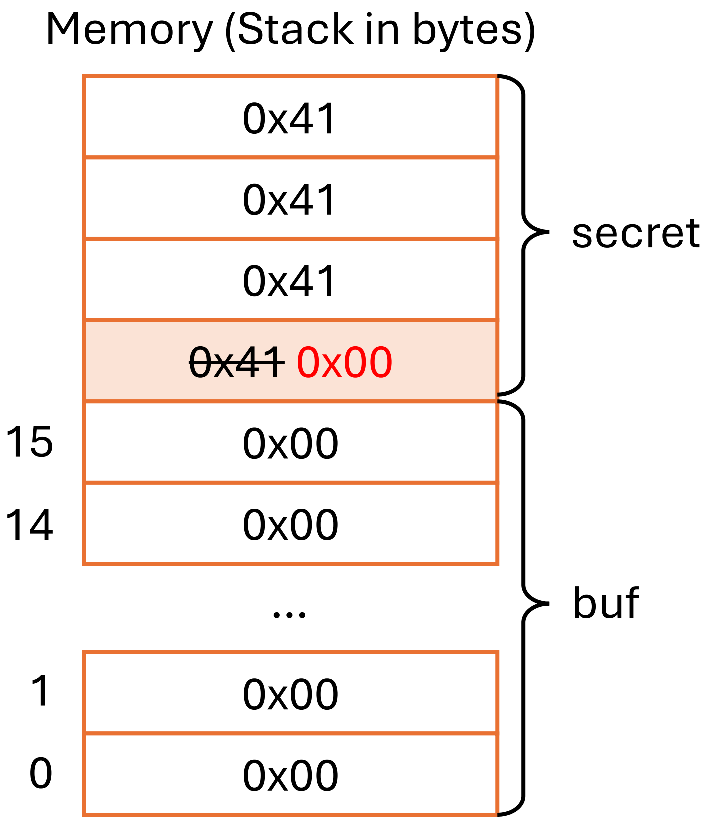
Top 25 Home | Share via: X | View in table format | Key Insights | Methodology

1 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CWE-79 | CVEs in KEV: 3 | Rank Last Year: 2 (up 1) ▲

2 Out-of-bounds Write
CWE-787 | CVEs in KEV: 18 | Rank Last Year: 1 (down 1) ▼

Issue:
- There is an out-of-bounds buffer write!
- But the program will not crash!
- How do we still detect the bug?

```c
int process_input(const uint8_t *data, size_t size) {
  if (size < 2) return 0; // need at least 2 bytes: cmd + length
  uint8_t cmd = data[0];
  uint8_t n = data[1]; // number of bytes to copy

  char buf[16]; // fixed-size buffer
  int secret = 0x41414141; // adjacent variable

  // If cmd == 0x42 we write to buffer
  if (cmd == 0x42) {
    // Intentionally naive: no bounds check on n relative to buf size
    buf[n] = 0x00;
  }

  // Return secret so program behavior may subtly change if corrupted
  return secret;
}
```




```
process_input({0x42, 16}, 2)
```

Issue:

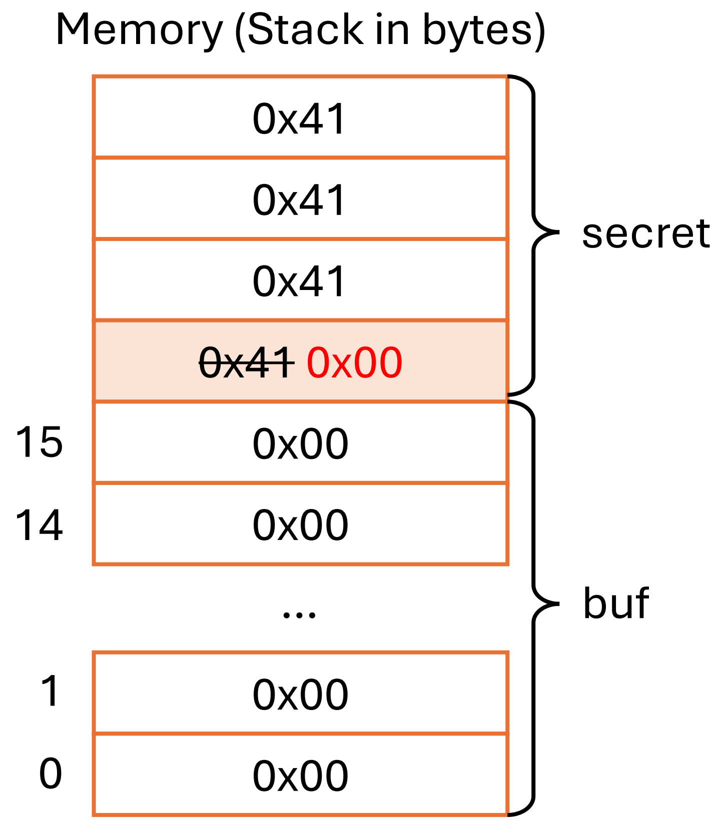
- There is an out-of-bounds buffer write!
- But the program will not crash!
- How do we still detect the bug?

Solution:

- Add Sanitizer!

```c
int process_input(const uint8_t *data, size_t size) {
  if (size < 2) return 0; // need at least 2 bytes: cmd + length
  uint8_t cmd = data[0];
  uint8_t n = data[1]; // number of bytes to copy

  char buf[16]; // fixed-size buffer
  int secret = 0x41414141; // adjacent variable

  // If cmd == 0x42 we write to buffer
  if (cmd == 0x42) {
    if (n >= 16) SANITIZER_ERROR("out-of-bounds write")
    buf[n] = 0x00;
  }

  // Return secret so program behavior may subtly change if corrupted
  return secret;
}
```

```c
process_input({0x42, 16}, 2)
```



Issue:

- There is an out-of-bounds buffer write!
- But the program will not crash!
- How do we still detect the bug?

Solution:

- Add Sanitizer!

Program Instrumentation

P

Original Program

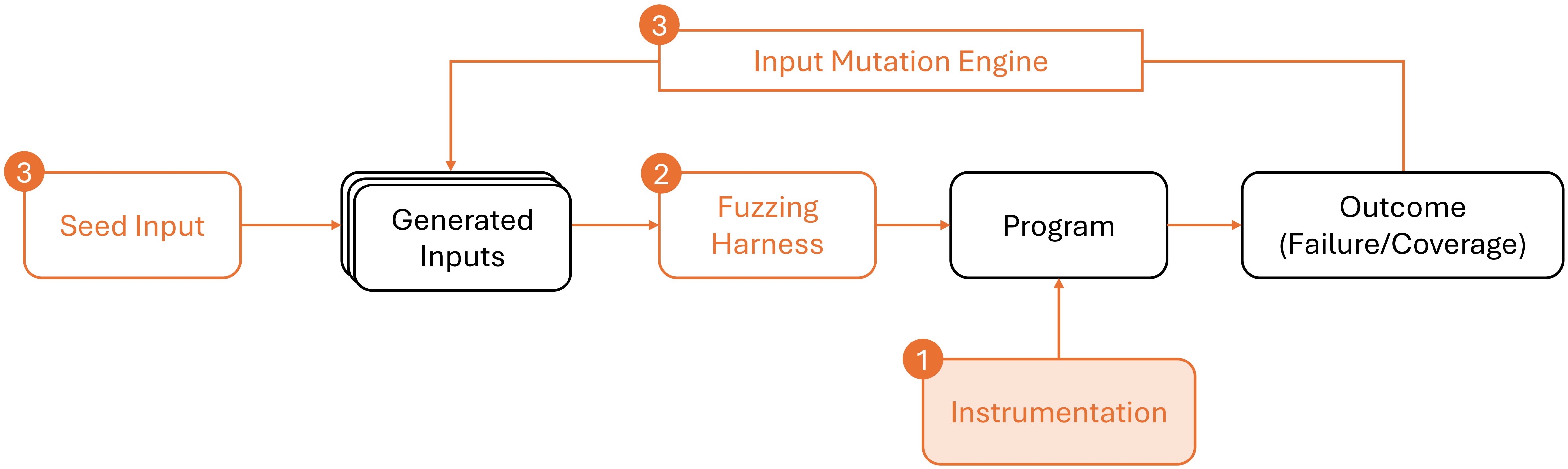
P'

Instrumented Program

P and P' are equivalent under normal behavior

P' can expose more failure modes than P

Fuzzing: The Loop



Fuzzing Harness

- For unit tests:
  - We come up with a single input-output pair
  - We write a test case that runs the program on input, compares the output
- For fuzzing:
  - We have automatically generated input
  - We write a test case that runs the program on input // ← HARNESS

Fuzzing Harness

- For unit tests:
  - We come up with a single input-output pair
  - We write a test case that runs the program on input, compares the output
- For fuzzing:
  - We have automatically generated input
  - We write a test case that runs the program on input // ← HARNESS

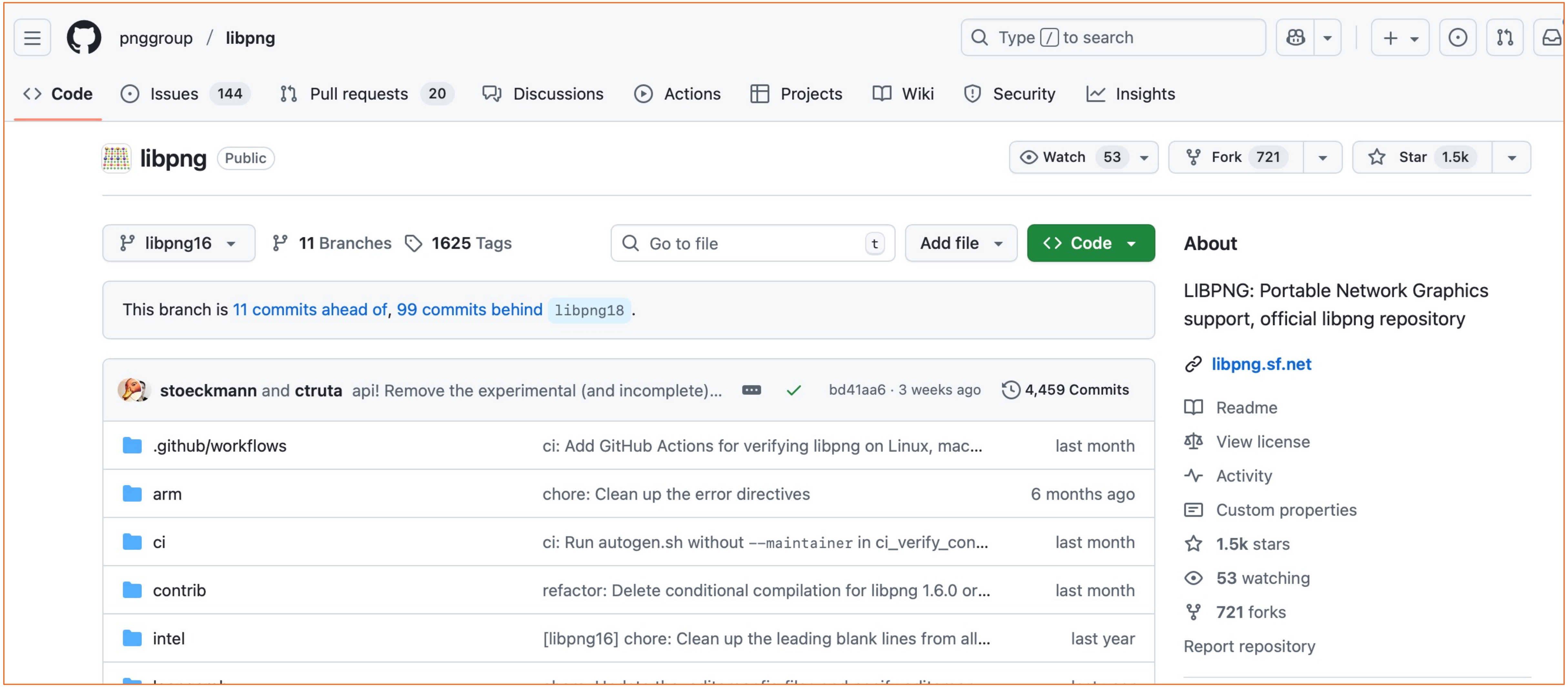
We have to do this for a lot of times!

Fuzzing Harness

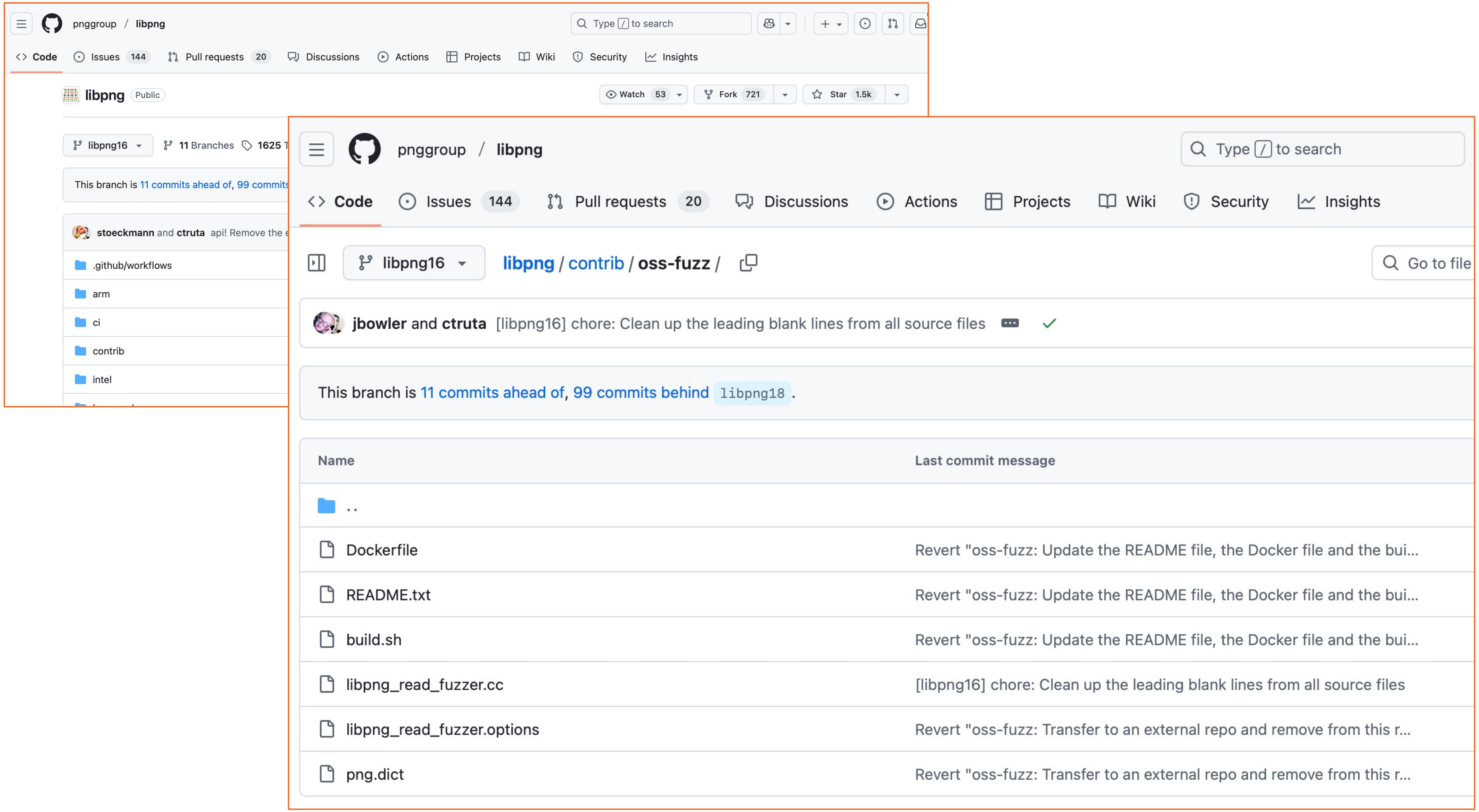
- For unit tests:
  - We come up with a single input-output pair
  - We write a test case that runs the program on input, compares the output
- For fuzzing:
  - We have automatically generated input
  - We write a test case that runs the program on input // ← **HARNESS**

We have to do this for a lot of times!

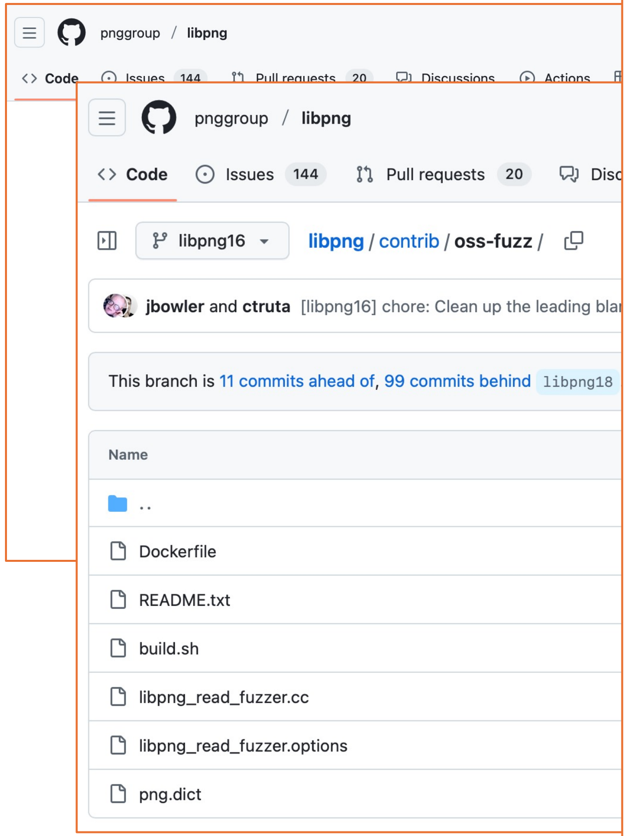
Can we just do this one time or only a few times?



https://github.com/pnggroup/libpng



https://github.com/pnggroup/libpng



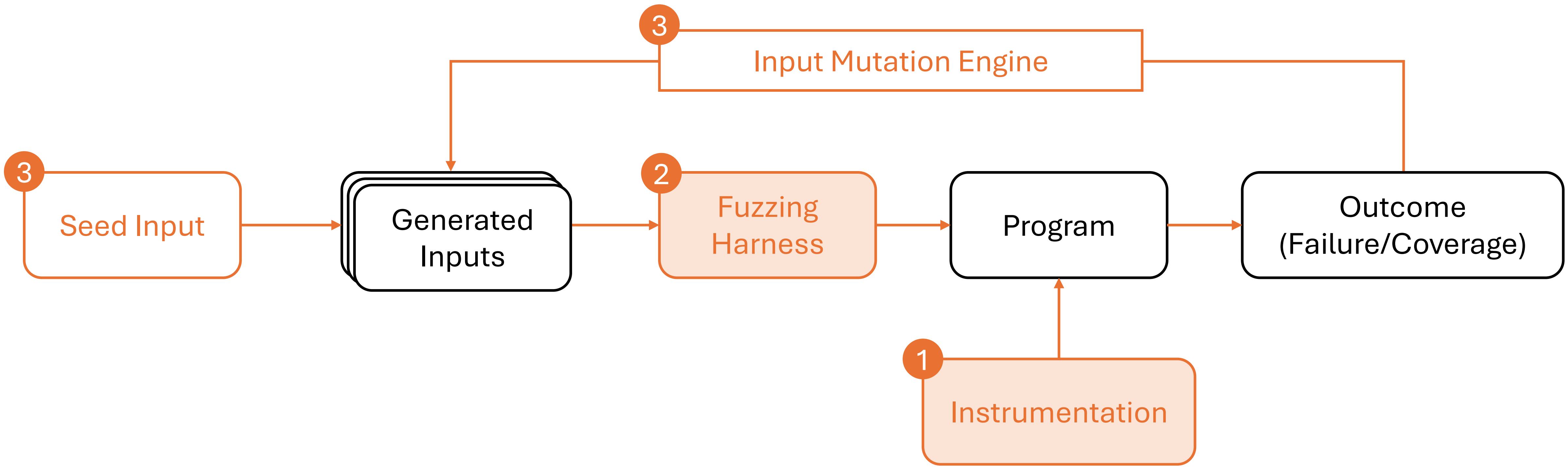

```
// Entry point for LibFuzzer.
// Roughly follows the libpng book example:
// http://www.libpng.org/pub/png/book/chapter13.html
extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
  if (size < kPngHeaderSize) {
    return 0;
  }

  std::vector<unsigned char> v(data, data + size);
  if (png_sig_cmp(v.data(), 0, kPngHeaderSize)) {
    // not a PNG.
    return 0;
  }

  PngObjectHandler png_handler;
  png_handler.png_ptr = nullptr;
  png_handler.row_ptr = nullptr;
  png_handler.info_ptr = nullptr;
  png_handler.end_info_ptr = nullptr;

  png_handler.png_ptr = png_create_read_struct
    (PNG_LIBPNG_VER_STRING, nullptr, nullptr, nullptr);
  if (!png_handler.png_ptr) {
    return 0;
  }

  png_handler.info_ptr = png_create_info_struct(png_handler.png_ptr);
  if (!png_handler.info_ptr) {
    PNG_CLEANUP
    return 0;
  }

  png_handler.end_info_ptr = png_create_info_struct(png_handler.png_ptr);
  if (!png_handler.end_info_ptr) {
    PNG_CLEANUP
    return 0;
  }
```

https://github.com/pnggroup/libpng/blob/libpng16/contrib/oss-fuzz/libpng_read_fuzzer.cc

Fuzzing Harness: Key Message

- A single wide-capturing test case that takes a single buffer as input
  - Reject bad inputs that we do not care about in the beginning
  - Simulate complete usage of a library or a package
  - Trigger sanitization errors or capture failures and exceptions
- In practice, a library may be accompanied multiple harnesses
  - But not to the level of the amount of unit test cases

Fuzzing: The Loop



Seed Input and Mutation Engine

- Seed Input
  - Typically some valid inputs to a program
  - E.g., an HTML for a browser, a PNG for libpng, a JSON for a JSON parser
- Mutation Engine
  - With a single input, we may get feedback from the execution:
    - E.g., coverage, error, command line outputs, etc.
  - We may use the coverage to guide the mutation engine to change the seed input to something else
    - E.g., JSON "`[1, 2, 3]`" → "`[1, 2,]`"
  - Hopefully, the mutation leads us to a better coverage

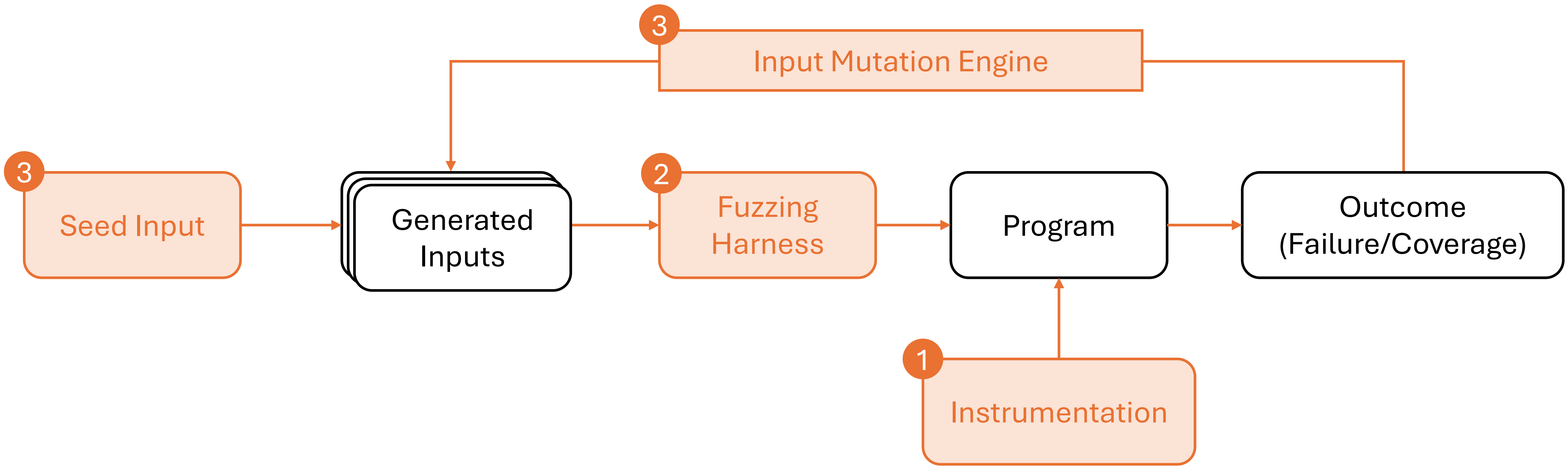
Infinite Monkey Theorem

"a monkey hitting keys independently and at random on a typewriter keyboard for an infinite amount of time will almost surely type any given text, including the complete works of William Shakespeare."

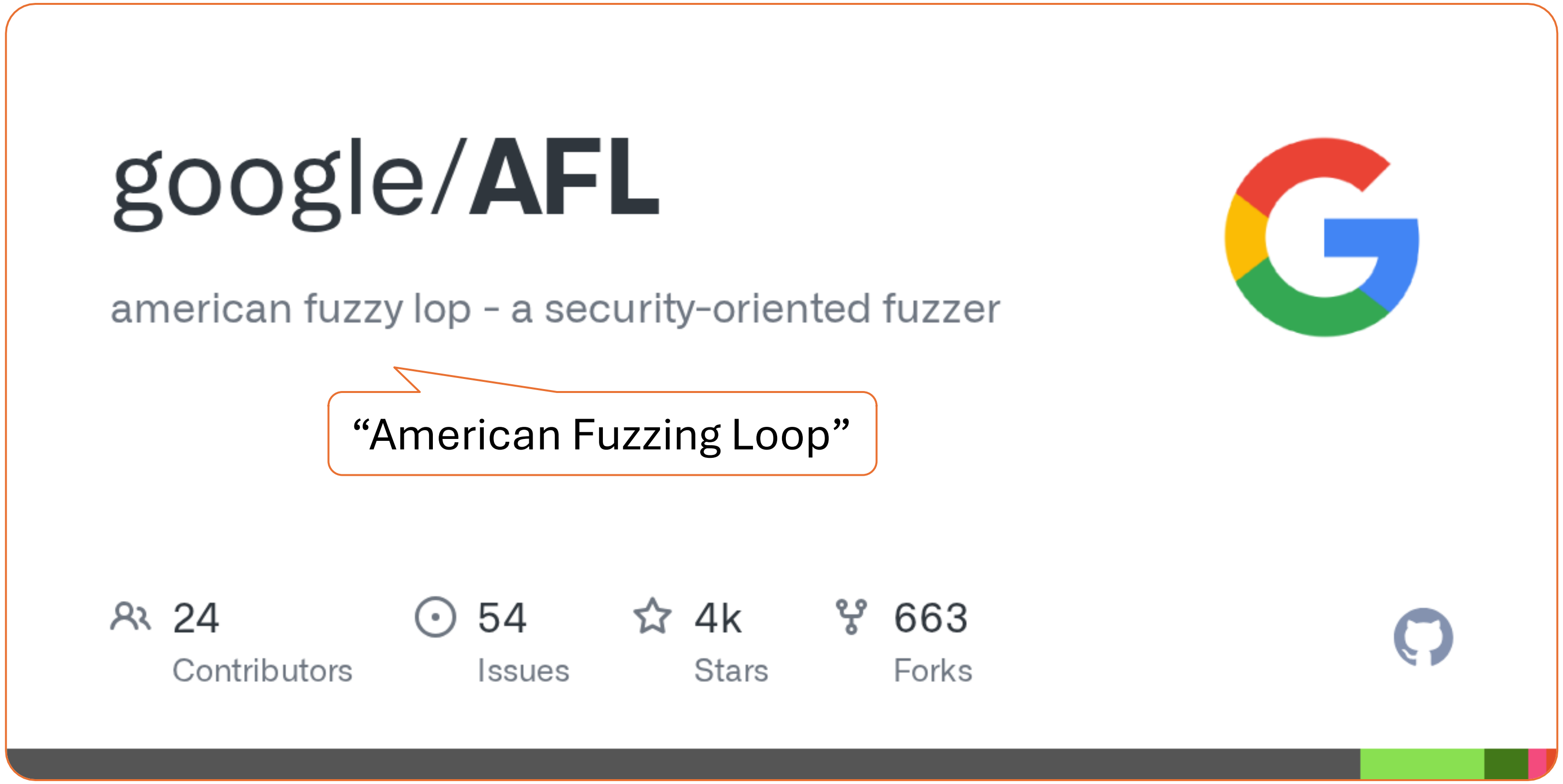
Seed Input and Mutation Engine

- Seed Input
  - Typically some valid inputs to a program
  - E.g., an HTML for a browser, a PNG for libpng, a JSON for a JSON parser
- Mutation Engine
  - With a single input, we may get feedback from the execution:
    - E.g., coverage, error, command line outputs, etc.
  - We may use the coverage to guide the mutation engine to change the seed input to something else
    - E.g., JSON "`[1, 2, 3]`" → "`[1, 2,]`"
  - Hopefully, the mutation leads us to a better coverage

Fuzzing: The Loop



Fuzzing in the Wild: AFL



“American Fuzzing Loop”

AFL++: Combining Incremental Steps of Fuzzing Research

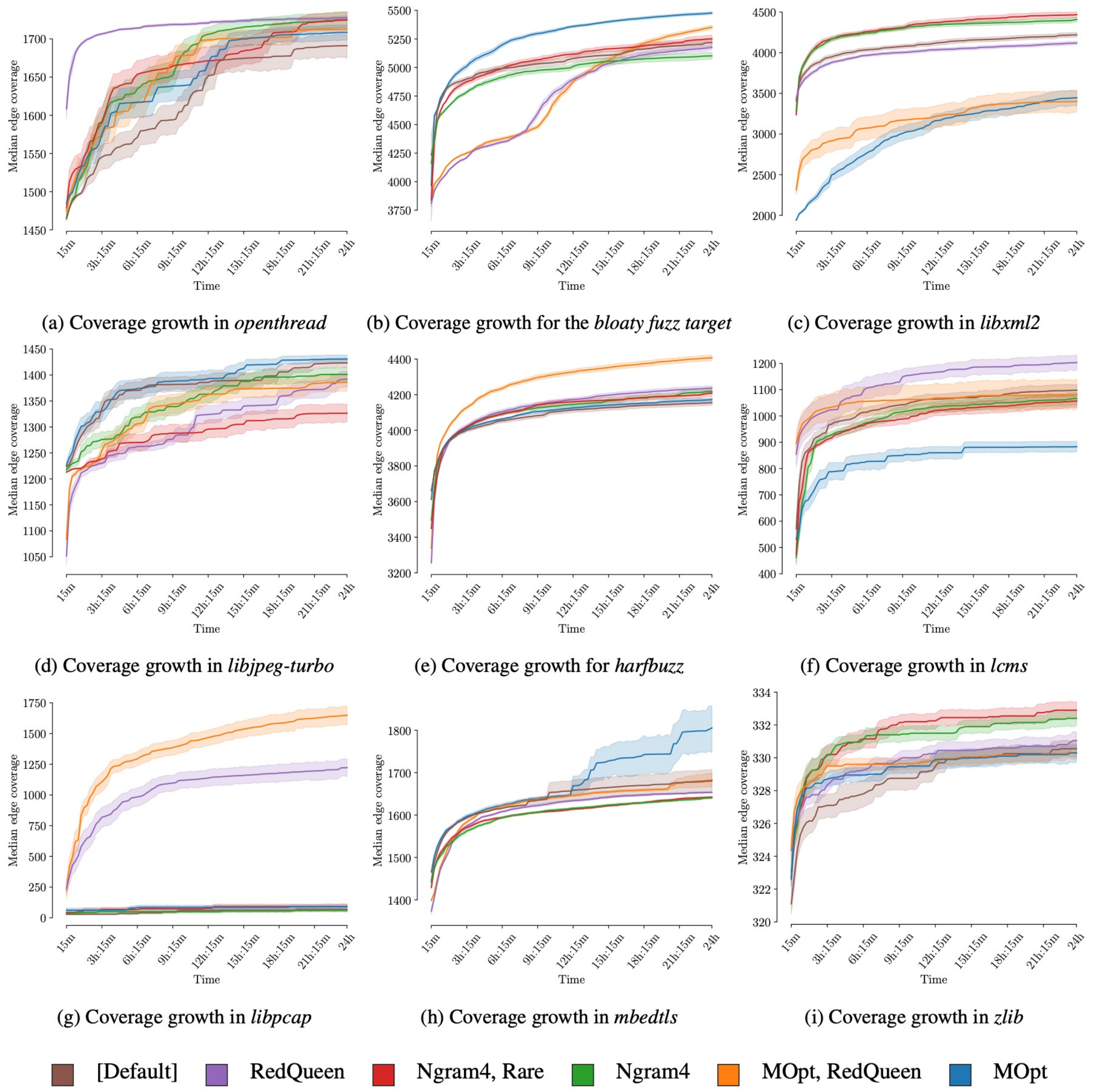
Andrea Fioraldi[†], Dominik Maier[‡], Heiko Eißfeldt, Marc Heuse[§]
{andrea, dominik, heiko, marc}@aflplus.plus
†Sapienza University of Rome, ‡TU Berlin, §The Hacker's Choice

AFL++: Combining Inc

Andrea Fioraldi†, Domin

{andrea, domi

†Sapienza University o



(a) Coverage growth in *openthread*

(b) Coverage growth for the *bloaty fuzz target*

(c) Coverage growth in *libxml2*

(d) Coverage growth in *libjpeg-turbo*

(e) Coverage growth for *harfbuzz*

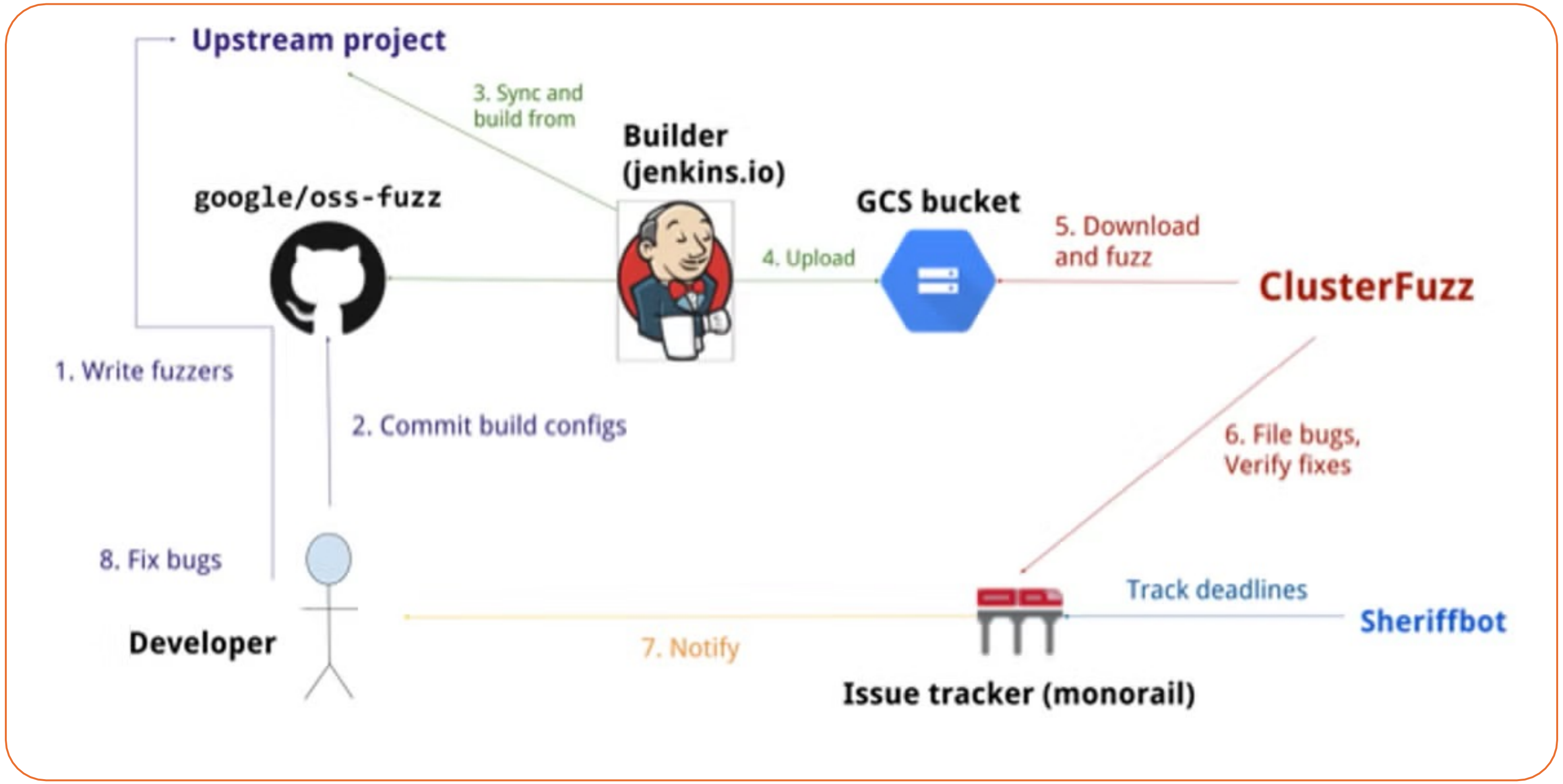
(f) Coverage growth in *lcms*

(g) Coverage growth in *libpcap*

(h) Coverage growth in *mbedtls*

(i) Coverage growth in *zlib*

Fuzzing in the Wild: OSS-Fuzz



Fuzzing in the Wild: OSS-Fuzz

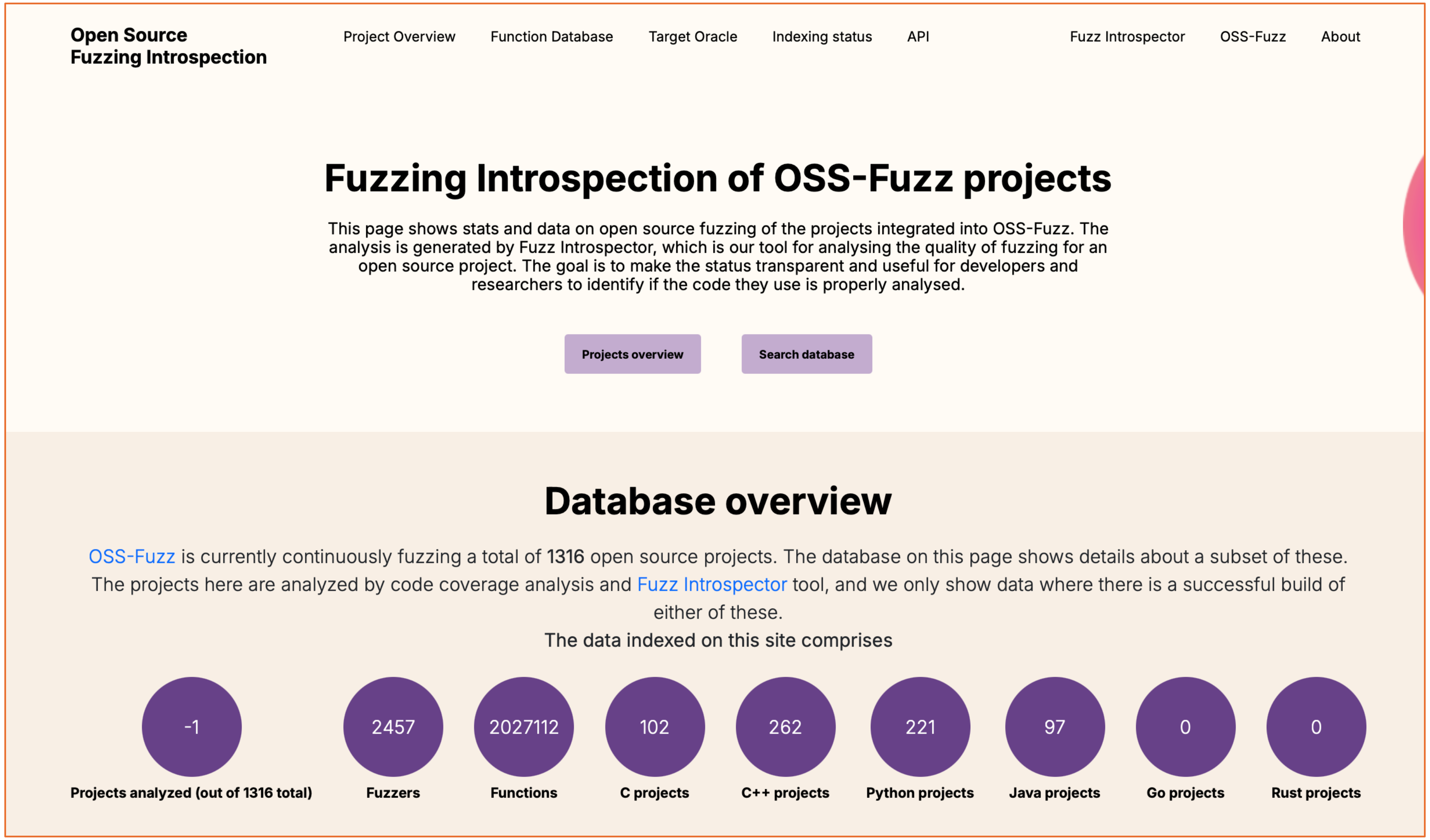
Coverage Report

View results by: Directories | Files

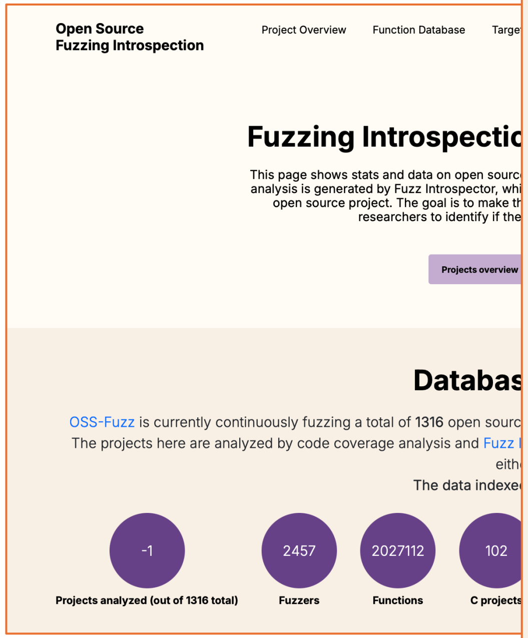
PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
autofit/	89.65% (6801/7586)	88.64% (78/88)	86.09% (3280/3810)
base/	73.47% (7650/10413)	77.81% (249/320)	62.89% (5010/7966)
bdf/	94.60% (2503/2646)	97.30% (36/37)	92.54% (2133/2305)
cff/	91.68% (5048/5506)	92.08% (93/101)	89.75% (2880/3209)
cid/	96.03% (1717/1788)	93.94% (31/33)	94.75% (1100/1161)
gzip/	95.54% (1435/1502)	94.12% (32/34)	88.73% (1299/1464)
lzw/	79.80% (482/604)	83.33% (15/18)	76.90% (263/342)
pcf/	94.46% (1722/1823)	83.87% (26/31)	93.17% (1445/1551)
pfr/	91.69% (2603/2839)	91.49% (43/47)	92.17% (2049/2223)
psaux/	91.19% (11504/12615)	96.12% (198/206)	88.69% (5990/6754)
pshinter/	95.20% (2957/3106)	92.41% (73/79)	91.25% (1356/1486)
psnames/	99.36% (466/469)	100.00% (11/11)	98.70% (227/230)
raster/	96.07% (2002/2084)	90.48% (38/42)	91.34% (1539/1685)
sfnt/	89.03% (8089/9086)	96.00% (144/150)	87.35% (6187/7083)
smooth/	90.62% (899/992)	82.14% (23/28)	85.52% (685/801)
truetype/	95.29% (12299/12907)	99.63% (269/270)	89.28% (7409/8299)
type1/	93.10% (4154/4462)	93.94% (62/66)	90.94% (2811/3091)
type42/	95.79% (1662/1735)	90.62% (29/32)	94.50% (1099/1163)
winfonts/	96.16% (826/859)	100.00% (13/13)	94.33% (632/670)
TOTALS	90.12% (74819/83022)	91.10% (1463/1606)	85.71% (47394/55293)

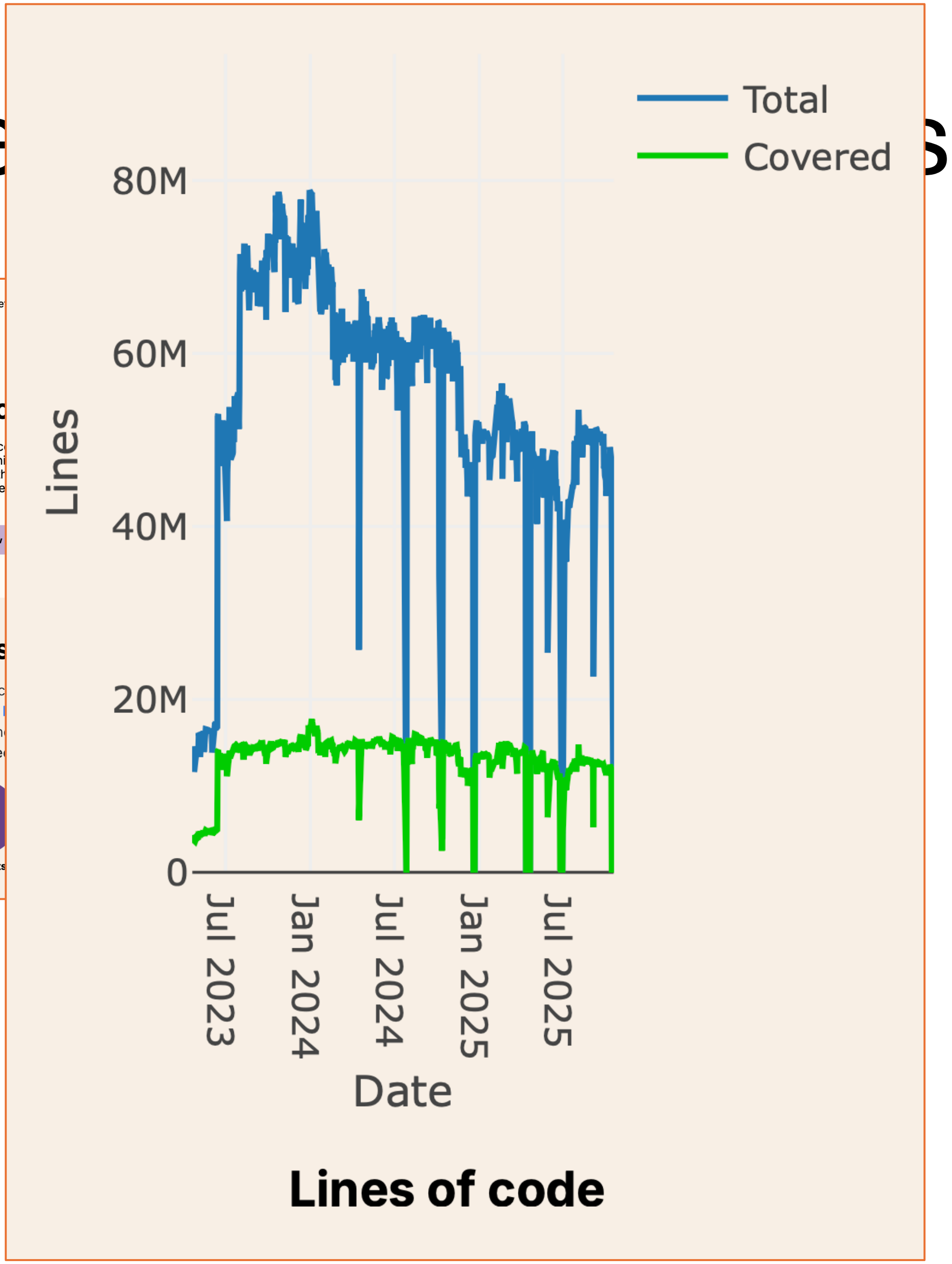
```
           /* multiply a given value by the CORDIC shrink factor */
           static FT_Fixed
           ft_trig_downscale( FT_Fixed  val )
   111k    {
   111k      FT_Int  s = 1;
   111k
   111k
   111k      if ( val < 0 )
      0      {
      0        val = -val;
      0        s = -1;
      0      }
   111k
   111k      /* 0x40000000 comes from regression analysis between true */
   111k      /* and CORDIC hypotenuse, so it minimizes the error       */
   111k      val = (FT_Fixed)(
                     ( (FT_UInt64)val * FT_TRIG_SCALE + 0x40000000UL ) >> 32 );
   111k
   111k      return s < 0 ? -val : val;
   111k    }
```

Fuzzing in the Wild: Fuzzing Introspection

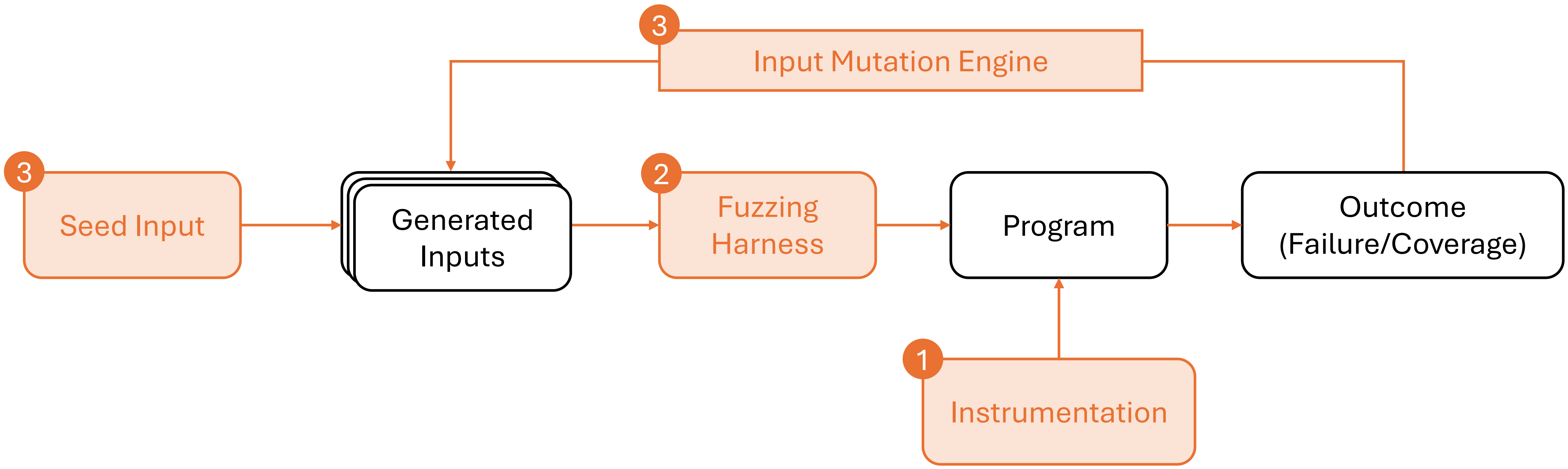


Fuzzing in the [...]spection





How Can LLM Help?



How Can LLM Help?

- LLM for instrumentation:
  - Capture more vulnerability patterns than traditional sanitizers
  - E.g., Path Traversal, Reading from external logs, etc.
- LLM for fuzzing harness generation:
  - Capture more library usage pattern
  - E.g., For libpng, simulate browser usage, PDF reader usage, etc.
- LLM for seed input generation & mutation:
  - Similar to unit test generation, but can accept more types of feedback
  - E.g., Looking at uncovered lines, mutate input towards those lines

How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation

Cen Zhang
Nanyang Technological University
Singapore

Yaowen Zheng*
Nanyang Technological University
Singapore

Mingqiang Bai
IIE, CAS; Sch of Cyber Security, UCAS
Beijing, China

Yeting Li
IIE, CAS; Sch of Cyber Security, UCAS
Beijing, China

Wei Ma
Nanyang Technological University
Singapore

Xiaofei Xie
Singapore Management University
Singapore

Yuekang Li
The University of New South Wales
Sydney, Australia

Limin Sun
IIE, CAS; Sch of Cyber Security, UCAS
Beijing, China

Yang Liu
Nanyang Technological University
Singapore

How Effective Are They? Exploring Large Language Model Based

Cen Z
Nanyang Technol
Singa
Yetin
IIE, CAS; Sch of Cyb
Beijing,
Yuekai
The University of N
Sydney, A

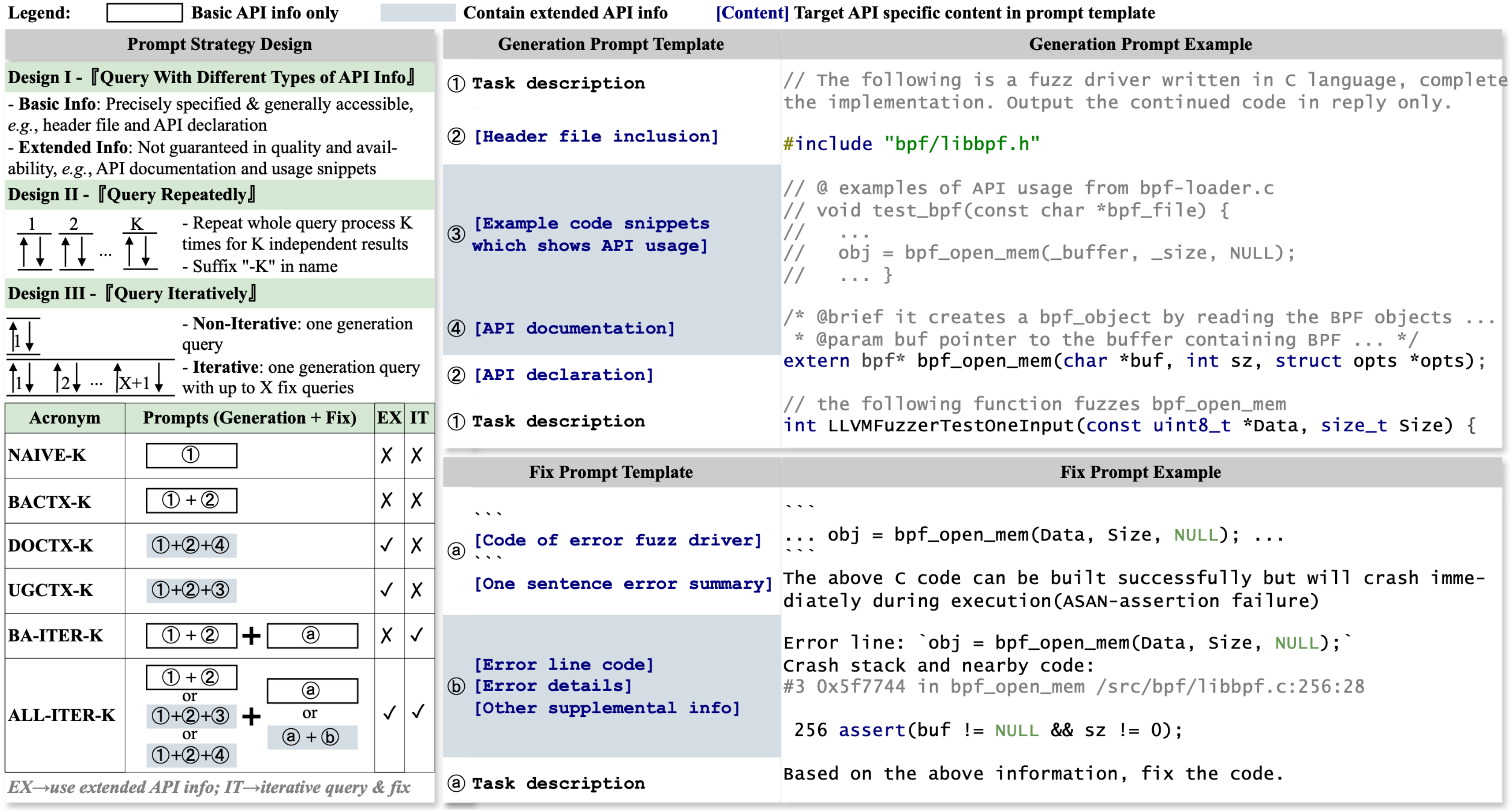
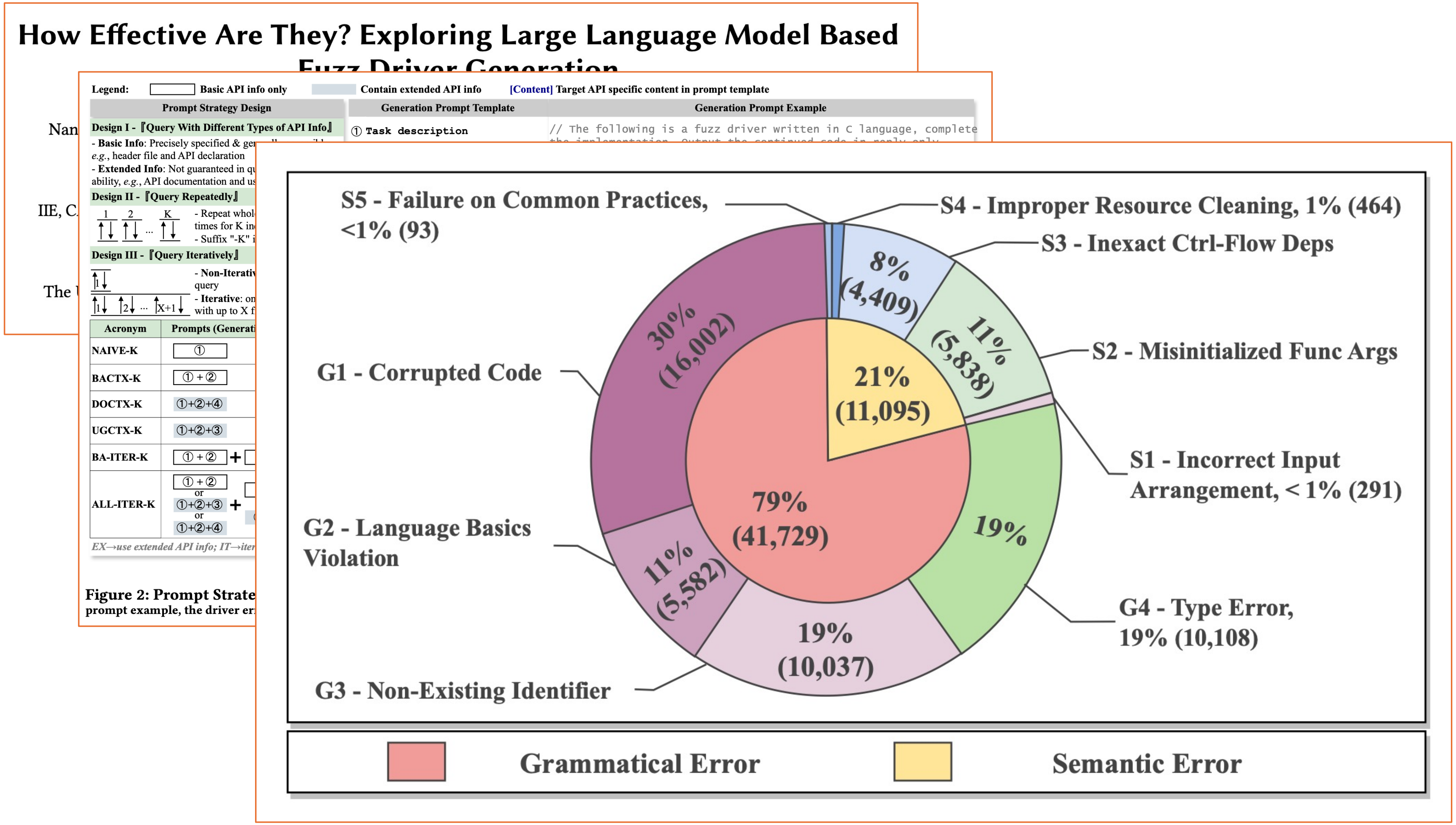
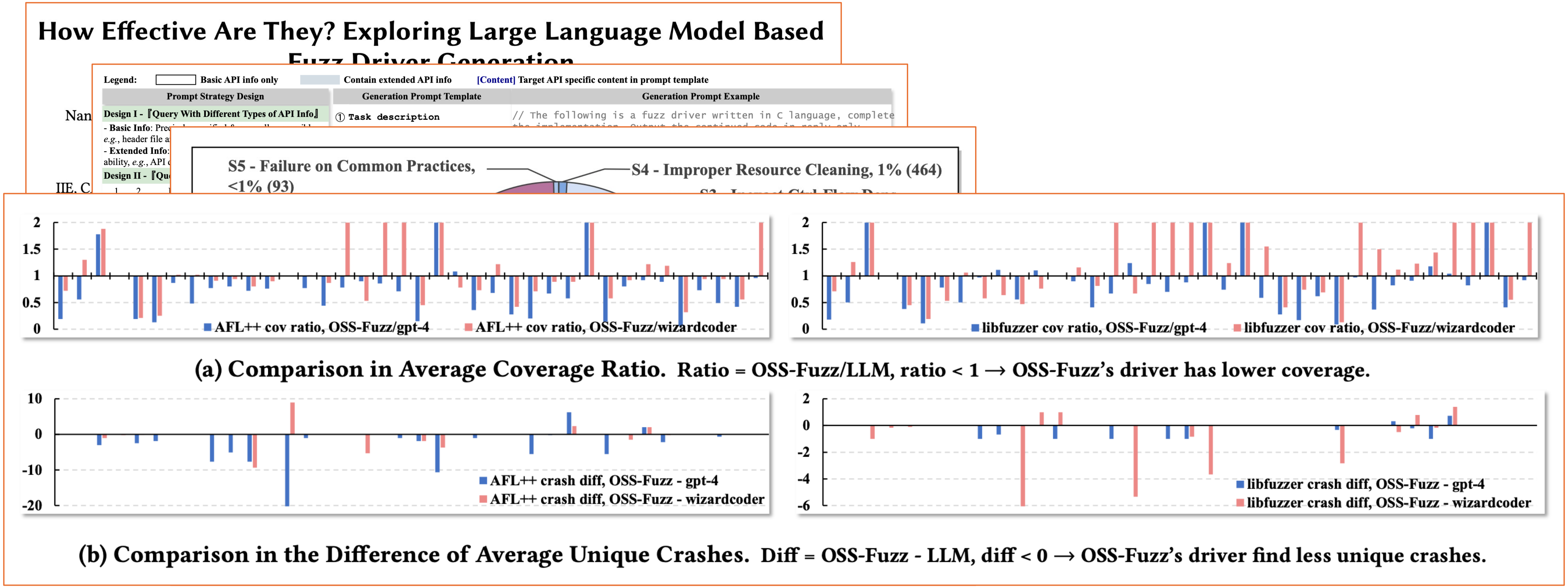


Figure 2: Prompt Strategies Overview. K is 1 or 40 in our evaluation and X is 5. The examples are simplified for demonstration purpose. In the fix prompt example, the driver error is caused by missing check of Size > 0 before calling the API, and the nearby code of #3 stack frame hints the error.


How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation
Legend: Basic API info only
Contain extended API info
[Content] Target API specific content in prompt template
Prompt Strategy Design
Generation Prompt Template
Generation Prompt Example
Design I - 『Query With Different Types of API Info』
Design II - 『Query Repeatedly』
Design III - 『Query Iteratively』
① Task description
// The following is a fuzz driver written in C language, complete
Acronym
NAIVE-K
BACTX-K
DOCTX-K
UGCTX-K
BA-ITER-K
ALL-ITER-K
Figure 2: Prompt Strate
S5 - Failure on Common Practices, <1% (93)
S4 - Improper Resource Cleaning, 1% (464)
S3 - Inexact Ctrl-Flow Deps
S2 - Misinitialized Func Args
S1 - Incorrect Input Arrangement, < 1% (291)
G1 - Corrupted Code
G2 - Language Basics Violation
G3 - Non-Existing Identifier
G4 - Type Error, 19% (10,108)
30% (16,002)
11% (5,582)
19% (10,037)
19%
8% (4,409)
11% (5,838)
21% (11,095)
79% (41,729)
Grammatical Error
Semantic Error



(a) Comparison in Average Coverage Ratio. Ratio = OSS-Fuzz/LLM, ratio < 1 → OSS-Fuzz's driver has lower coverage.

(b) Comparison in the Difference of Average Unique Crashes. Diff = OSS-Fuzz - LLM, diff < 0 → OSS-Fuzz's driver find less unique crashes.

Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models

Yinlin Deng
University of Illinois
Urbana-Champaign, USA
yinlind2@illinois.edu

Chunqiu Steven Xia
University of Illinois
Urbana-Champaign, USA
chunqiu2@illinois.edu

Haoran Peng
University of Science and
Technology of China, China
hurrypeng@mail.ustc.edu.cn

Chenyuan Yang
University of Illinois
Urbana-Champaign, USA
cy54@illinois.edu

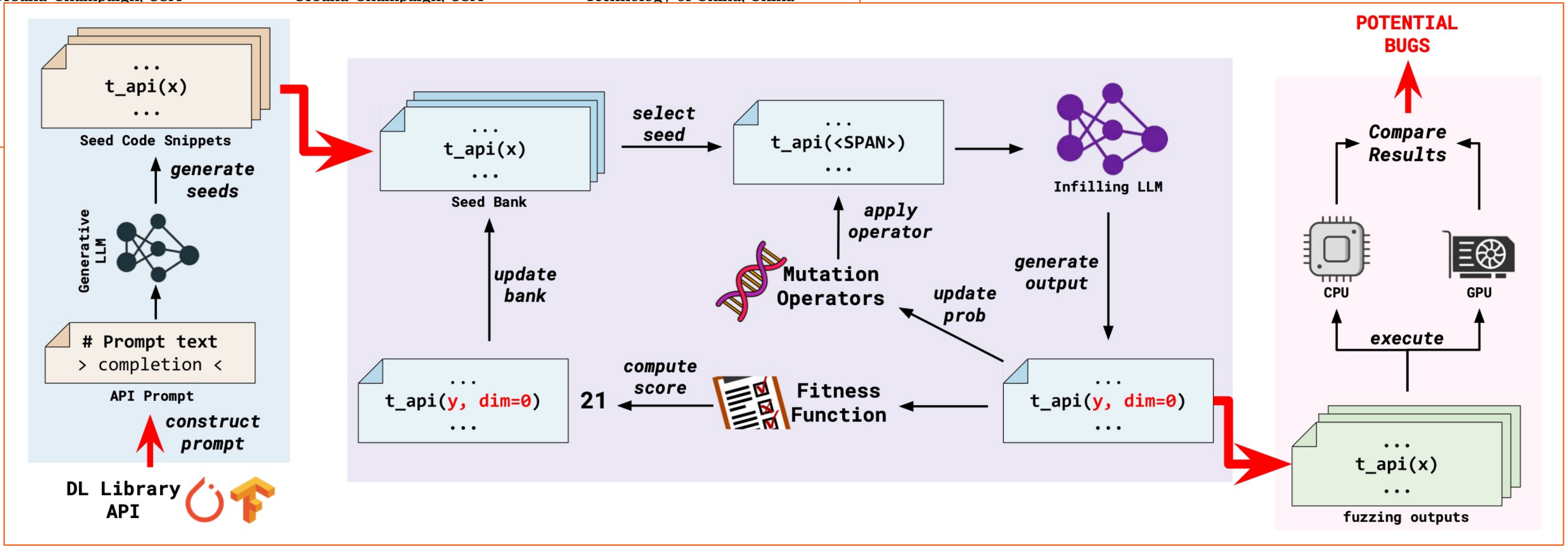
Lingming Zhang
University of Illinois
Urbana-Champaign, USA
lingming@illinois.edu

Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models

Yinlin Deng
University of Illinois
Urbana-Champaign, USA

Chunqiu Steven Xia
University of Illinois
Urbana-Champaign, USA

Haoran Peng
University of Science and
Technology of China, China



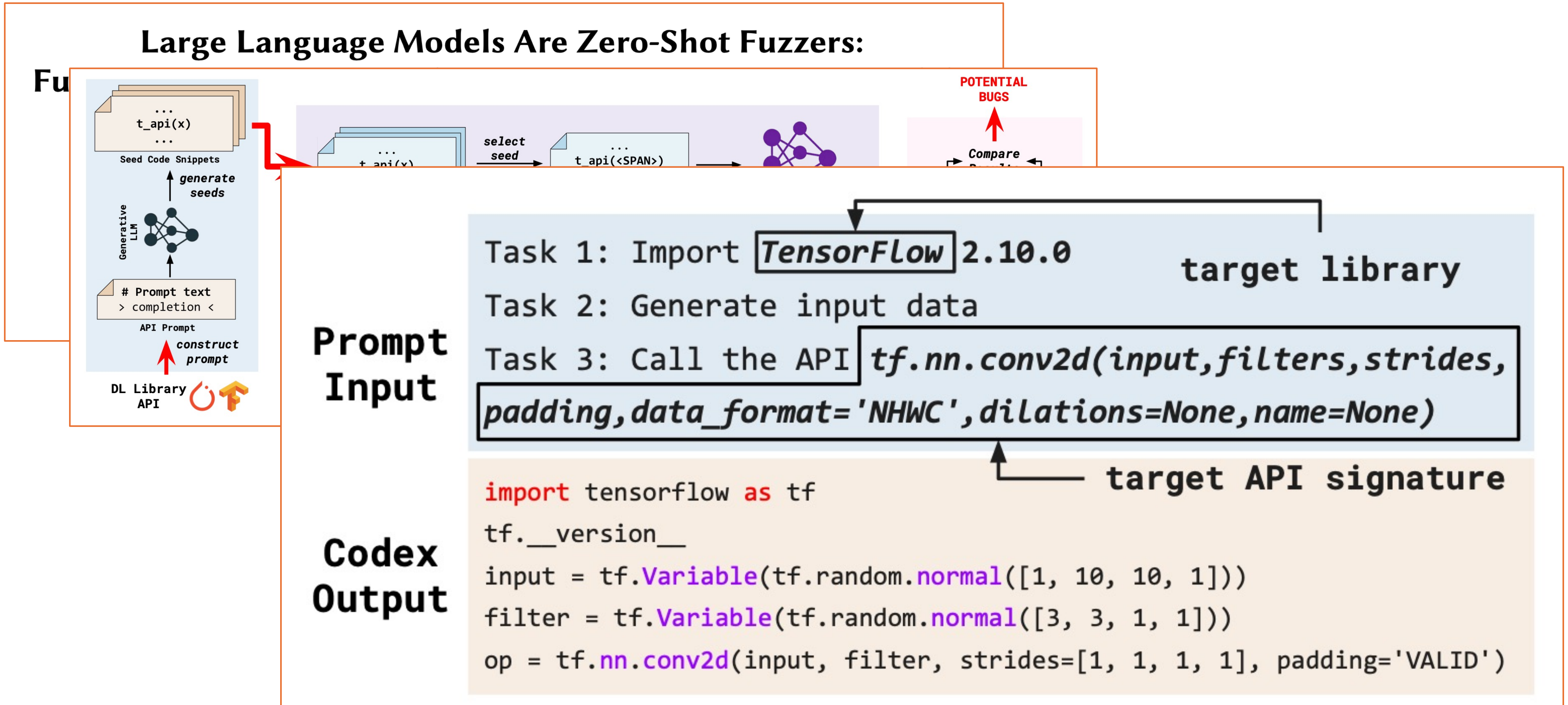


Figure 5: Example generation from the Codex model.

Large Language Models Are Zero-Shot Fuzzers:

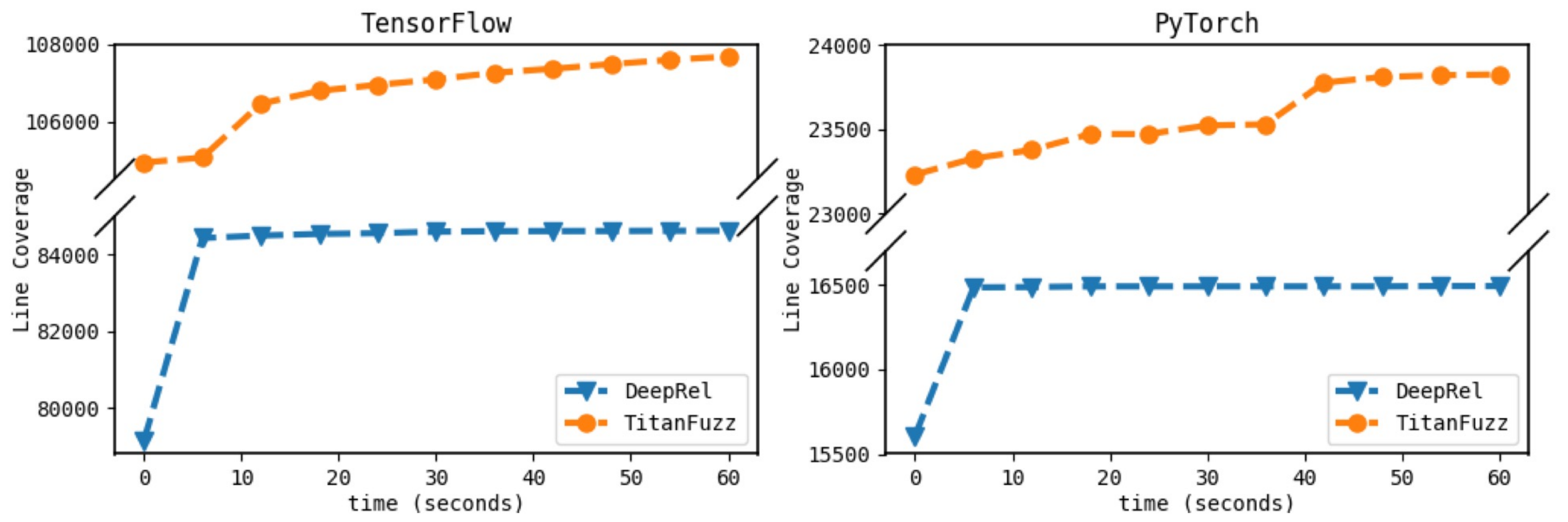
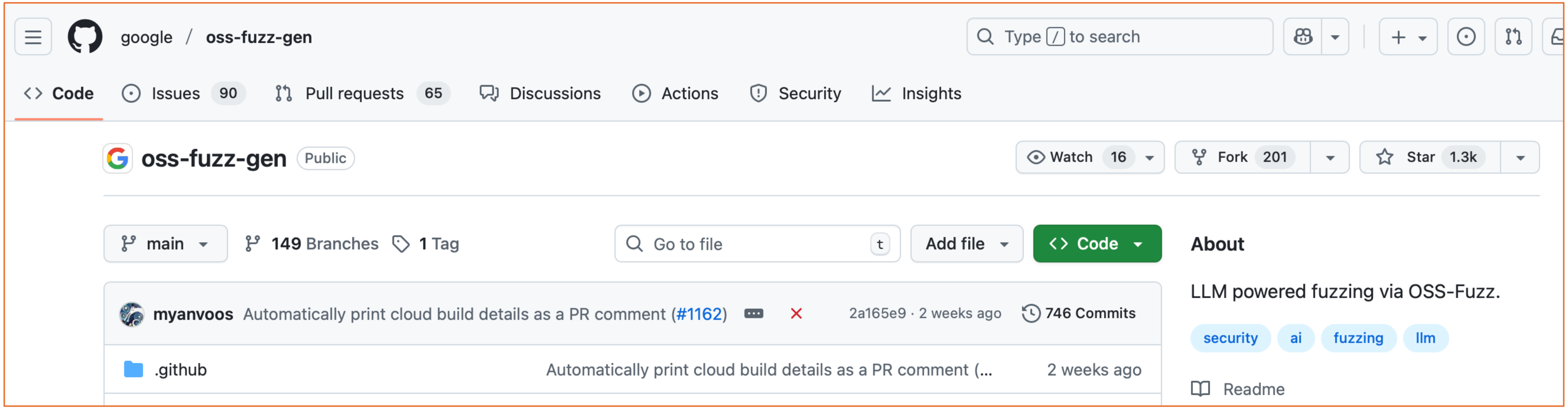
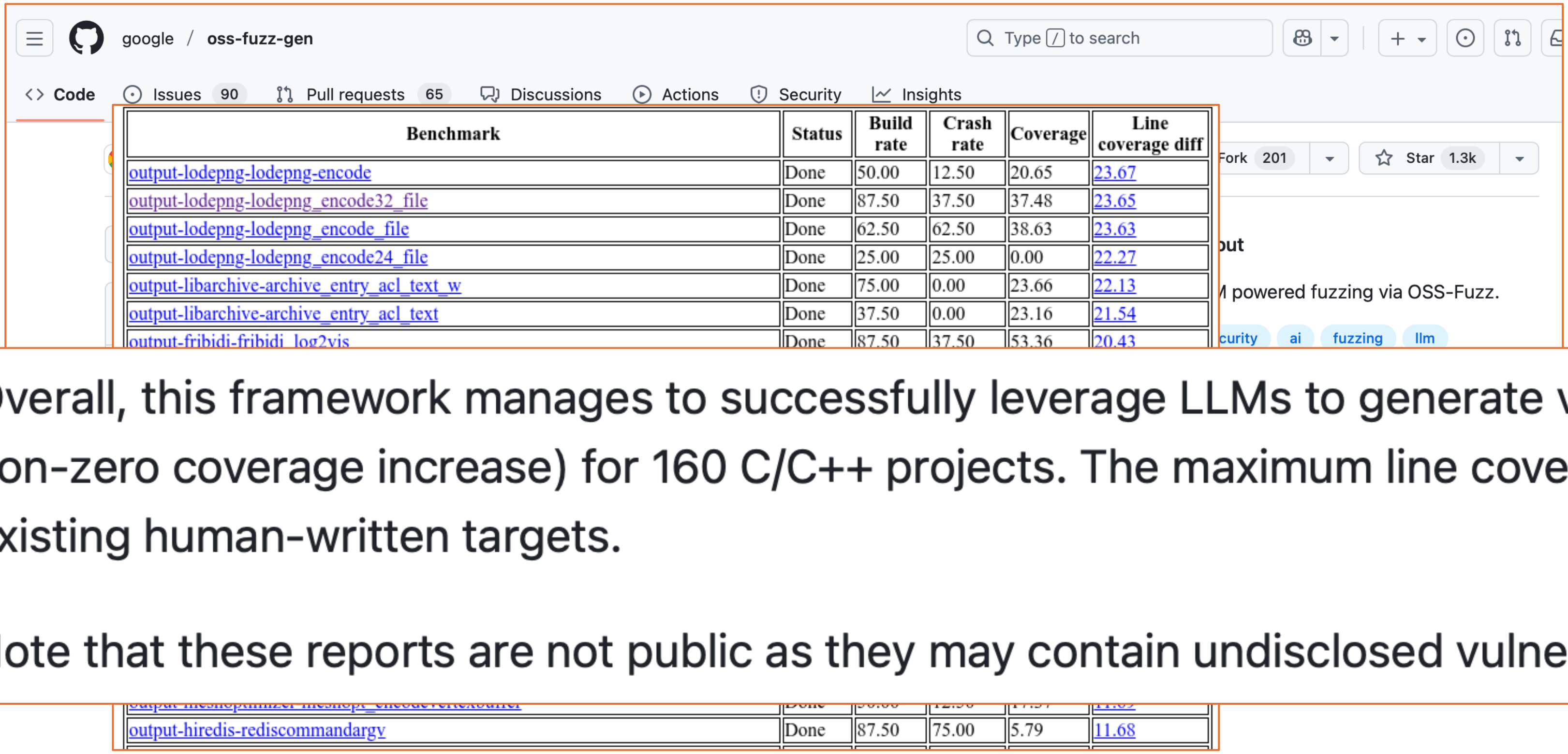


Figure 7: Coverage trend against DeepREL



https://github.com/google/oss-fuzz-gen



Benchmark	Status	Build rate	Crash rate	Coverage	Line coverage diff
output-lodepng-lodepng-encode	Done	50.00	12.50	20.65	23.67
output-lodepng-lodepng_encode32_file	Done	87.50	37.50	37.48	23.65
output-lodepng-lodepng_encode_file	Done	62.50	62.50	38.63	23.63
output-lodepng-lodepng_encode24_file	Done	25.00	25.00	0.00	22.27
output-libarchive-archive_entry_acl_text_w	Done	75.00	0.00	23.66	22.13
output-libarchive-archive_entry_acl_text	Done	37.50	0.00	23.16	21.54
output-fribidi-fribidi_log2vis	Done	87.50	37.50	53.36	20.43
output-hiredis-rediscommandargv	Done	87.50	75.00	5.79	11.68

Overall, this framework manages to successfully leverage LLMs to generate valid fuzz targets (which generate non-zero coverage increase) for 160 C/C++ projects. The maximum line coverage increase is 29% from the existing human-written targets.

Note that these reports are not public as they may contain undisclosed vulnerabilities.

https://github.com/google/oss-fuzz-gen