Loop

A Patch Circuit That Iteratively Generates Patch Candidates

Minjae Gwon
minjae.gwon@postech.ac.kr
https://bxta.kr

Introduction

- Motivation: CodeRL
- Suggestion: Loop

Loop

- Overview
- Components

Example

- CADET-00001

DevOps

- Objective
- Methods

POSTECH

Introduction

Motivation and Suggestion

POSTECH

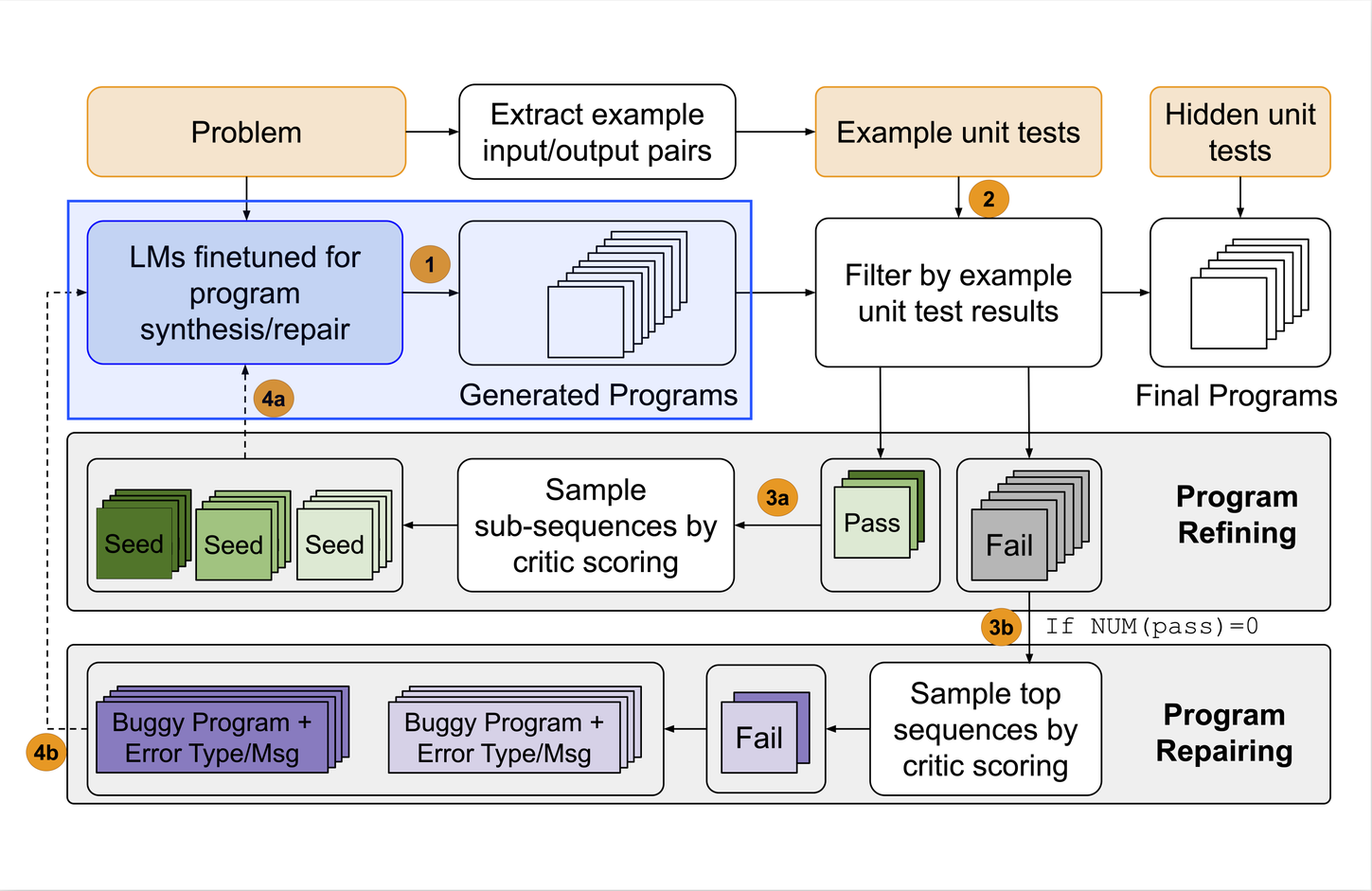
Motivation: CodeRL

Introduction

- A reinforcement learning framework.
  - Applies unit tests to an Actor-Critic model to solve a given problem.

Motivation: CodeRL

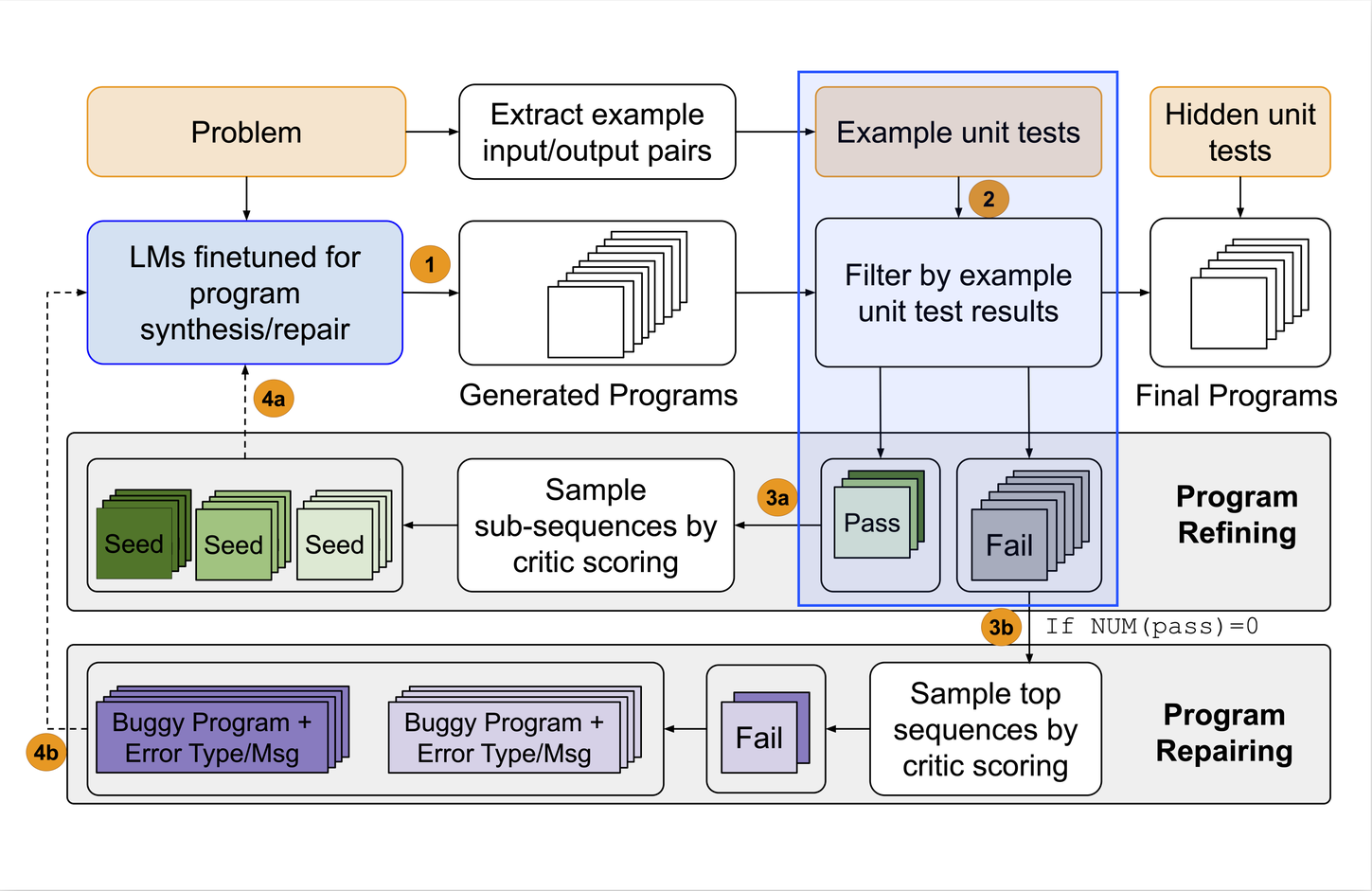
Introduction



1. Actor generates N code samples.
   - The Actor is a neural network that generates code samples.
   - The N code samples are generated based on the problem description.

Motivation: CodeRL

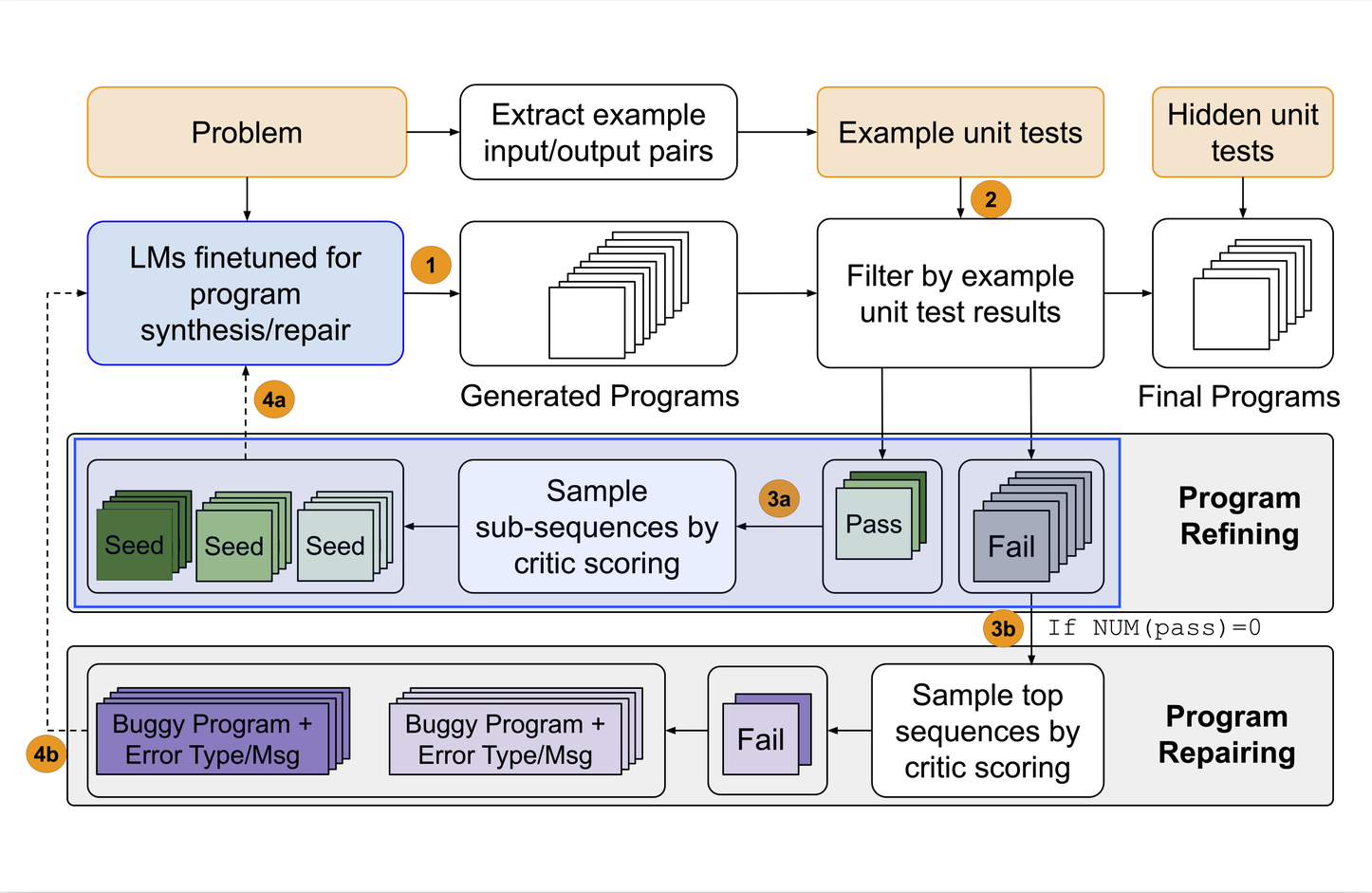
Introduction



2. The code samples are classified through unit tests.
   - The unit tests are used to evaluate the correctness of the code samples.
   - A code sample is classified as "unit test failed" if it fails at least one of the unit tests.

Motivation: CodeRL

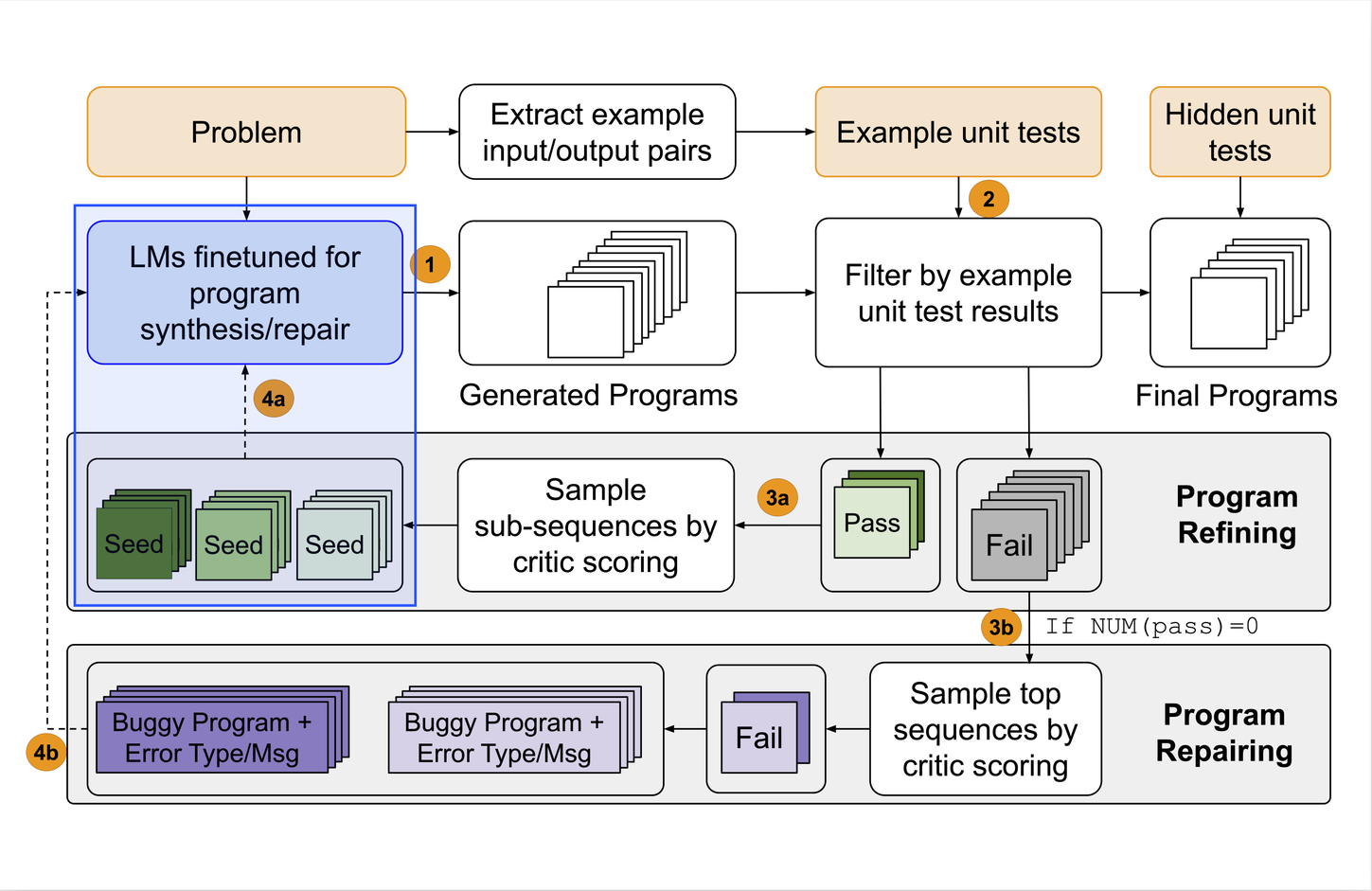
Introduction



3. Sub-sequencing is performed based on the scores of the Critic model for each sample.
   - The Critic is a neural network that assigns scores to code samples.

Motivation: CodeRL

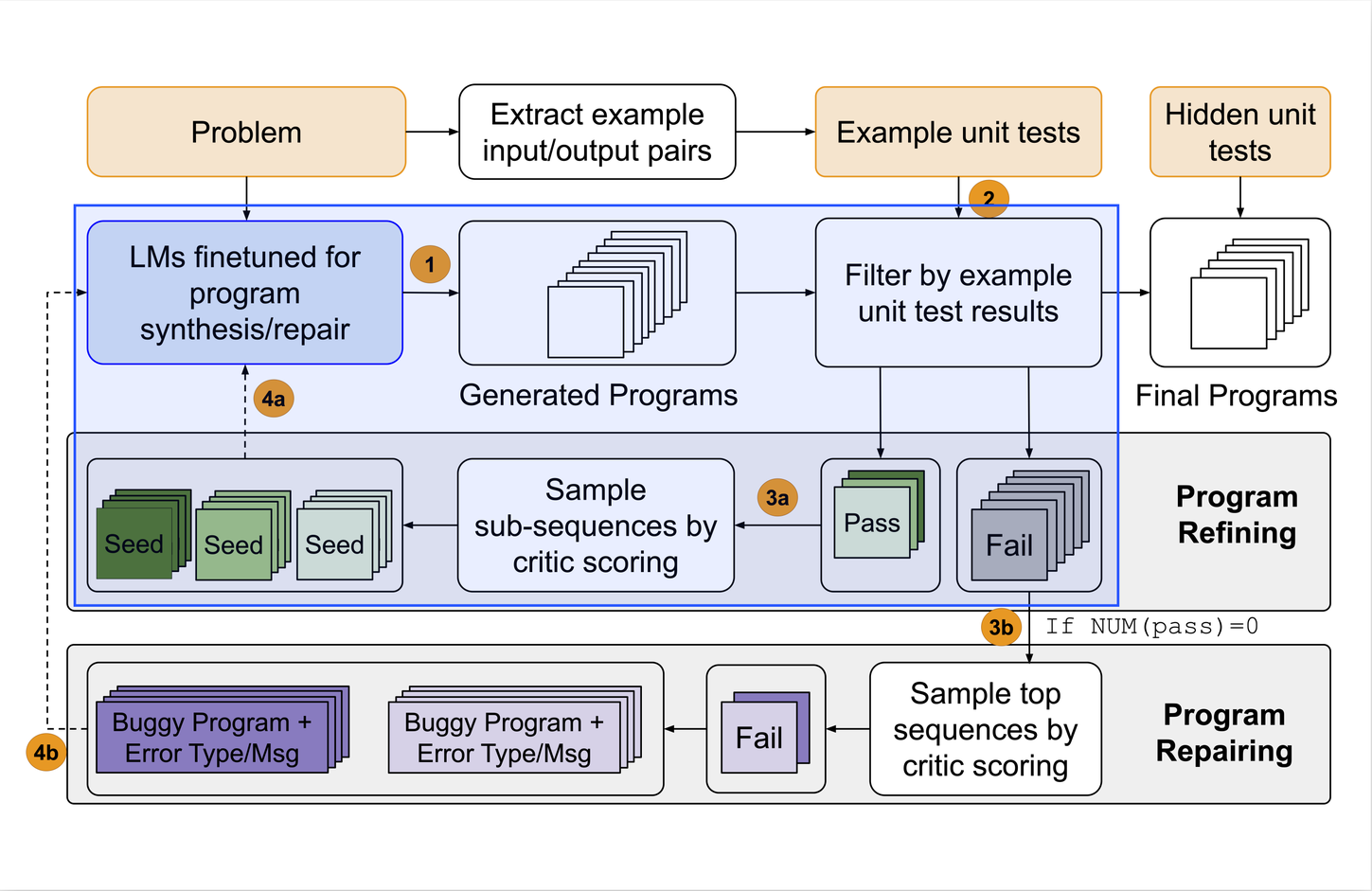
Introduction



4. The subsequences are considered seeds and shown to the Actor.
   - The Actor uses the subsequences as seeds to generate new code samples.
   - i.e. Code completion tasks.

Motivation: CodeRL

Introduction



5. The process continues until the condition is met according to the policy.
   - i.e. the results of the unit tests and the number of iterations.

Suggestion: Loop

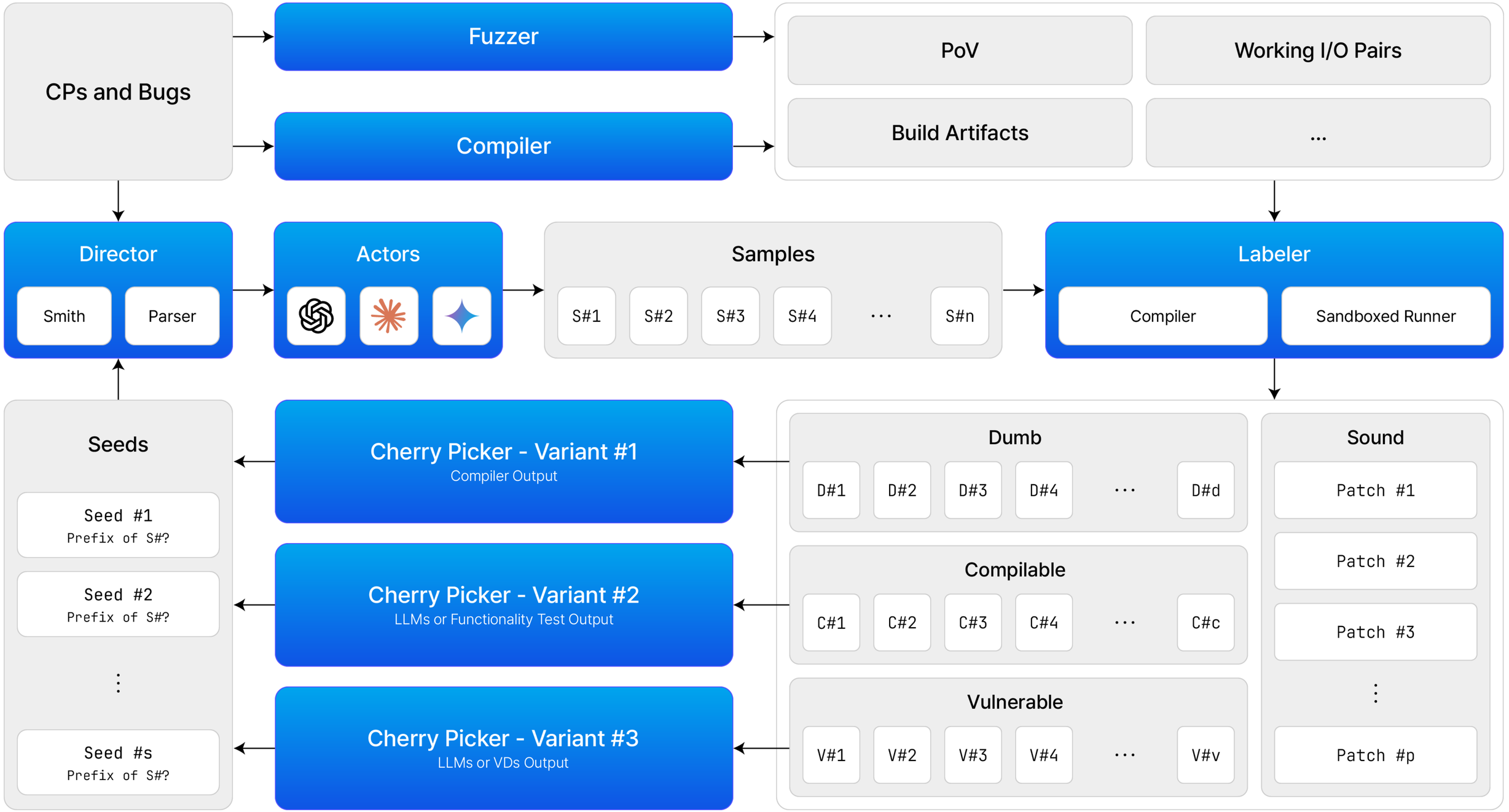
Introduction

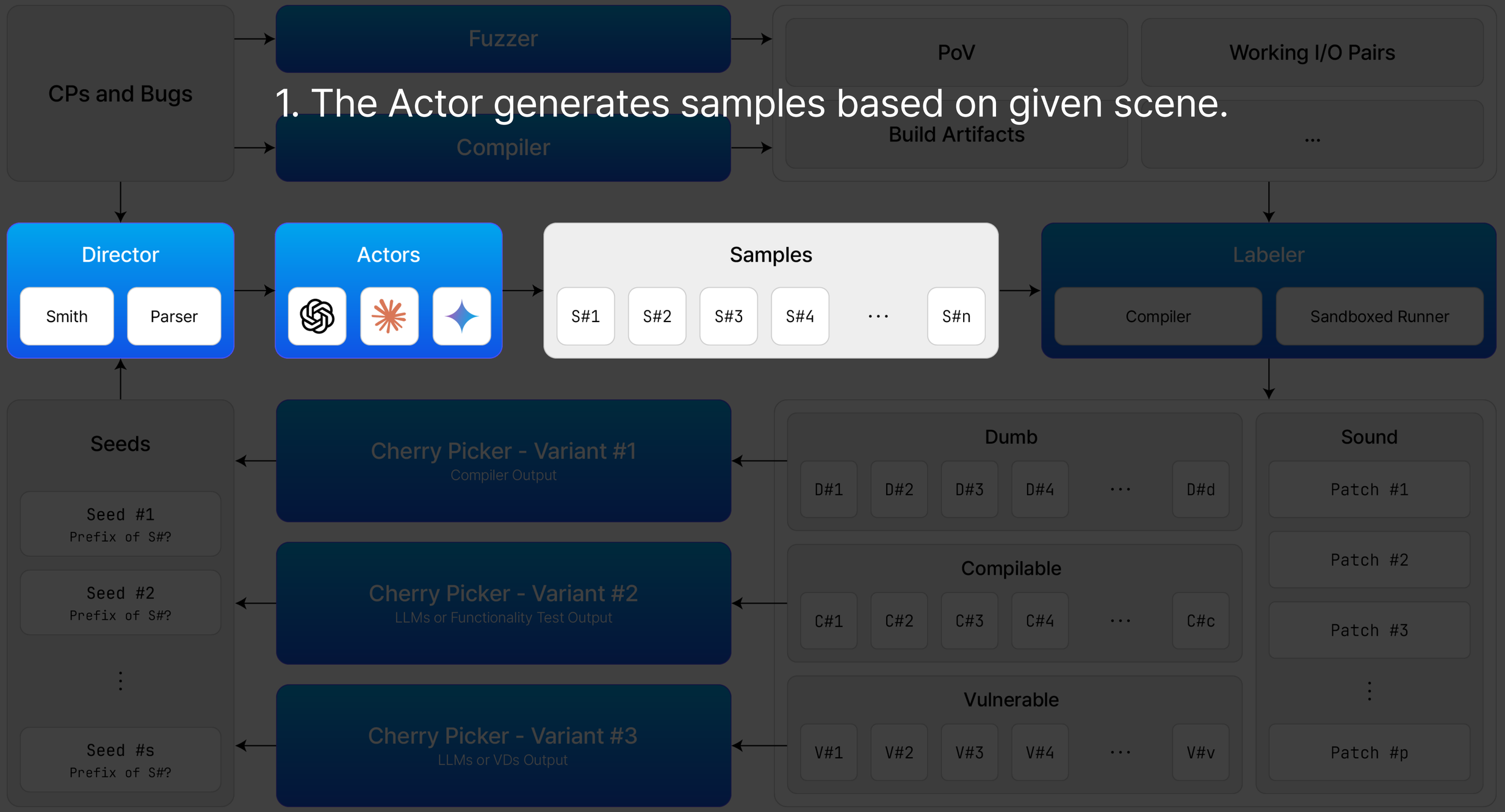
- Loop is a patch circuit that iteratively generates patch candidates.
  - Inspired by the code generation process of CodeRL.
- Loop can leverage various feedback mechanisms.
  - Such as fuzzers or vulnerability detectors (VDs).
  - This iterative approach allows Loop to explore:
    - A wider range of potential solutions.
    - Identify effective patches for the program.

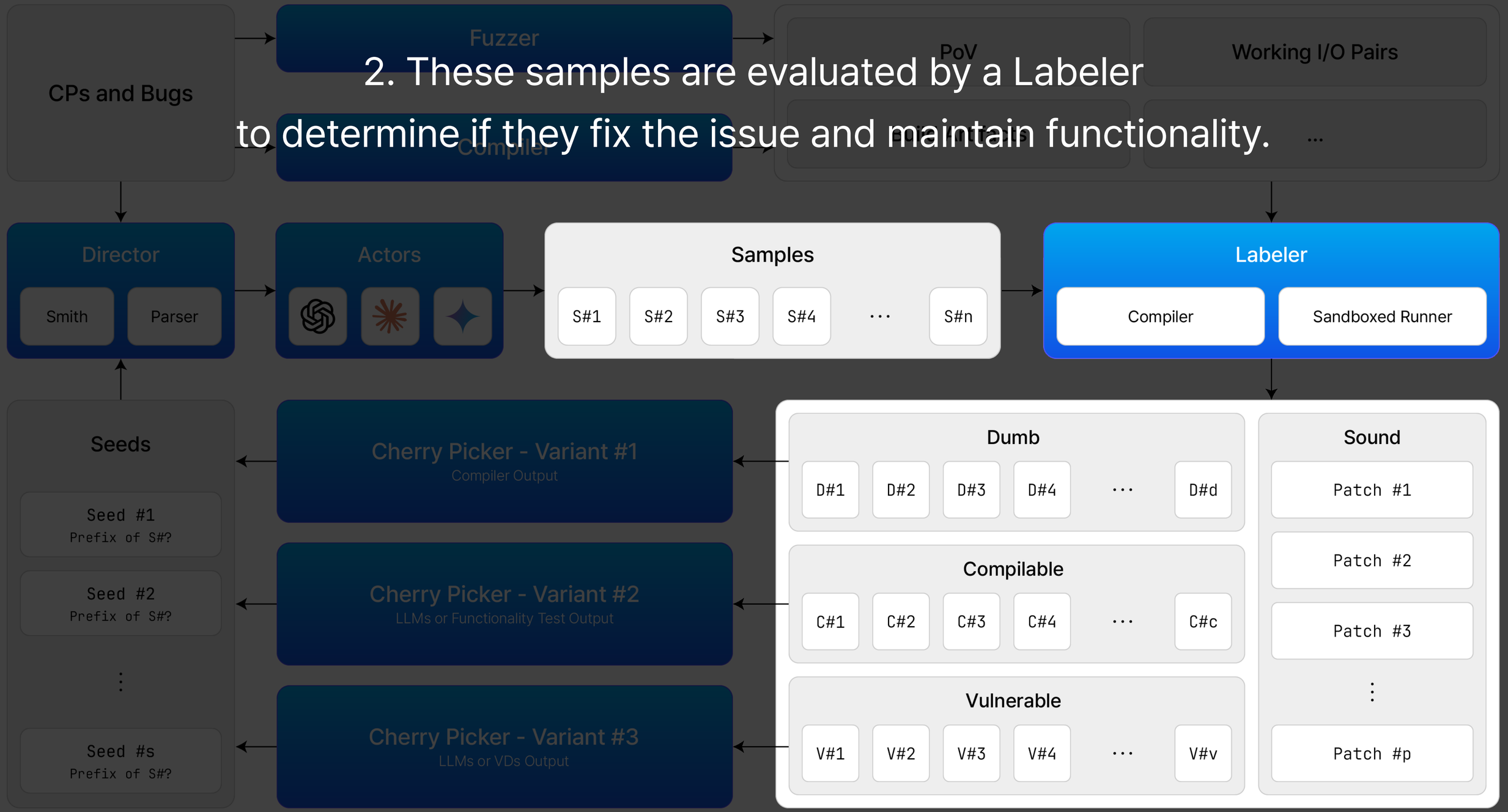
Loop

Overview and Components

POSTECH

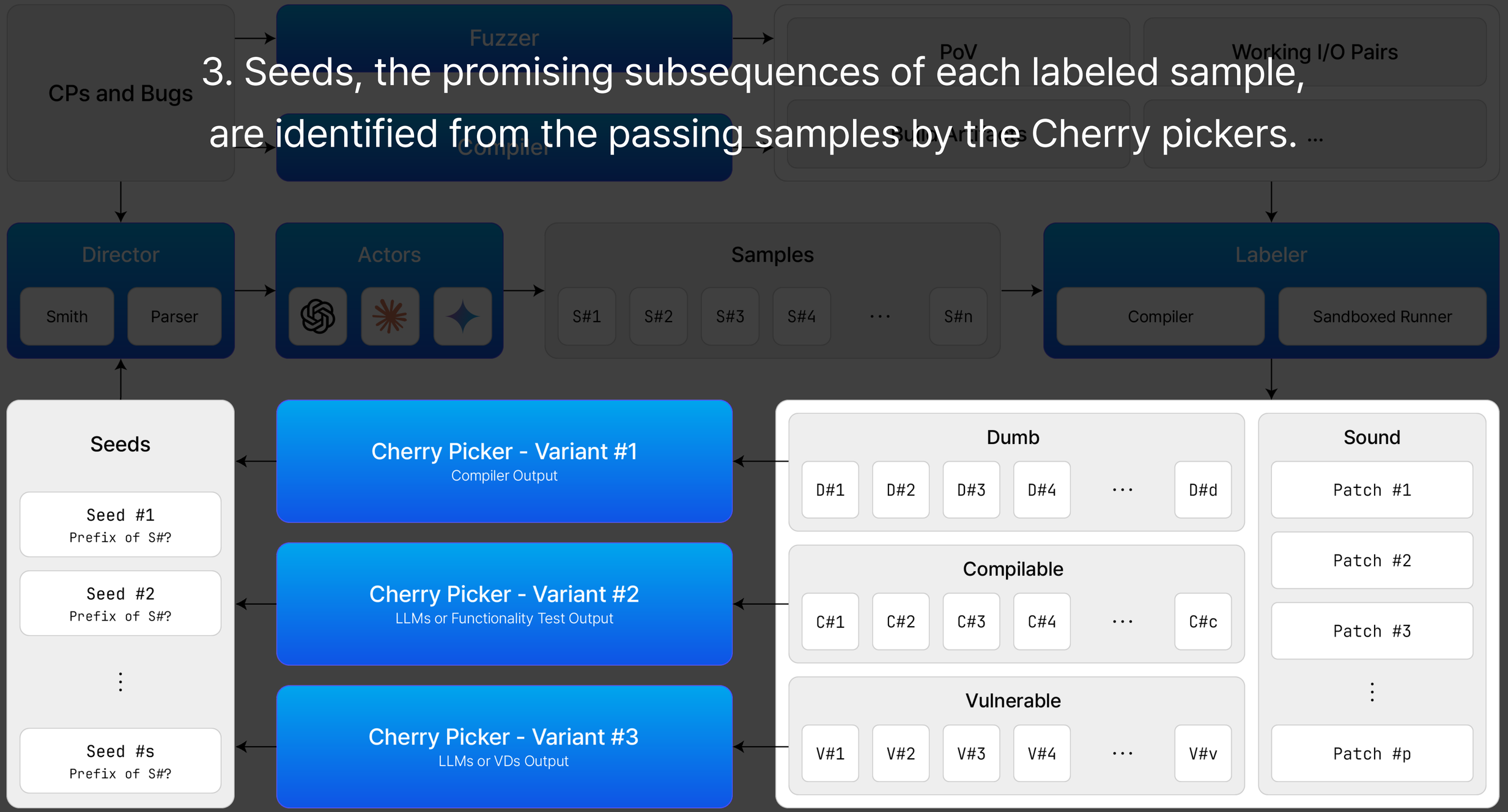

CPs and Bugs
Fuzzer
Compiler
PoV
Working I/O Pairs
Build Artifacts
...
Director
Smith
Parser
Actors
Samples
S#1
S#2
S#3
S#4
...
S#n
Labeler
Compiler
Sandboxed Runner
Seeds
Seed #1
Prefix of S#?
Seed #2
Prefix of S#?
Seed #s
Prefix of S#?
Cherry Picker - Variant #1
Compiler Output
Cherry Picker - Variant #2
LLMs or Functionality Test Output
Cherry Picker - Variant #3
LLMs or VDs Output
Dumb
D#1
D#2
D#3
D#4
...
D#d
Compilable
C#1
C#2
C#3
C#4
...
C#c
Vulnerable
V#1
V#2
V#3
V#4
...
V#v
Sound
Patch #1
Patch #2
Patch #3
Patch #p


CPs and Bugs
Fuzzer
Compiler
PoV
Working I/O Pairs
Build Artifacts
...
1. The Actor generates samples based on given scene.
Director
Smith
Parser
Actors
Samples
S#1
S#2
S#3
S#4
...
S#n
Labeler
Compiler
Sandboxed Runner
Seeds
Seed #1
Prefix of S#?
Seed #2
Prefix of S#?
Seed #s
Prefix of S#?
Cherry Picker - Variant #1
Compiler Output
Cherry Picker - Variant #2
LLMs or Functionality Test Output
Cherry Picker - Variant #3
LLMs or VDs Output
Dumb
D#1
D#2
D#3
D#4
...
D#d
Compilable
C#1
C#2
C#3
C#4
...
C#c
Vulnerable
V#1
V#2
V#3
V#4
...
V#v
Sound
Patch #1
Patch #2
Patch #3
Patch #p

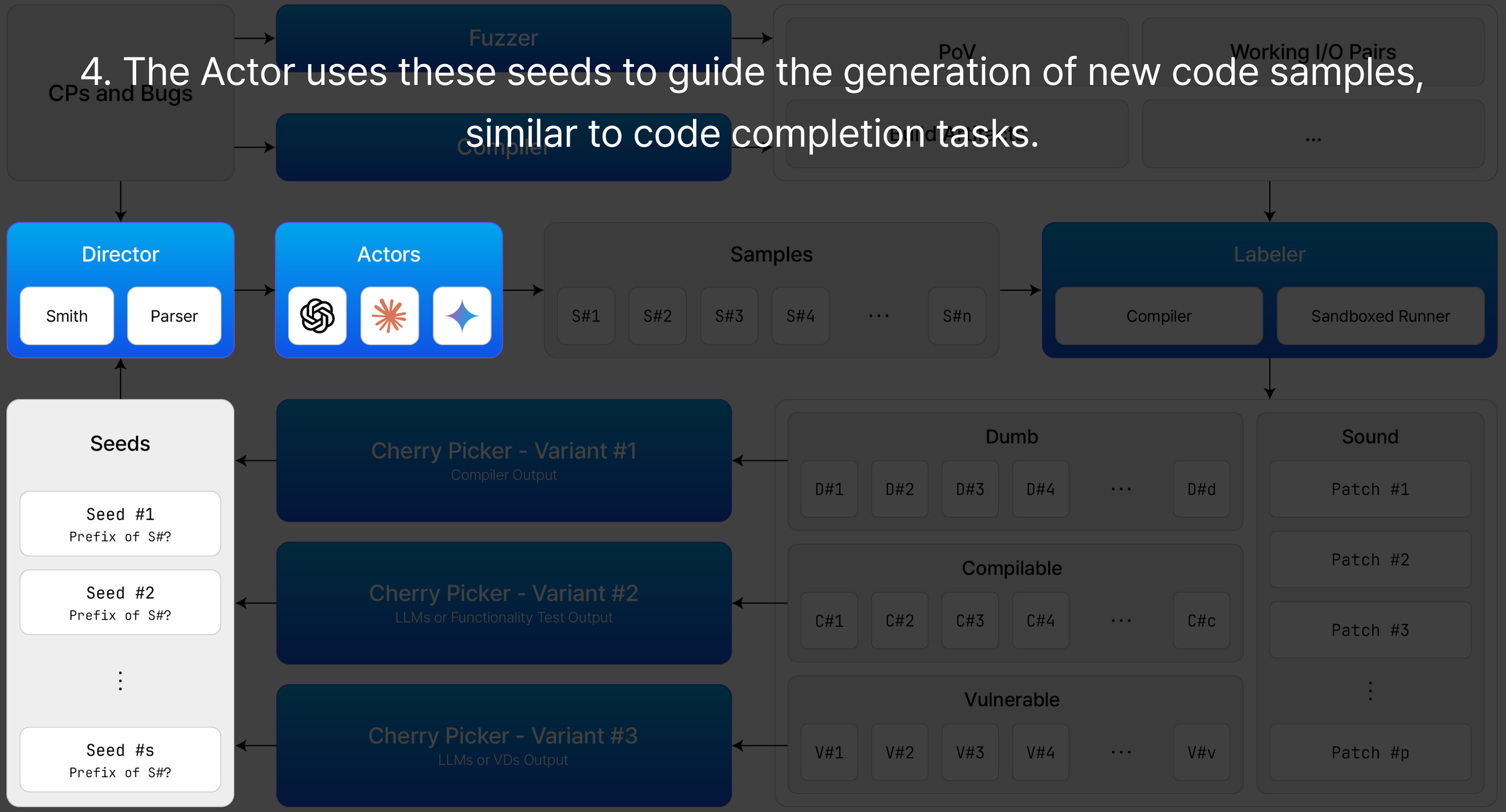


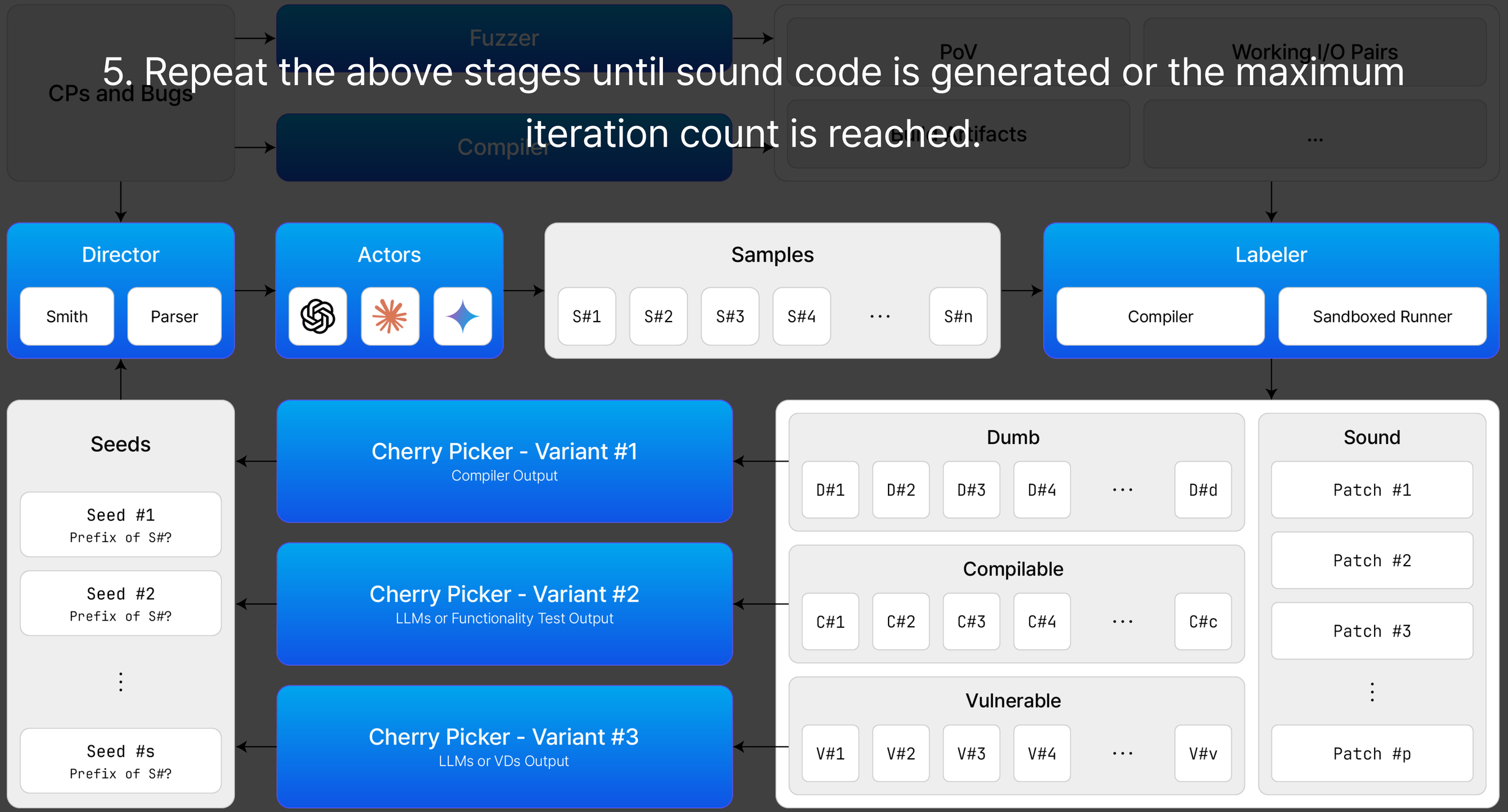
2. These samples are evaluated by a Labeler to determine if they fix the issue and maintain functionality.

3. Seeds, the promising subsequences of each labeled sample, are identified from the passing samples by the Cherry pickers. ...



4. The Actor uses these seeds to guide the generation of new code samples, similar to code completion tasks.




5. Repeat the above stages until sound code is generated or the maximum iteration count is reached.
CPs and Bugs
Fuzzer
Compiler
PoV
Working I/O Pairs
Artifacts
...
Director
Smith
Parser
Actors
Samples
S#1
S#2
S#3
S#4
...
S#n
Labeler
Compiler
Sandboxed Runner
Seeds
Seed #1
Prefix of S#?
Seed #2
Prefix of S#?
Seed #s
Prefix of S#?
Cherry Picker - Variant #1
Compiler Output
Cherry Picker - Variant #2
LLMs or Functionality Test Output
Cherry Picker - Variant #3
LLMs or VDs Output
Dumb
D#1
D#2
D#3
D#4
...
D#d
Compilable
C#1
C#2
C#3
C#4
...
C#c
Vulnerable
V#1
V#2
V#3
V#4
...
V#v
Sound
Patch #1
Patch #2
Patch #3
Patch #p

Director and Scene

Loop

- The scene acts as a blueprint for the Actor.
- The director composes scene based on challenges and bugs.
- It provides key information about the bug:
  - **File;** Contents of a file.
  - **Location;** Function name where the bug occurs.
  - **Function;** i.e. “check”.

Director and Scene

Loop

- Question: How to enhance the scene?
  - Smith
    - Some localization data from Smith would be beneficial for improve the quality of the scene.
  - Fuzzer Data
    - Information from fuzzers, like identified crashes or unexpected outputs, could be included.
    - This helps the Actor understand the problem's symptoms and target the root cause.
  - Vulnerability Details
    - Data from VDs could be valuable.

Actor and Samples

Loop

- Think of Actor as a code-fixing factory.
  - It leverages the power of LLMs like Claude, ChatGPT, or Gemini.
- Actor generates multiple code samples.
  - Each representing a potential fix for the bug in the provided code.
  - The system can explore a wider range of solutions, increasing the chances of finding an effective patch.
- Question: Ensemble?
  - What about employing multiple LLMs to generate a variety of samples?
  - As many LLMs as we can use via LiteLLM!

Labeler and Labeled Samples

Loop

- Labeler analyzes each sample based on three key aspects:
  - **Compilability**: Does the code compile without errors?
  - **Functionality**: Does the code address the intended functionality as defined by the problem?
    - Functionality tests, often relying on input-output pairs generated by fuzzers, can help ensure the code doesn't introduce crashes.
  - **Safety**: Does the code introduce any new vulnerabilities or security risks?

Labeler and Labeled Samples

Loop

- Based on this evaluation, the Labeler assigns a category to each sample:

	Compilablity	Functionality	Safety
Dumb	X	-	-
Compilable	O	X	-
Vulnerable	O	O	X
Sound	O	O	O

Cherry Picker and Seeds

Loop

- Cherry pickers identify promising subsequences from the samples.
- These subsequences, then referred to as seeds, serve as starting points for the Actor.

Cherry Picker and Seeds

Loop

- The cherry picking strategy varies depending on the category of the sample:
  - Dumb Samples:
    - Find prefixes that is compilable before encountering errors.
  - Compilable Samples:
    - Find the longest prefix before the code breaks functionality using LLMs via prompting.
    - Alternatively, minimal common paths identified through functionality tests can be used as seeds.
  - Vulnerable Samples:
    - Find the safe prefix using LLMs.
    - Alternatively, use the prefix before the line pinpointed by VDs.
      - Note that this approach cannot cover situations where the buggy location and the patch location differ.

Cherry Picker and Seeds

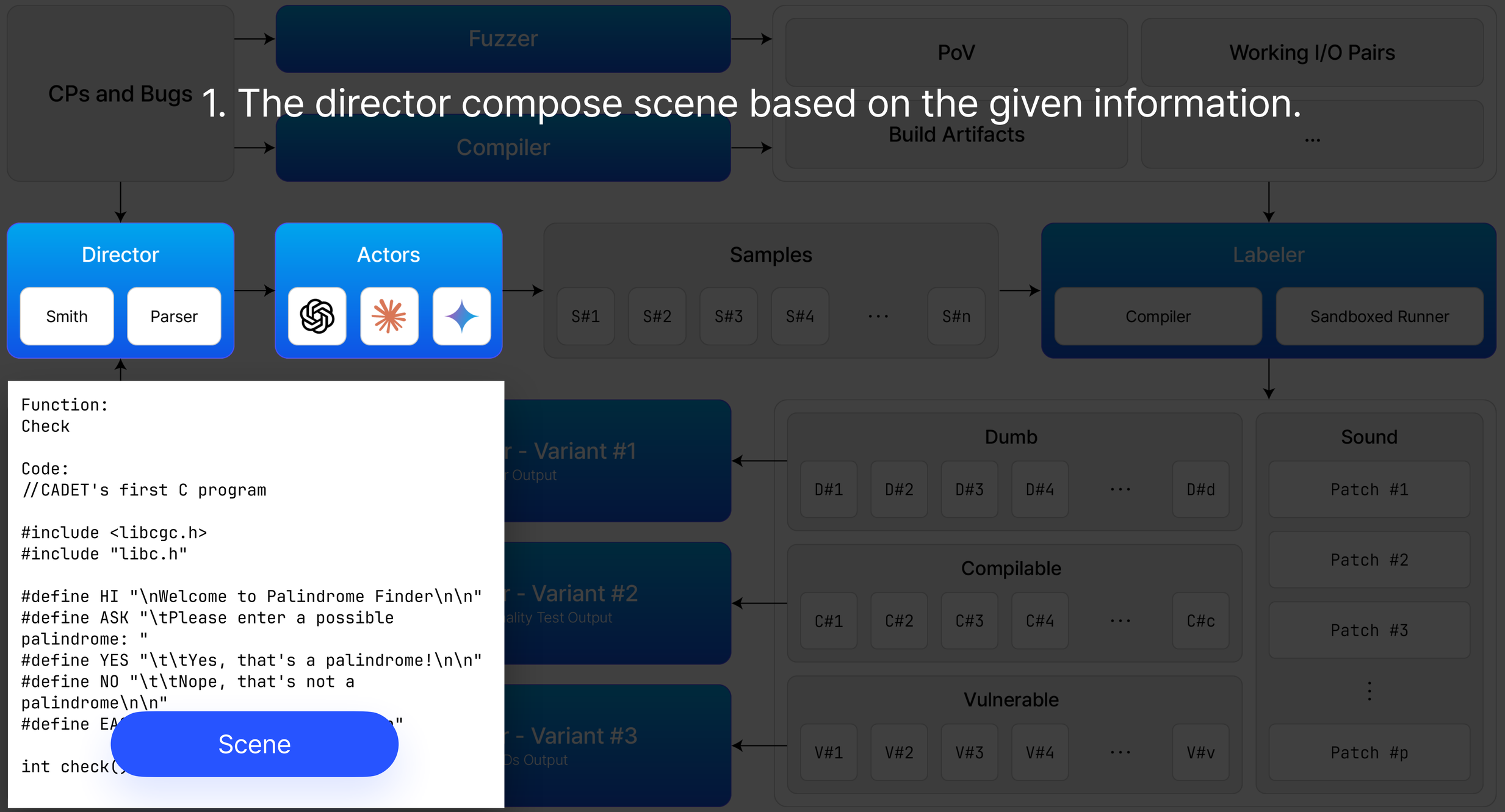
Loop

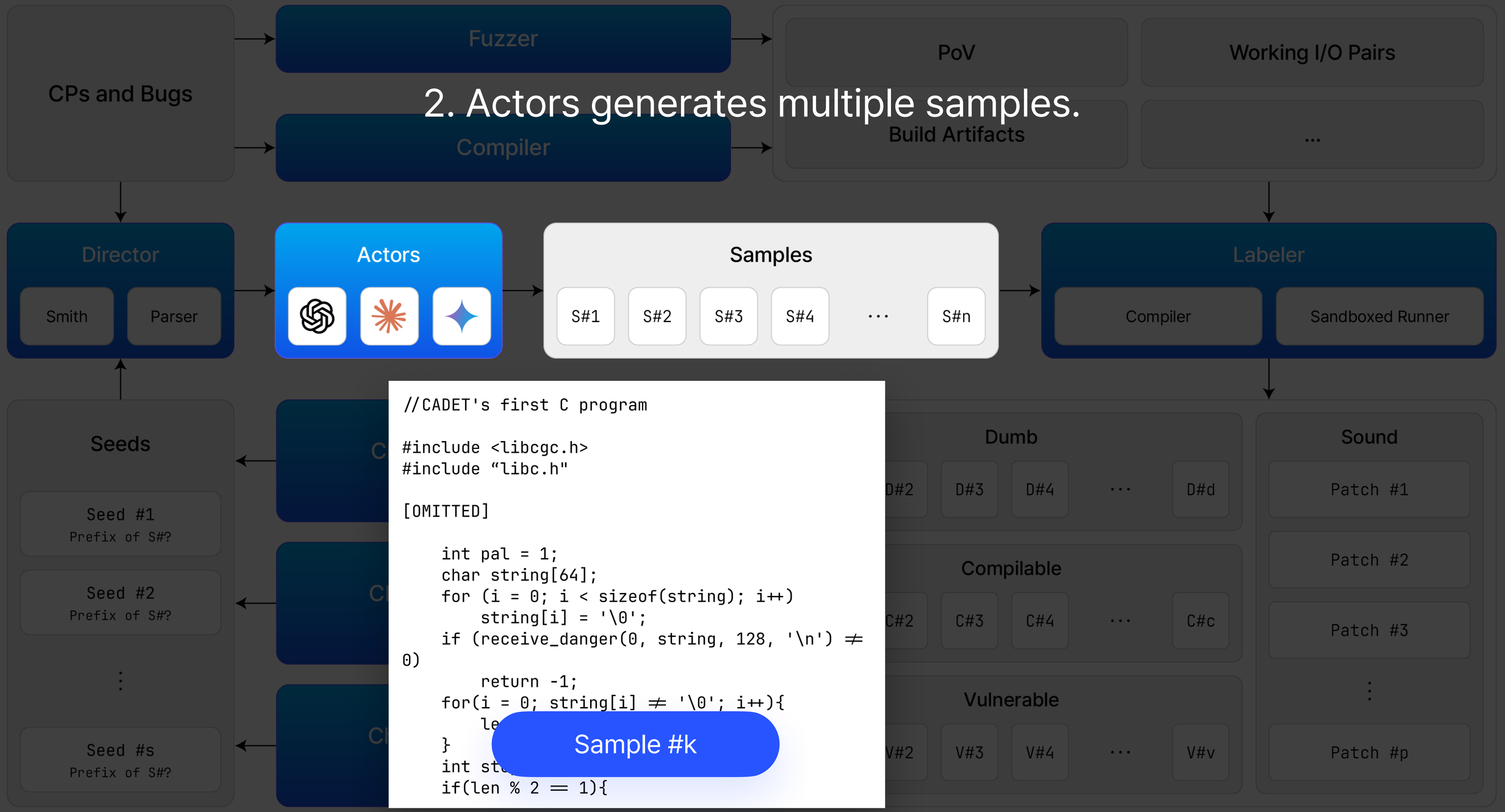
- Question: Local Critic Model
  - CodeRL relies on a powerful local model called the Critic.
    - Critic doesn't require a complete code snippet.
    - Critic can predict the possibilities of compilation failure, runtime failure, unit test failure, and soundness of incomplete code.
  - If we could use a simple quantized Seq2Seq model as the Critic within a sandbox, could it be a promising option?
    - Not that the authors report that the Critic could be a small model.
- Question: Prompt
  - What additional information could be beneficial to include in prompts when cherry-picking?

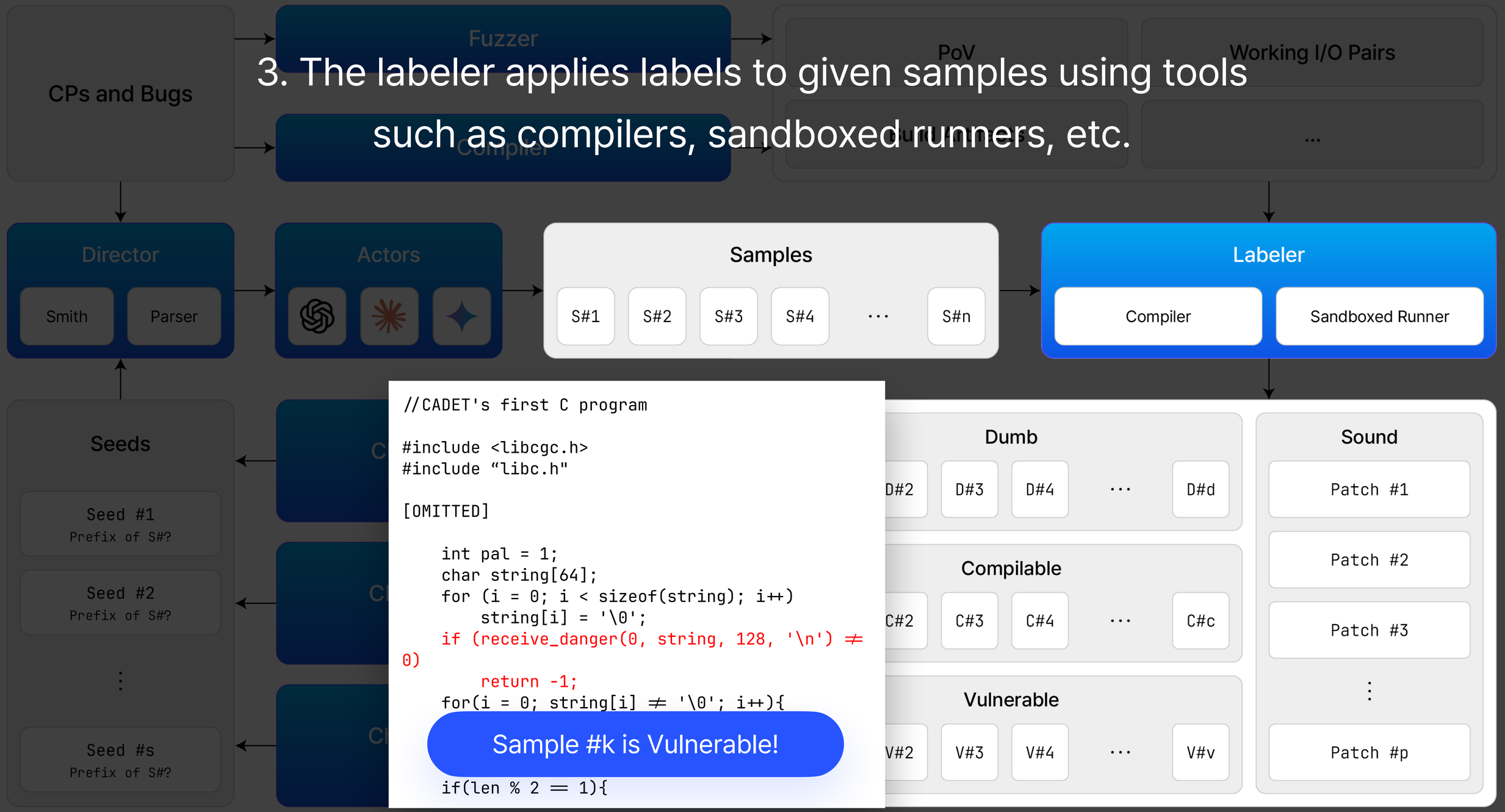
Example

CADET-00001

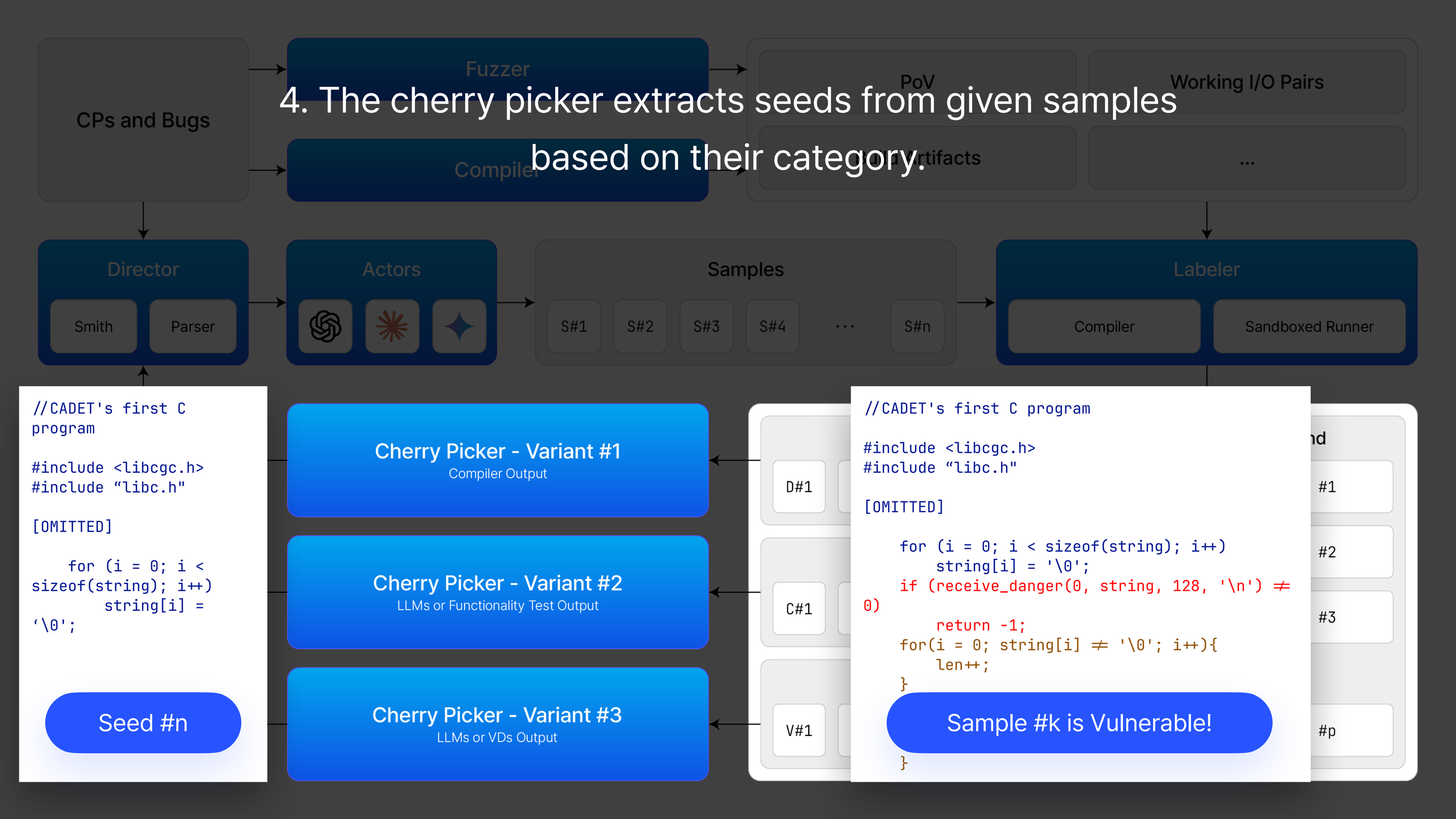
POSTECH

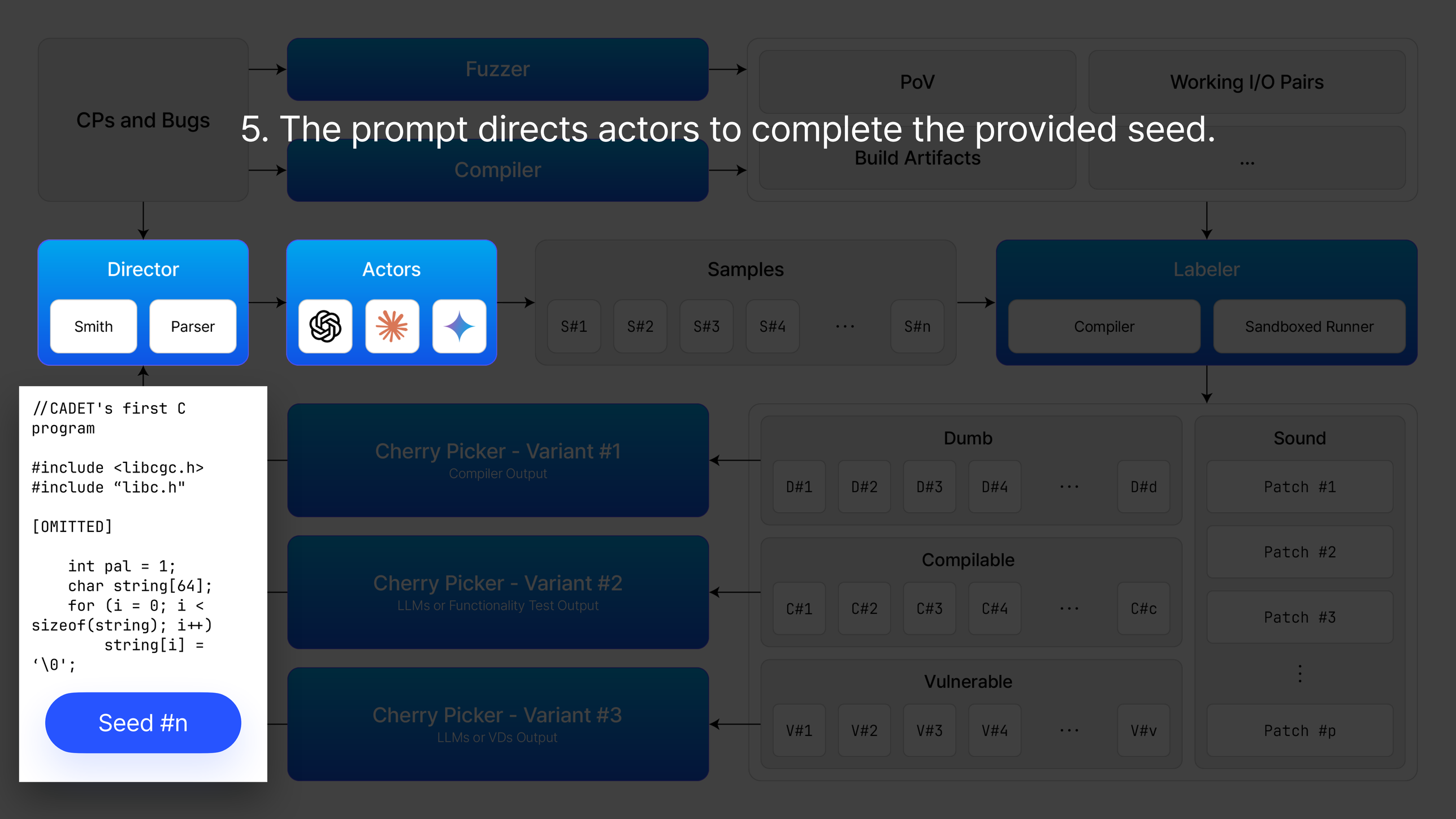

CPs and Bugs
Fuzzer
Compiler
PoV
Working I/O Pairs
Build Artifacts
...
1. The director compose scene based on the given information.
Director
Smith
Parser
Actors
Samples
S#1
S#2
S#3
S#4
...
S#n
Labeler
Compiler
Sandboxed Runner
Function:
Check
Code:
//CADET's first C program
#include <libcgc.h>
#include "libc.h"
#define HI "\nWelcome to Palindrome Finder\n\n"
#define ASK "\tPlease enter a possible
palindrome: "
#define YES "\t\tYes, that's a palindrome!\n\n"
#define NO "\t\tNope, that's not a
palindrome\n\n"
#define EA
int check(
Scene
- Variant #1
Output
- Variant #2
ality Test Output
- Variant #3
Ds Output
Dumb
D#1
D#2
D#3
D#4
...
D#d
Compilable
C#1
C#2
C#3
C#4
...
C#c
Vulnerable
V#1
V#2
V#3
V#4
...
V#v
Sound
Patch #1
Patch #2
Patch #3
Patch #p


CPs and Bugs
Fuzzer
Compiler
PoV
Working I/O Pairs
Build Artifacts
...
2. Actors generates multiple samples.
Director
Smith
Parser
Actors
Samples
S#1
S#2
S#3
S#4
...
S#n
Labeler
Compiler
Sandboxed Runner
Seeds
Seed #1
Prefix of S#?
Seed #2
Prefix of S#?
Seed #s
Prefix of S#?
//CADET's first C program
#include <libcgc.h>
#include “libc.h"
[OMITTED]
int pal = 1;
char string[64];
for (i = 0; i < sizeof(string); i++)
string[i] = '\0';
if (receive_danger(0, string, 128, '\n') ≠ 0)
return -1;
for(i = 0; string[i] ≠ '\0'; i++){
}
if(len % 2 == 1){
Sample #k
Dumb
D#2
D#3
D#4
...
D#d
Compilable
C#2
C#3
C#4
...
C#c
Vulnerable
V#2
V#3
V#4
...
V#v
Sound
Patch #1
Patch #2
Patch #3
Patch #p

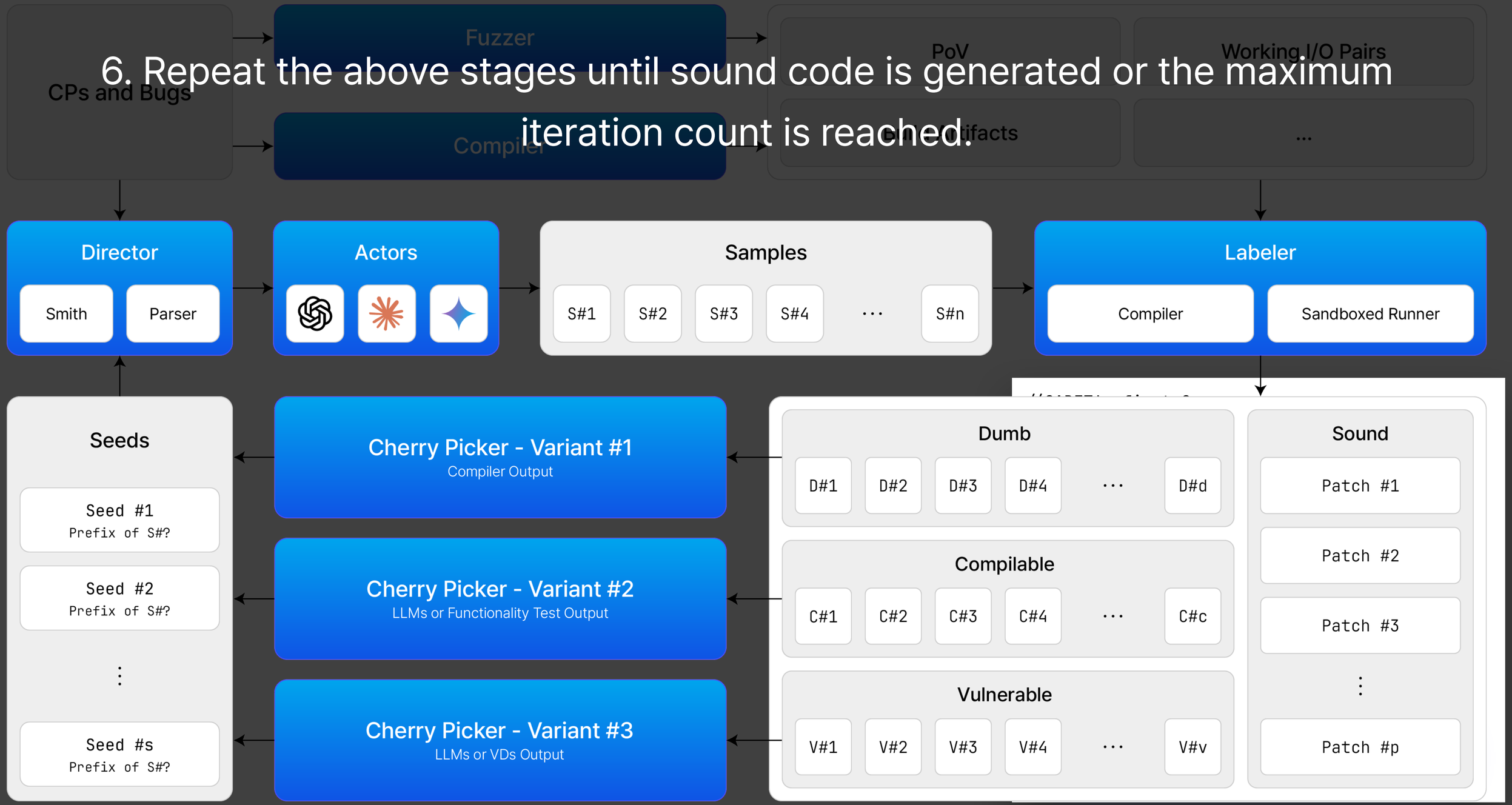


3. The labeler applies labels to given samples using tools such as compilers, sandboxed runners, etc.


4. The cherry picker extracts seeds from given samples based on their category.
CPs and Bugs
Fuzzer
Compiler
PoV
Working I/O Pairs
...
Director
Smith
Parser
Actors
Samples
S#1
S#2
S#3
S#4
S#n
Labeler
Compiler
Sandboxed Runner
//CADET's first C program
#include <libcgc.h>
#include “libc.h"
[OMITTED]
for (i = 0; i < sizeof(string); i++)
string[i] = '\0';
Seed #n
Cherry Picker - Variant #1
Compiler Output
Cherry Picker - Variant #2
LLMs or Functionality Test Output
Cherry Picker - Variant #3
LLMs or VDs Output
D#1
C#1
V#1
#1
#2
#3
#p
//CADET's first C program
#include <libcgc.h>
#include “libc.h"
[OMITTED]
for (i = 0; i < sizeof(string); i++)
string[i] = '\0';
if (receive_danger(0, string, 128, '\n') != 0)
return -1;
for(i = 0; string[i] != '\0'; i++){
len++;
}
}
Sample #k is Vulnerable!


CPs and Bugs
Fuzzer
PoV
Working I/O Pairs
5. The prompt directs actors to complete the provided seed.
Compiler
Build Artifacts
...
Director
Smith
Parser
Actors
Samples
S#1
S#2
S#3
S#4
...
S#n
Labeler
Compiler
Sandboxed Runner
//CADET's first C program
#include <libcgc.h>
#include “libc.h"
[OMITTED]
int pal = 1;
char string[64];
for (i = 0; i < sizeof(string); i++)
string[i] = ‘\0';
Seed #n
Cherry Picker - Variant #1
Compiler Output
Cherry Picker - Variant #2
LLMs or Functionality Test Output
Cherry Picker - Variant #3
LLMs or VDs Output
Dumb
D#1
D#2
D#3
D#4
...
D#d
Compilable
C#1
C#2
C#3
C#4
...
C#c
Vulnerable
V#1
V#2
V#3
V#4
...
V#v
Sound
Patch #1
Patch #2
Patch #3
Patch #p

6. Repeat the above stages until sound code is generated or the maximum iteration count is reached.



DevOps

Objective and Methods

POSTECH

Objective: Empowering Everyone to Contribute

DevOps

- Contributors don't need to understand the inner workings of the framework.
  - To add new functionalities.
- The framework is designed for easy extension.
  - Allowing for seamless integration of new features.
- E.g. Django and Flask
  - Enable developers to focus solely on business logic.
  - Without delving into the framework's intricate details.

Methods

DevOps

- Separation of framework implementation from business logics.
- Dependency injection-based frameworks:
  - Hide complex logic through Inversion of Control.
  - Allowing feature developers to focus on functionality.
- Clear separation between framework maintainers and feature developers:
  - Framework maintainers build the core architecture
  - Feature developers concentrate on functionalities.
  - This empowers all developers and streamlines development.

Milestones

DevOps

- Phase 1: Lay the Groundwork
  - Establish a foundational framework for the circuit.
  - Validate the circuit's functionality using simulated services.
- Phase 2: Generate a Compilable Patch
  - Implement a MVP for the patching functionality
  - Only focus on ensuring the patch can be successfully compiled.
- Phase 3: Building a Sound Patch
  - Develop a comprehensive set of patching services.
    - Explore collaboration to integrate fuzzing tools and virtual devices (VDs) for enhanced testing.
  - Evaluate and refine the generated patch candidates to ensure their correctness.

Loop

A Patch Circuit That Iteratively Generates Patch Candidates

Minjae Gwon
minjae.gwon@postech.ac.kr
https://bxta.kr

ML@POSTECH
https://ml.postech.ac.kr