

# 효과적이고 효율적인 런타임 패칭 연구

김영재

Lab of Software@UNIST

# Contents

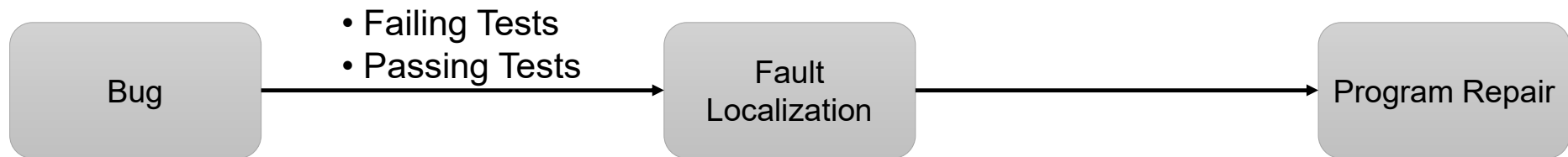
- 효과적인 런타임 패칭
  - Automated Program Repair (APR)
  - Jurigged
  - Runtime APR
  - Challenge
- 효율적인 패칭
  - APR and Fuzzing
  - Our Algorithm
  - Evaluation

# 효과적인 런타임 패칭

김영재

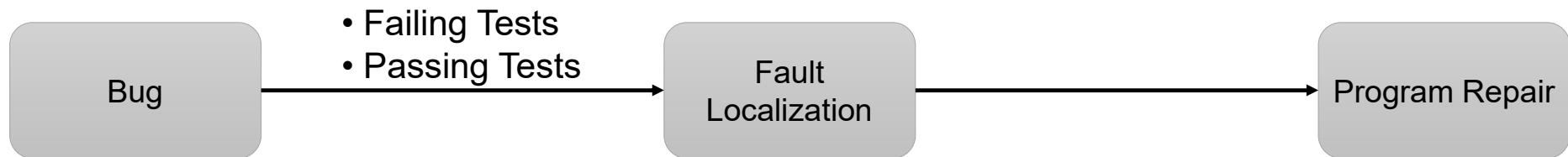
Lab of Software@UNIST

# Automated Program Repair



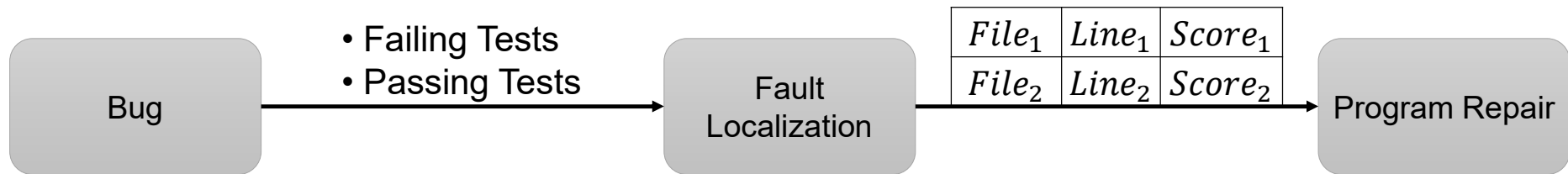
- Failing Tests: 버그를 유발하는 테스트들
- Passing Tests: 버그를 유발하지 않는 테스트들

# Automated Program Repair



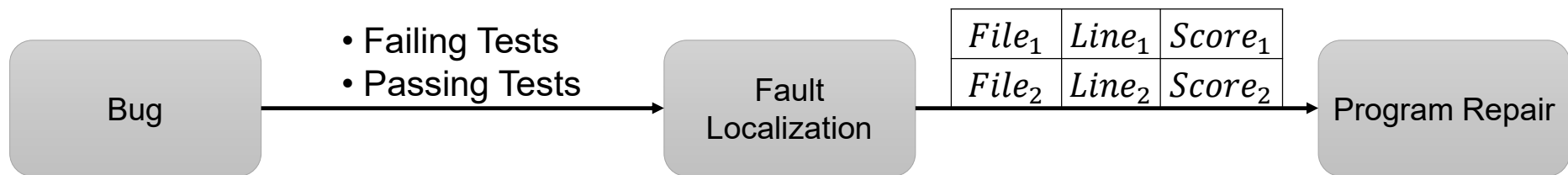
- Fault Localization (FL)
  - Failing과 Passing tests를 이용해 결함 위치 탐색
  - Spectrum-based Fault Localization (SBFL)

# Automated Program Repair



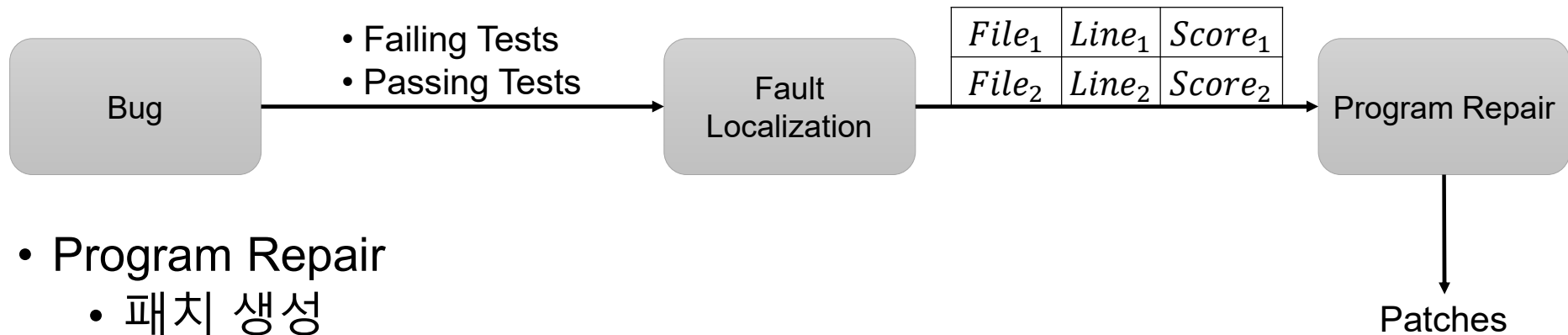
- Fault Localization (FL)
  - Failing과 Passing tests를 이용해 결함 위치 탐색
  - Spectrum-based Fault Localization (SBFL)
- File, Line, Score들의 목록

# Automated Program Repair



- Program Repair
  - 패치 생성
  - Template-based, Learning-based, ...

# Automated Program Repair

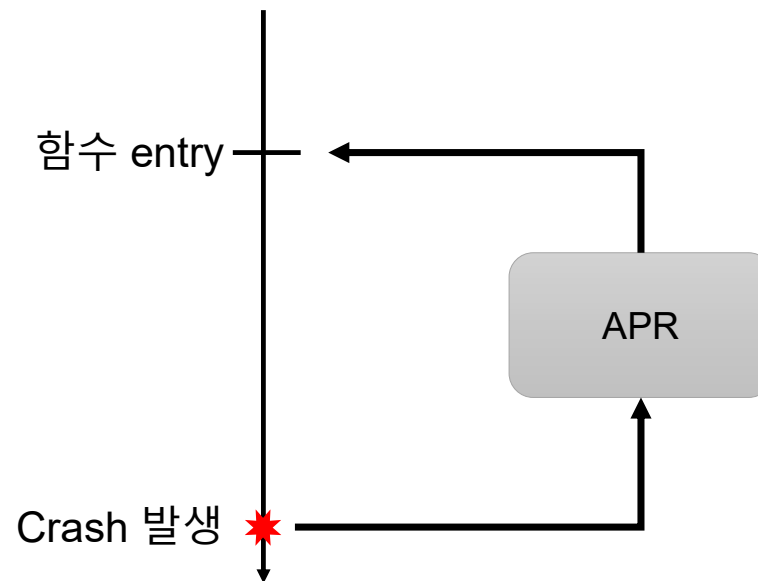


- Program Repair
  - 패치 생성
  - Template-based, Learning-based, ...
- Output
  - Valid (Plausible) patches
  - Failing과 Passing test들을 모두 pass하는 패치들



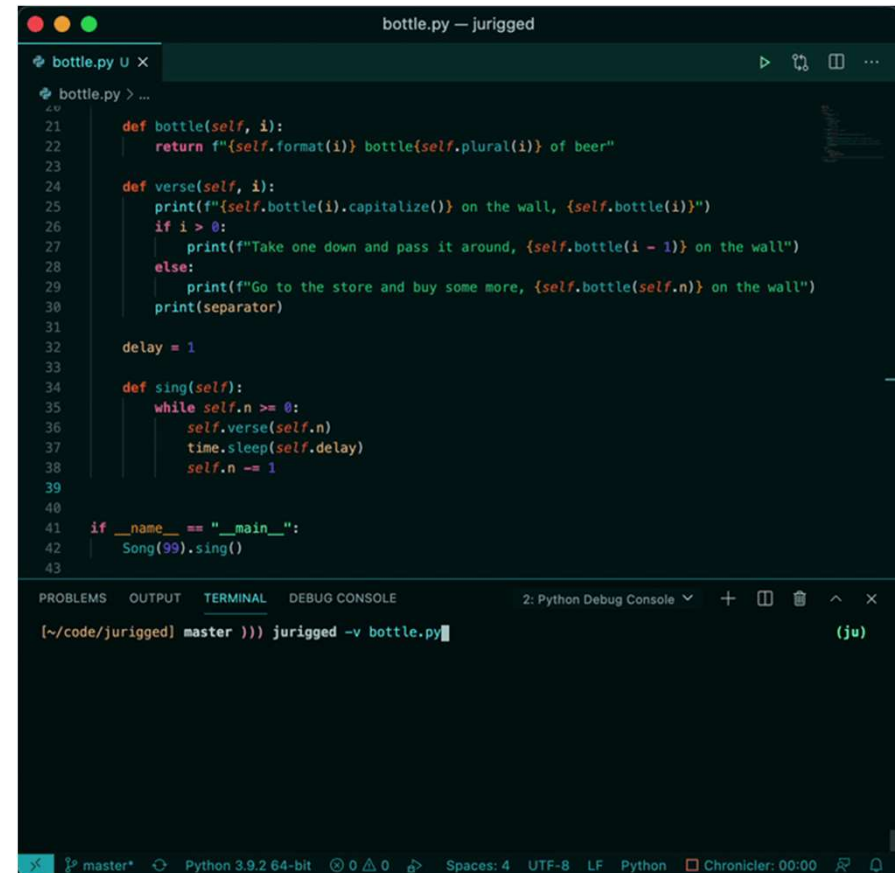
# Runtime APR

- Runtime APR
  - 프로그램이 실행 중에 crash 발생 → APR
  - Exception이 발생했을 때 종료해서는 안 됨



# Jurigged

- Python용 Live coding 도구
  - 프로그램 재실행 없이 코드 수정 반영
  - Crash 발생 시 프로그램 종료 없이 수정 가능
- <https://github.com/breuleux/jurigged>



```
def bottle(self, i):  
    return f"{self.format(i)} bottle{self.plural(i)} of beer"  
  
def verse(self, i):  
    print(f"{self.bottle(i).capitalize()} on the wall, {self.bottle(i)}")  
    if i > 0:  
        print(f"Take one down and pass it around, {self.bottle(i - 1)} on the wall")  
    else:  
        print(f"Go to the store and buy some more, {self.bottle(self.n)} on the wall")  
    print(separator)  
  
delay = 1  
  
def sing(self):  
    while self.n >= 0:  
        self.verse(self.n)  
        time.sleep(self.delay)  
        self.n -= 1  
  
if __name__ == "__main__":  
    Song(99).sing()
```

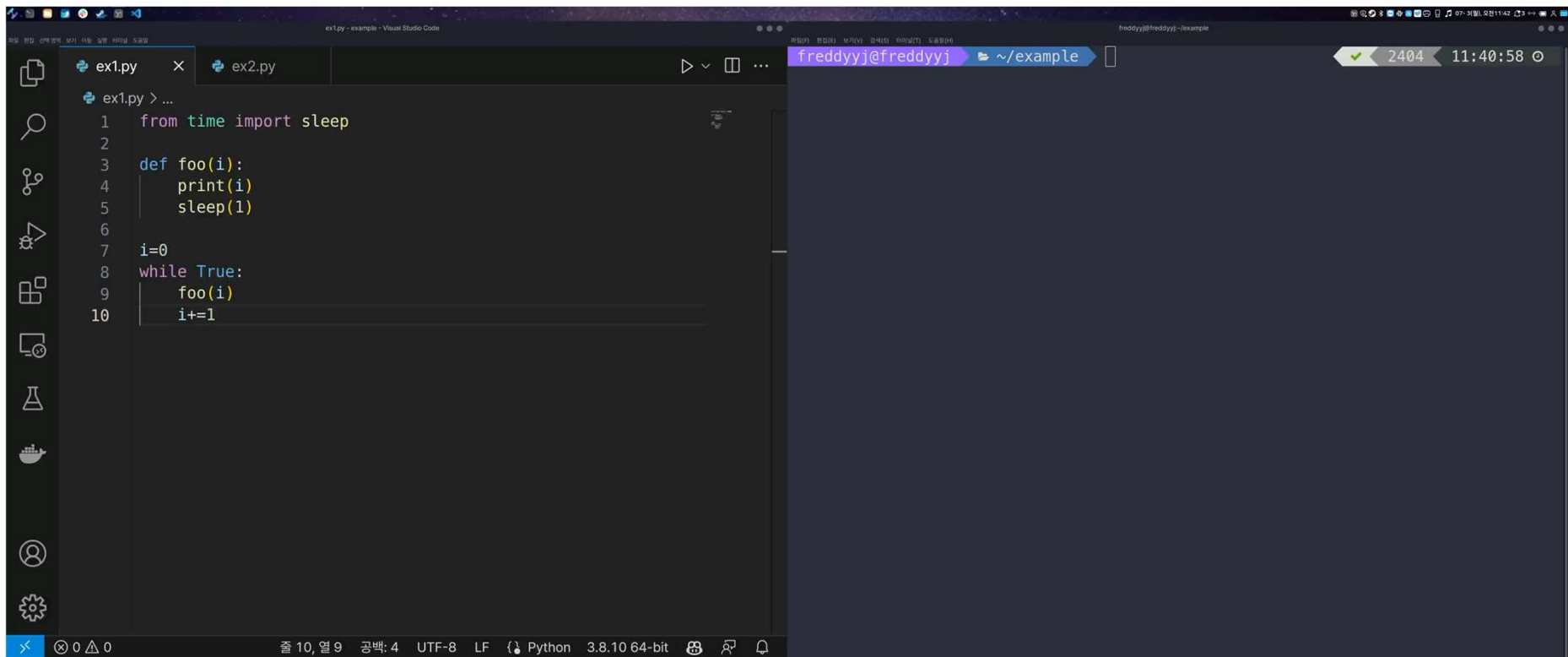
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE 2: Python Debug Console + - x

[~/code/jurigged] master ))) jurigged -v bottle.py (ju)

Python 3.9.2 64-bit Spaces: 4 UTF-8 LF Python Chronicer: 00:00

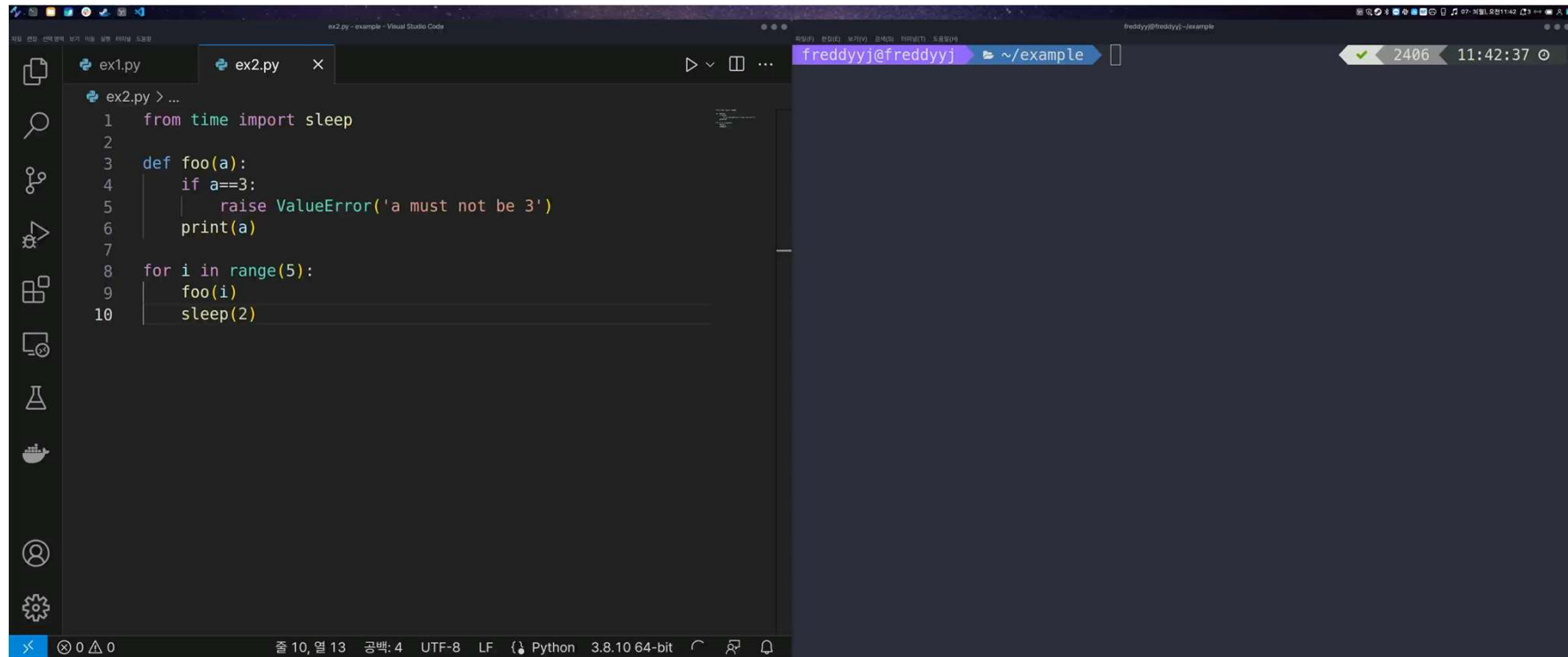
# Jurigged

- 프로그램 재실행 없이 코드 변경 내용 반영



# Jurigged

- --xloop: 프로그램에 Exception을 일으켰을 때 프로그램 일시정지
  - 코드가 수정되면 재개



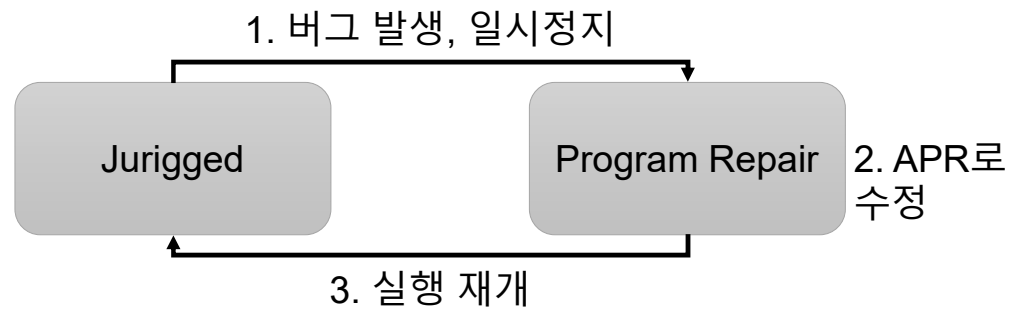
The screenshot displays a development environment with a code editor on the left and a terminal on the right. The code editor shows a Python file named `ex2.py` with the following content:

```
1 from time import sleep
2
3 def foo(a):
4     if a==3:
5         raise ValueError('a must not be 3')
6     print(a)
7
8 for i in range(5):
9     foo(i)
10    sleep(2)
```

The terminal window on the right shows the prompt `freddyj@freddyj: ~/example` and a green checkmark icon, indicating that the program is running successfully. The status bar at the bottom of the code editor shows the file encoding as UTF-8, the line ending as LF, and the Python version as 3.8.10 64-bit.

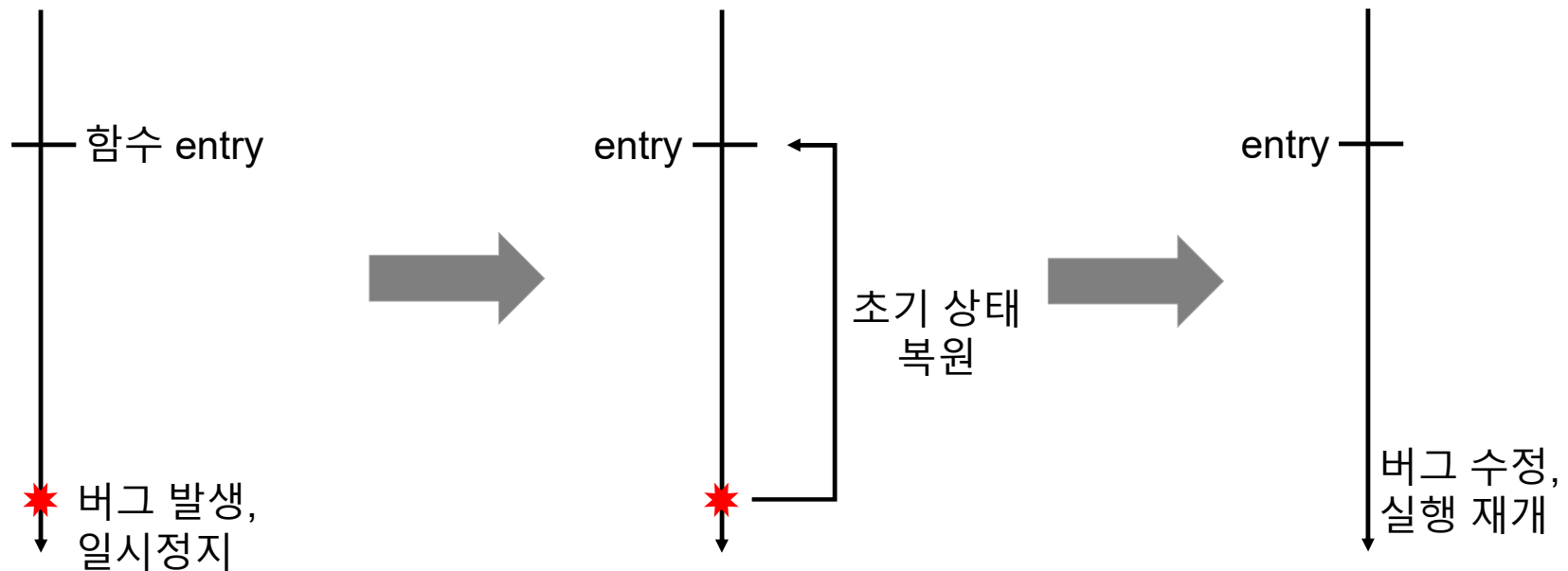
# Runtime APR

- Jurigged에서 Crash가 발생했을 때
  - 자동 프로그램 수정 (APR)
- 프로그램 종료 없이 런타임에 Hot-Patching



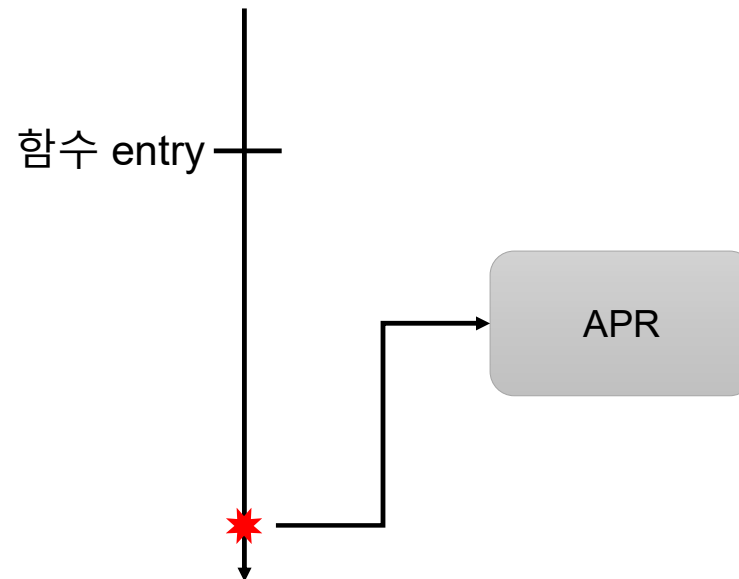
# Runtime APR

- Overview
  - 버그 관찰, 상태 복원, 버그 수정



# Runtime APR

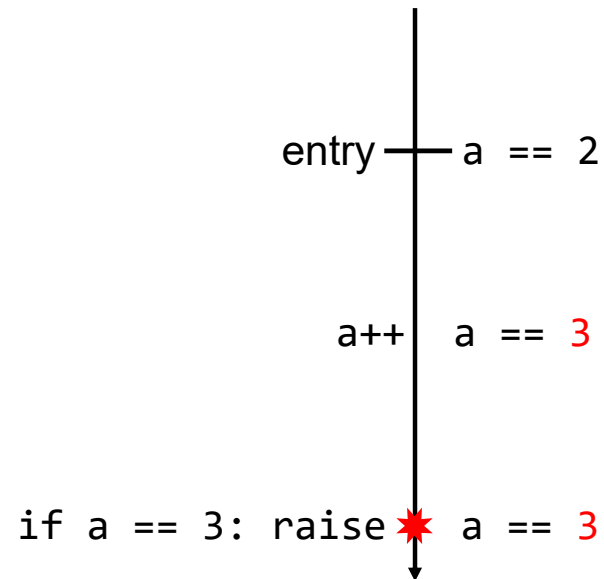
- 버그 관찰
  - Jurigged 활용
  - Crash 발생 시 APR 진행



# Runtime APR

- 초기 상태 복원 필요성
  - 함수 entry에서 a가 2일 때

```
a = 2
def inc():
    global a
    a = a + 1
    if a == 3:
        raise ValueError
    print(a)
```

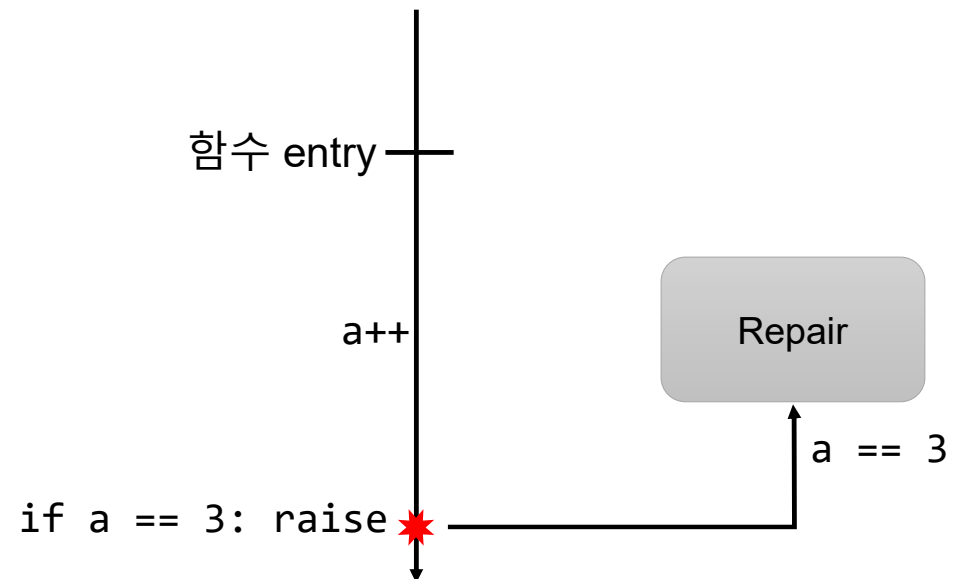




# Runtime APR

- 초기 상태 복원 필요성
  - ValueError 발생: repair로 이동

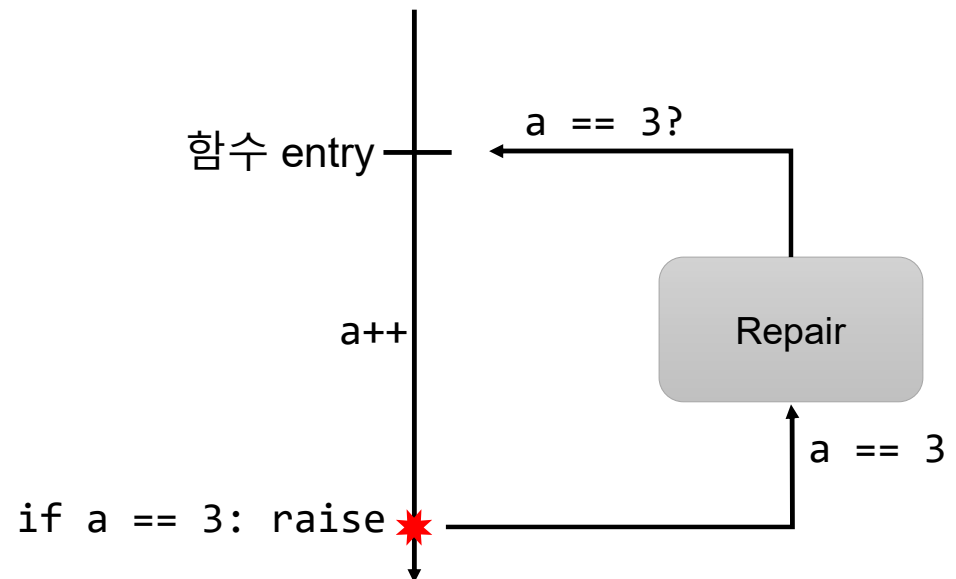
```
a = 2
def inc():
    global a
    a = a + 1
    if a == 3:
        raise ValueError
    print(a)
```



# Runtime APR

- 초기 상태 복원 필요성
  - Repair완료 후 함수 entry부터 다시 실행
    - a = 2부터 다시 실행해야 함

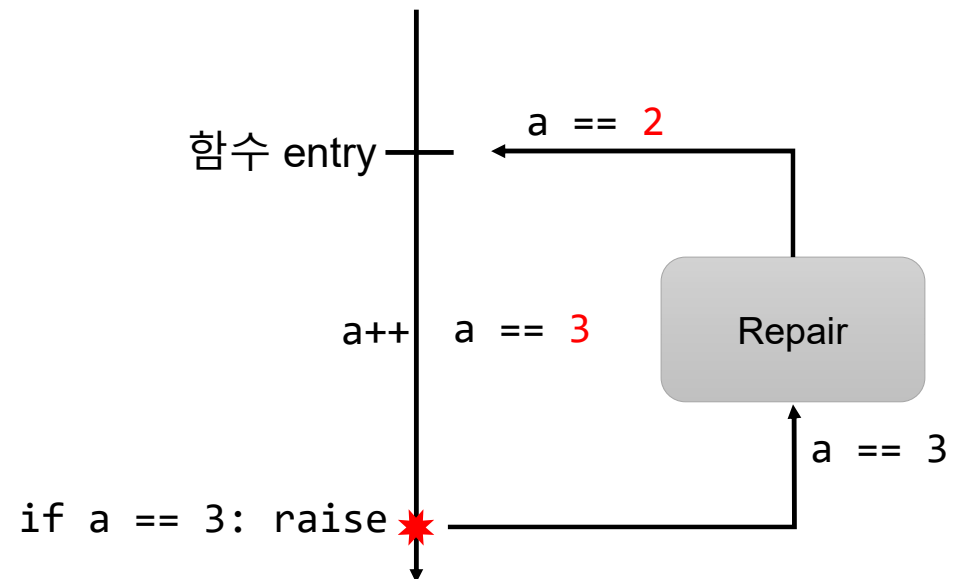
```
a = 2
def inc():
    global a
    a = a + 1
    if a == 3:
        raise ValueError
    print(a)
```



# Runtime APR

- 초기 상태 복원 필요성
  - 초기 상태 복원해서 함수 다시 실행

```
a = 2
def inc():
    global a
    a = a + 1
    if a == 3:
        raise ValueError
    print(a)
```



# Runtime APR

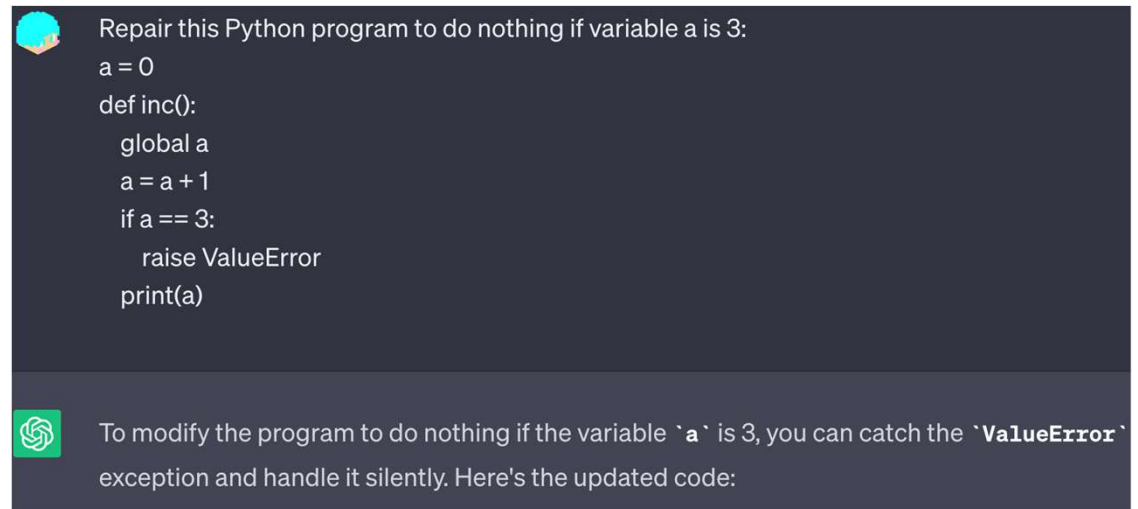
- 초기 상태 복원
  - Concolic 실행으로 함수 entry에서의 상태 복원
  - Issue: (해결중)
    - C-type 객체, C 라이브러리
    - 복잡한 Heap 객체
    - ...

# Runtime APR

- 버그 수정 (계획)
  - 기존 APR의 접근법 활용

```
+ if <condition>:  
+     <body>  
if a == 3:  
    raise ValueError
```

## 1. 템플릿 기반



The screenshot shows a chat interface with a user prompt and an AI response. The user prompt asks to repair a Python program to do nothing if variable 'a' is 3. The AI response provides the updated code, which catches the 'ValueError' exception and handles it silently.

```
Repair this Python program to do nothing if variable a is 3:  
a = 0  
def inc():  
    global a  
    a = a + 1  
    if a == 3:  
        raise ValueError  
    print(a)
```

To modify the program to do nothing if the variable `a` is 3, you can catch the `ValueError` exception and handle it silently. Here's the updated code:

## 2. 학습 기반

# Challenge

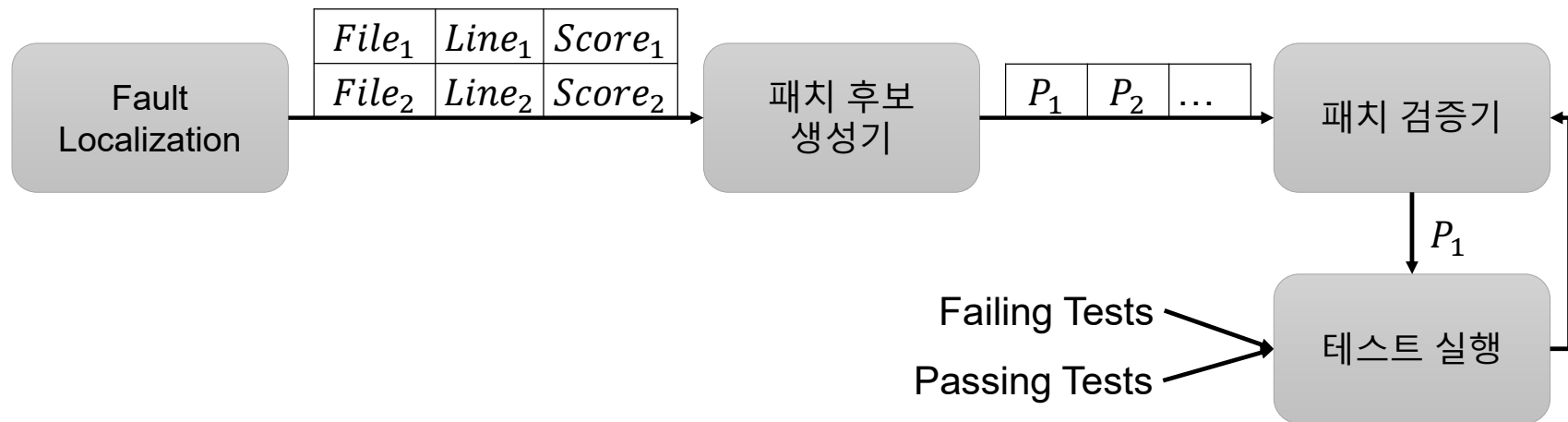
- Hot-patching: Crash 수정 시간
  - 완벽하지 않아도 **빠르게** crash 수정
  - 1시간 이내
- 기존 APR들은 성공률에 집중

# 효율적인 패칭

Automated Program Repair from Fuzzing Perspective (ISSTA'23)

김영재, 한승헌, Khamit Askar, 이주용

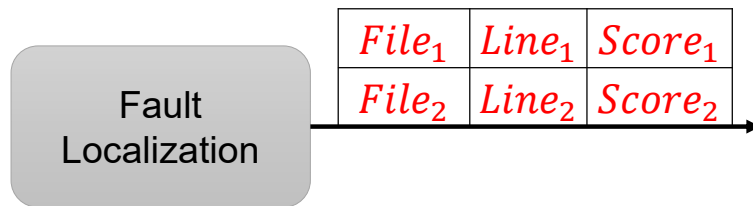
# Automated Program Repair (APR)



- Generation & Validation (G&V) APR
  - 패치 후보들 생성 후 하나씩 검증
  - 가장 일반적인 APR

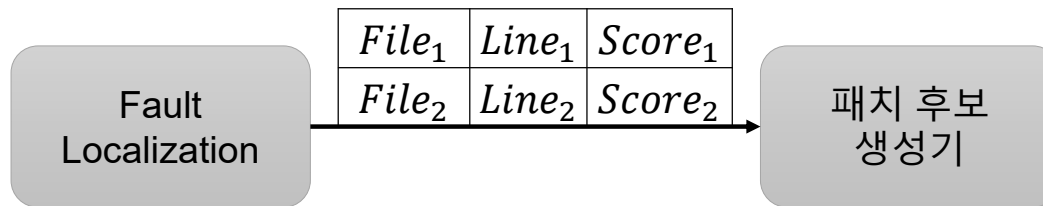


# Automated Program Repair (APR)



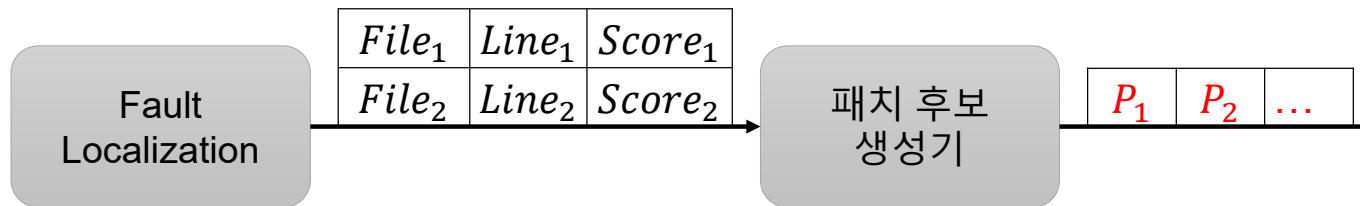
- Fault Localization (FL)
  - Spectrum-based FL (SBFL)
  - 출력: File, Line과 Score들의 목록

# Automated Program Repair (APR)



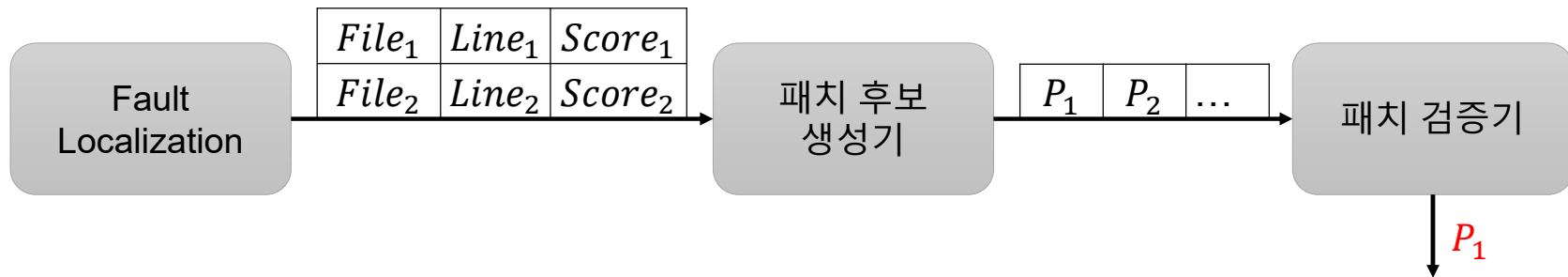
- 패치 후보 생성 (Generation)
  - 템플릿, 언어 모델, ...

# Automated Program Repair (APR)



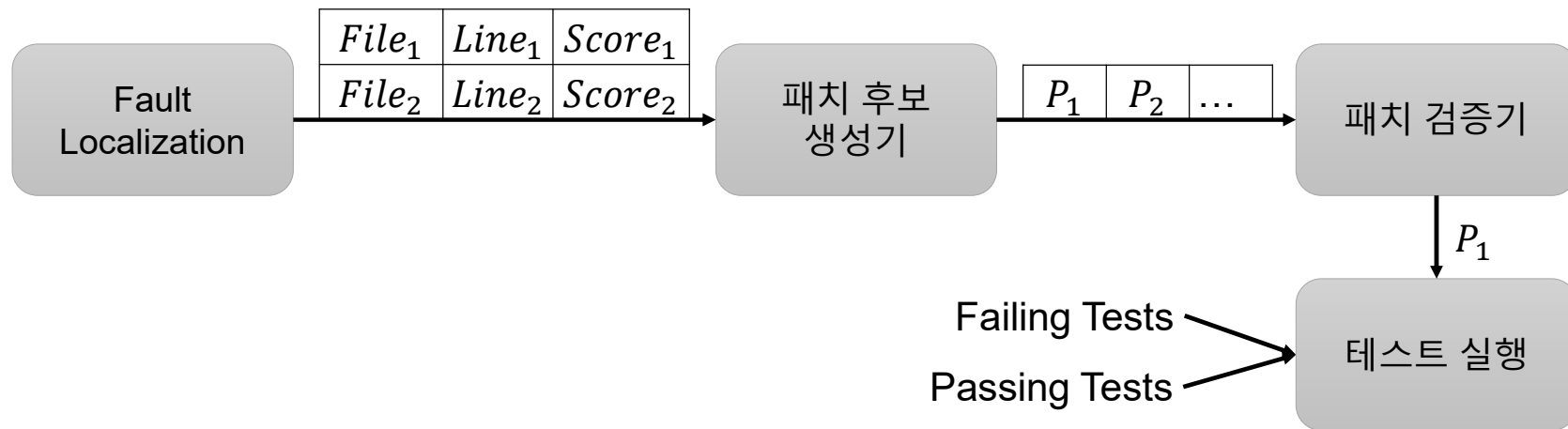
- 패치 후보 생성 (Generation)
  - 출력: **정렬된** 패치 후보 리스트
  - 후보가 왼쪽에 있을수록 패치일 가능성 ↑

# Automated Program Repair (APR)



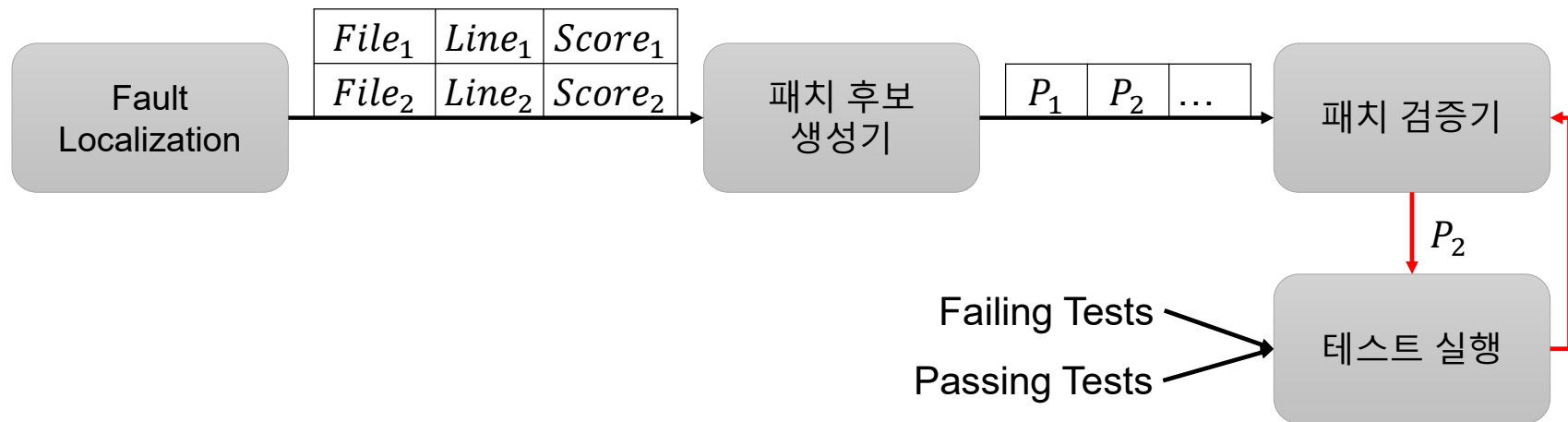
- 패치 후보 검증 (Validation)
  - 패치 후보 리스트에서 하나씩 검증
  - 탐색 알고리즘: **항상 맨 왼쪽** 후보 탐색

# Automated Program Repair (APR)



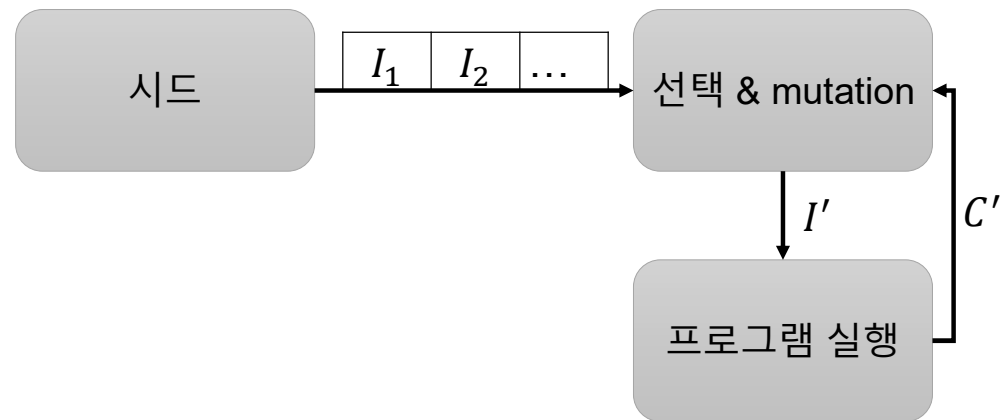
- 패치 후보 검증 (Validation)
  - 패치 후보 적용 후 테스트들 실행
  - Valid Patch: Failing과 Passing Tests 모두 pass

# Automated Program Repair (APR)



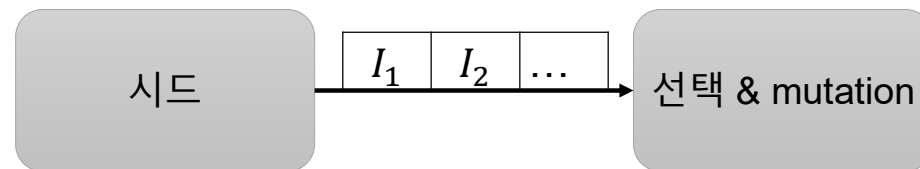
- 패치 후보 검증 (Validation)
  - 다음 후보 시도
  - 종료 조건 도달까지 계속 반복

# Fuzzing



- Fuzzing
  - 다양한 입력들을 생성해 프로그램 테스트, 버그 탐지

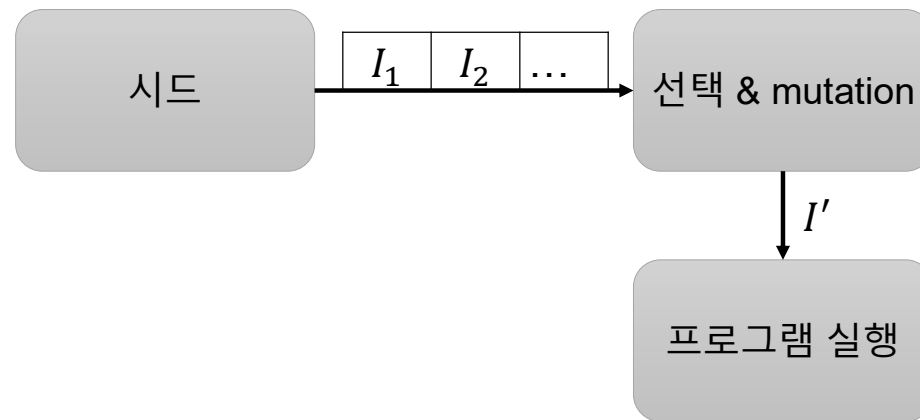
# Fuzzing



- 입력 선택 & mutation
  - 시드에서 랜덤하게 입력 선택
  - Mutation

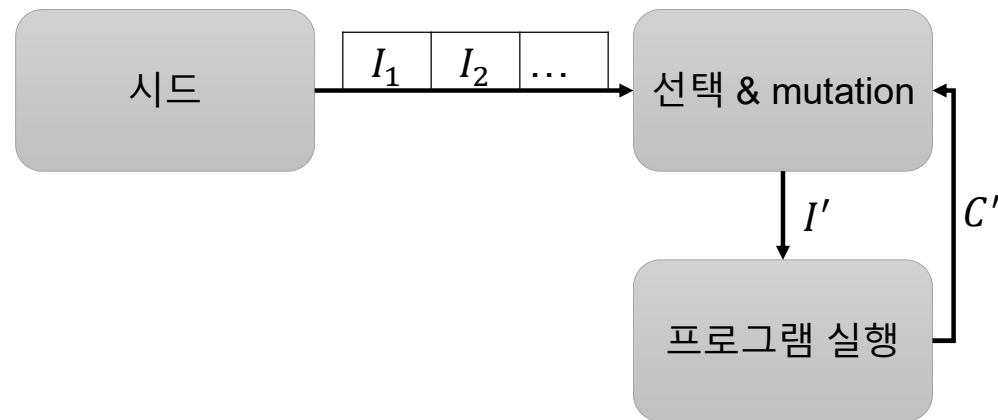


# Fuzzing



- 프로그램 실행
  - 생성한 입력으로 프로그램 실행

# Fuzzing

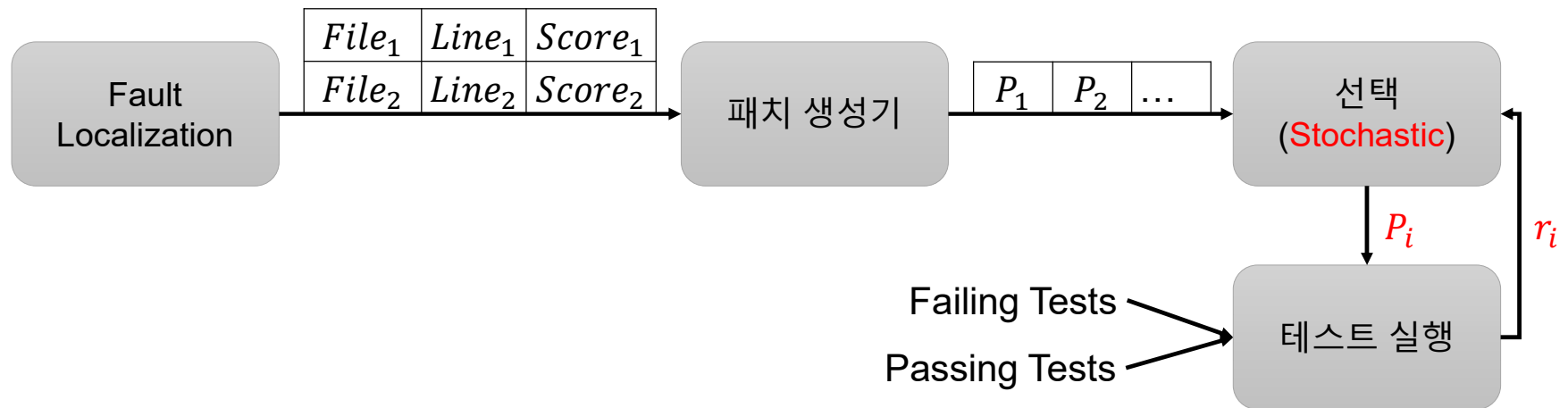


- 입력 corpus 업데이트
  - 실행 결과를 이용해 입력 corpus 업데이트
  - 커버리지, 출력, ...

# APR vs Fuzzing

	APR	Fuzzing
목적	버그 수정	버그 탐지
탐색 공간	패치 후보	프로그램 입력
탐색 목표	Valid Patch	버그를 유발하는 입력
알고리즘	대부분 Deterministic	대부분 Stochastic
성능	TBar: 101/395 (25.57%)	850개 프로젝트, 28,000 버그 탐지

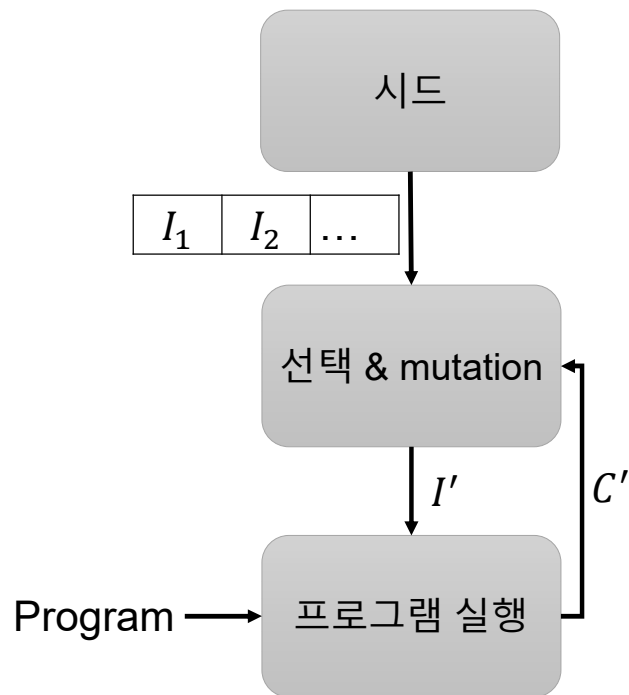
# APR from Fuzzing Perspective



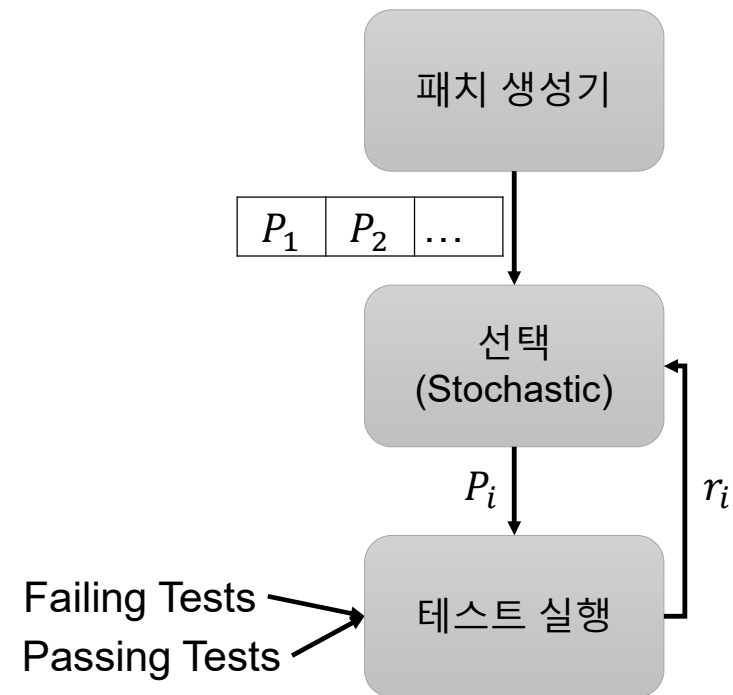
- Fuzzing 알고리즘을 활용한 APR
  - 패치 탐색 시 Stochastic하게 후보 선택
  - 실행 결과를 이용해 탐색 공간 업데이트

# APR from Fuzzing Perspective

- Fuzzing

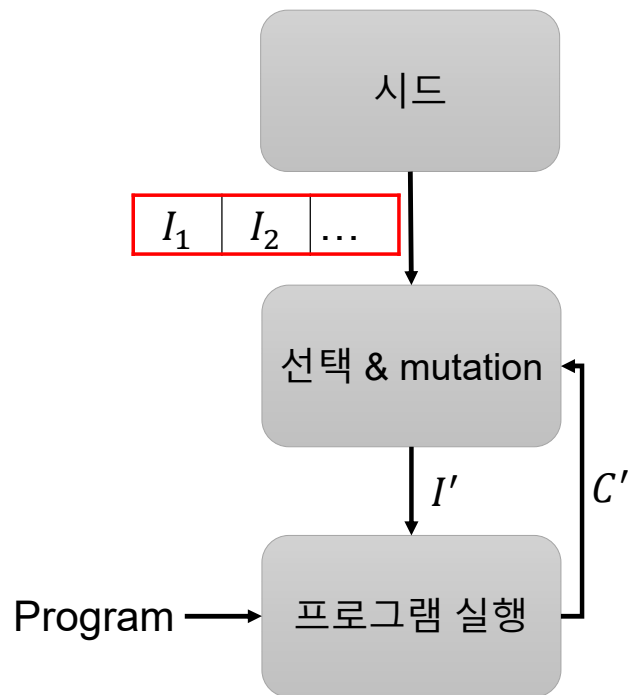


- Our algorithm

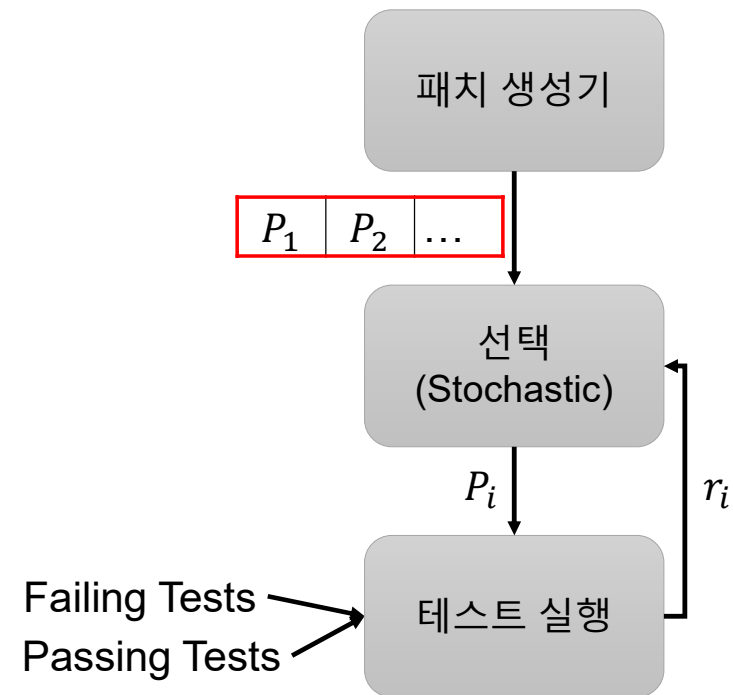


# Search Space

- Fuzzing

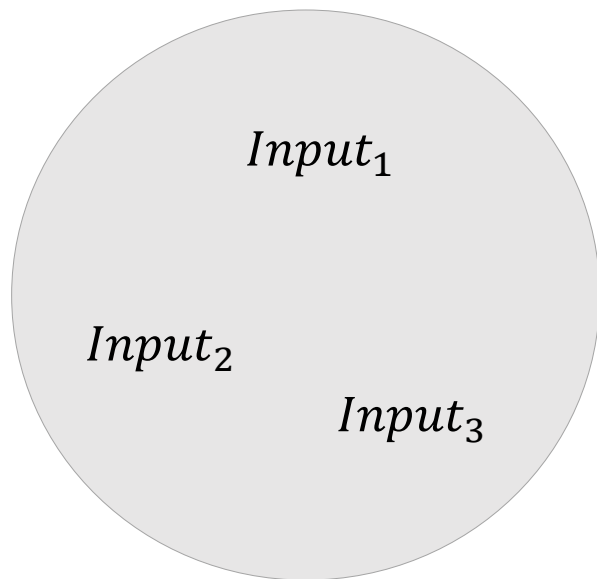


- Our algorithm

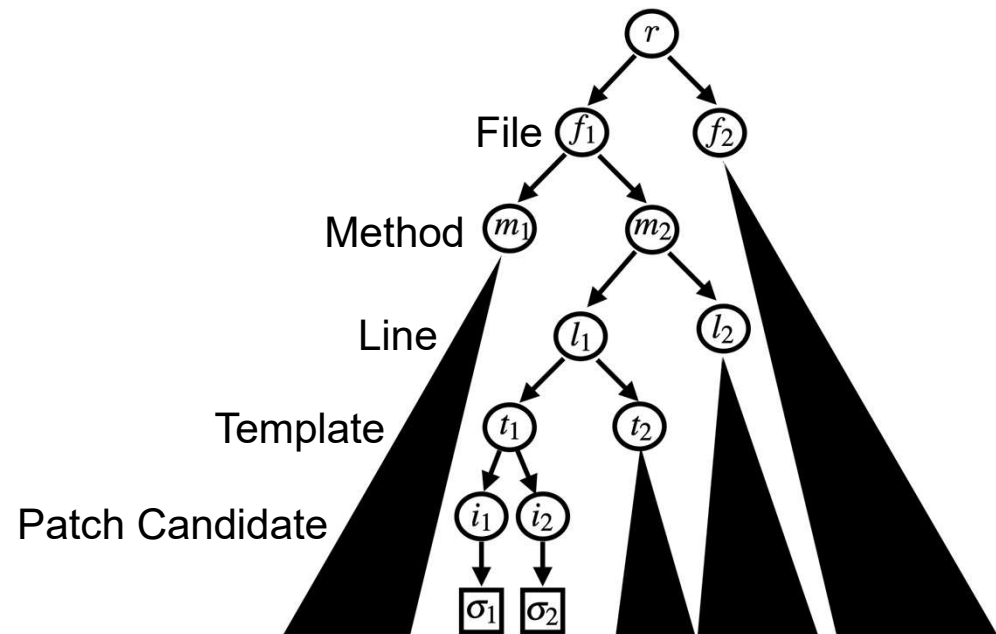


# Search Space

- Search space of Fuzzing
  - 입력 Corpus

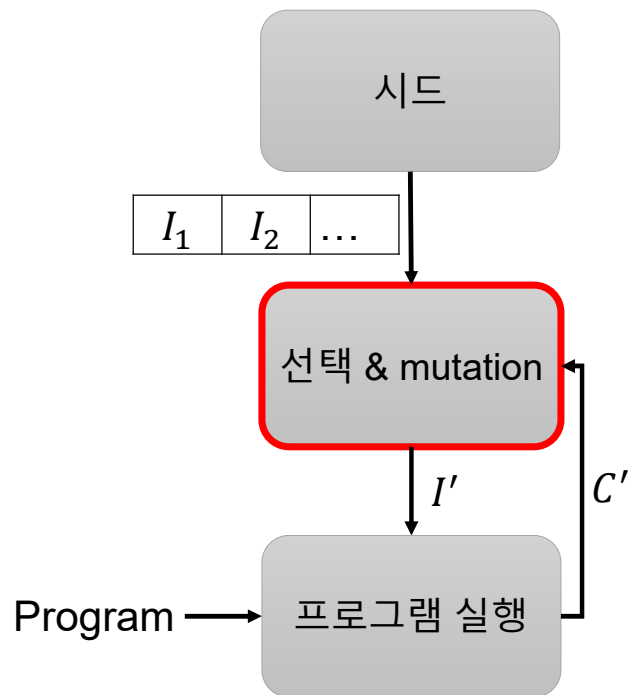


- Search space of Our algorithm
  - Patch Tree

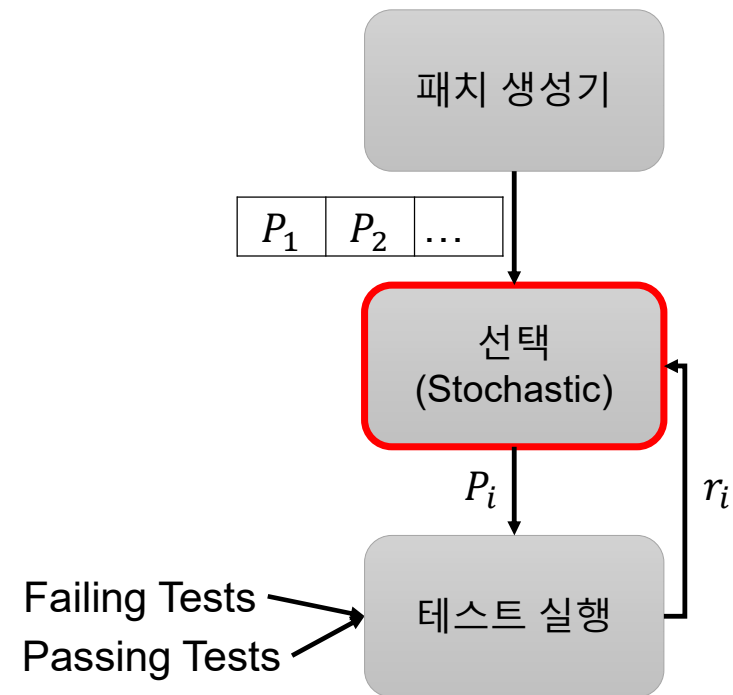


# Input/Patch Selection

- Fuzzing



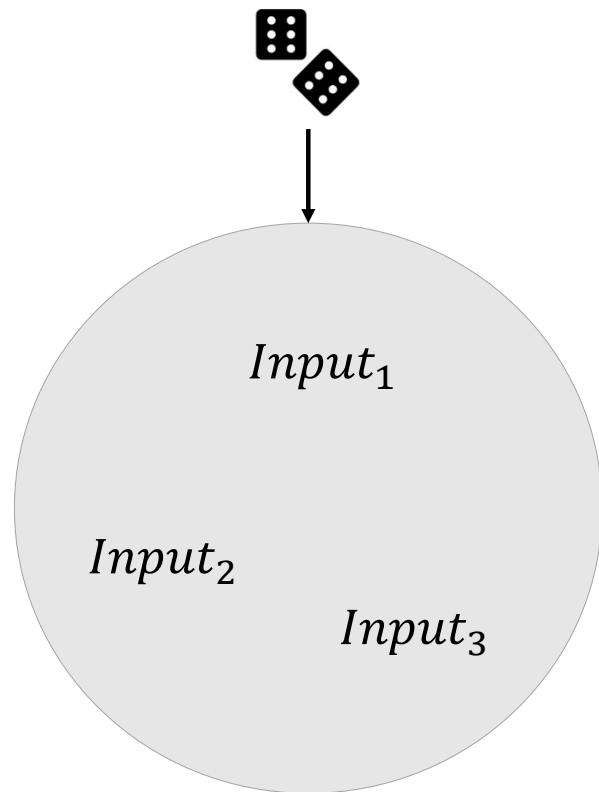
- Our algorithm



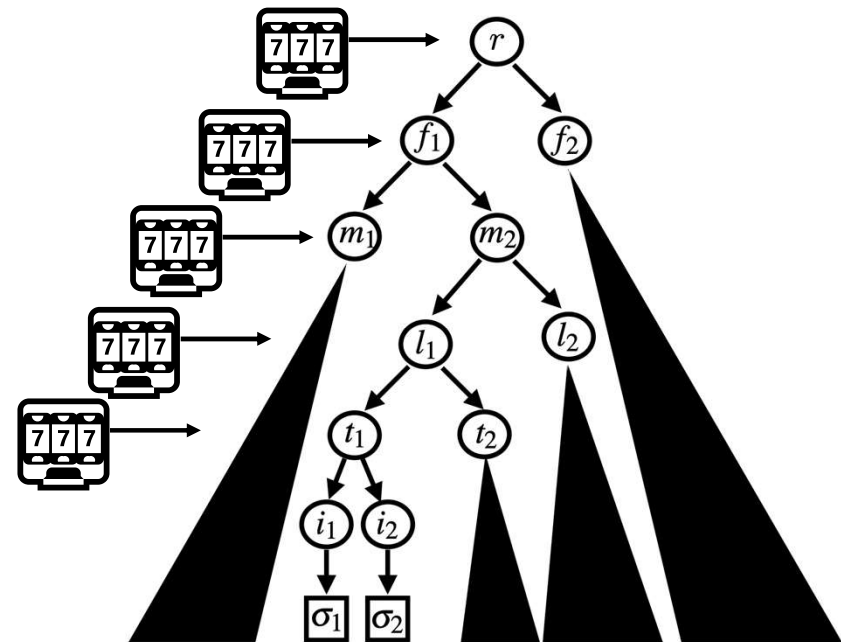


# Input/Patch Selection

- 입력 선택 in Fuzzing
  - 입력 corpus에서 랜덤하게

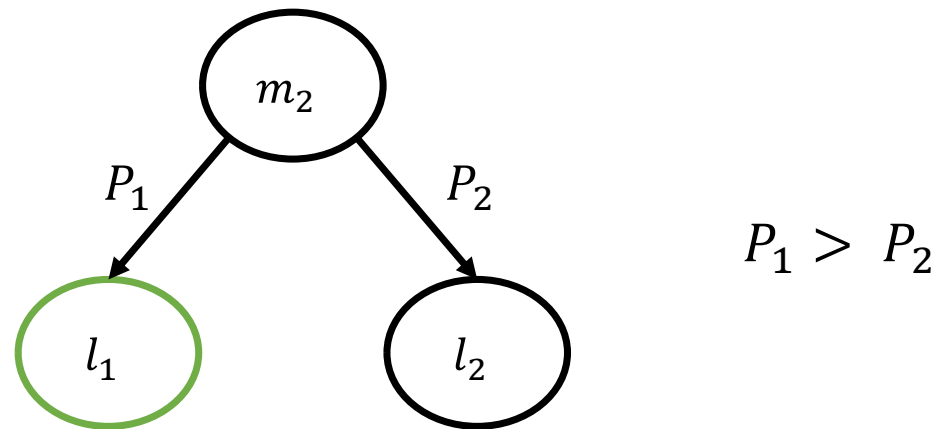


- 패치 후보 선택 in Our algorithm
  - 트리의 각 레벨에서 랜덤하게



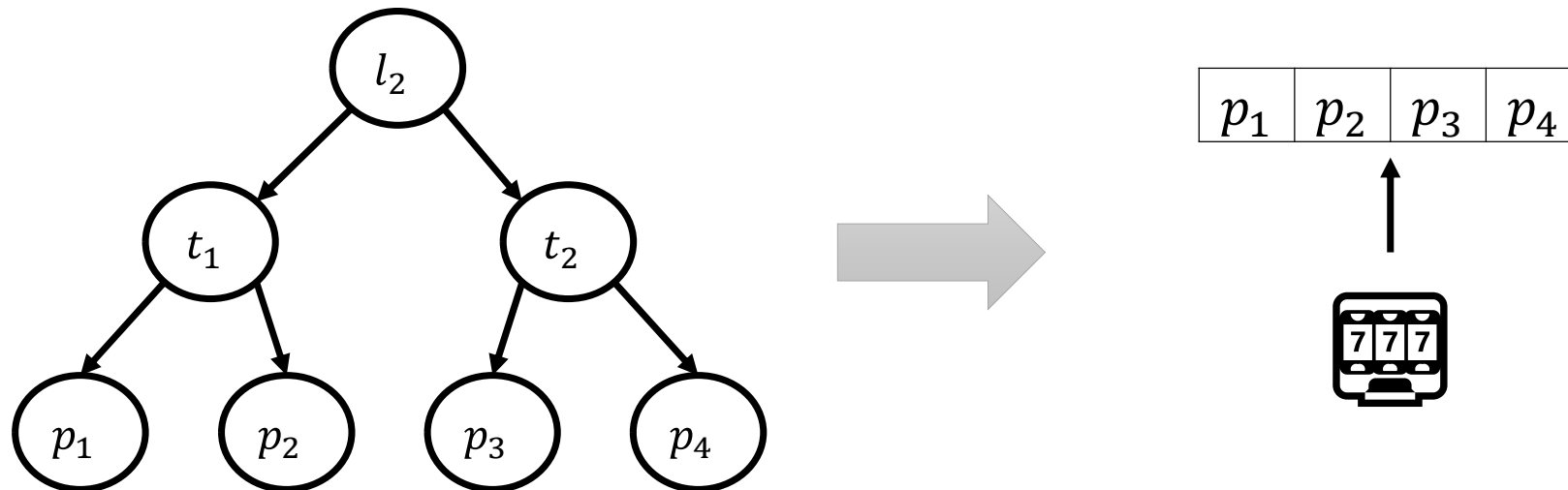
# Patch Selection

- 트리의 각 레벨에서 랜덤하게
  - Interesting Patch: Failing Tests 중 하나 이상 Pass
  - 트리의 노드에 Interesting Patch가 있을 때
    - Interesting Patch가 발견된 노드에 높은 확률



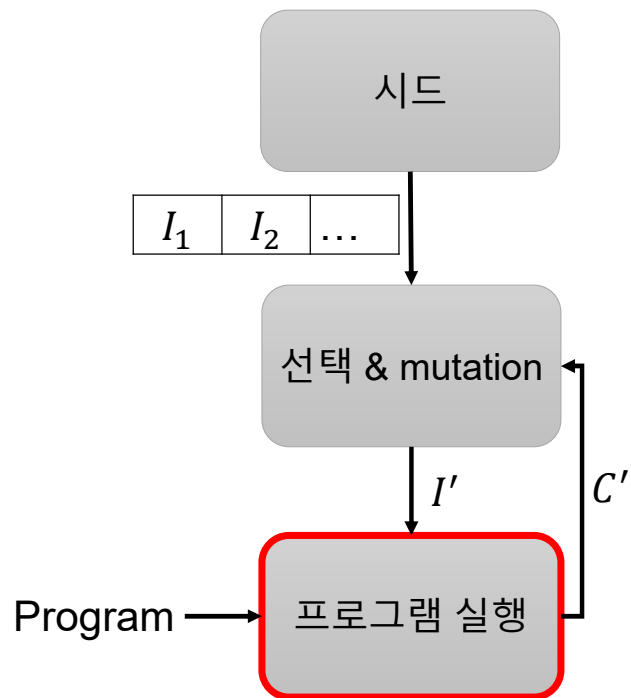
# Patch Selection

- 트리의 각 레벨에서 랜덤하게
  - Interesting Patch: Failing Tests 중 하나 이상 Pass
  - 트리의 노드에 Interesting Patch가 없을 때
    - Subtree에서 FL score가 가장 높은 패치 후보들 중 선택

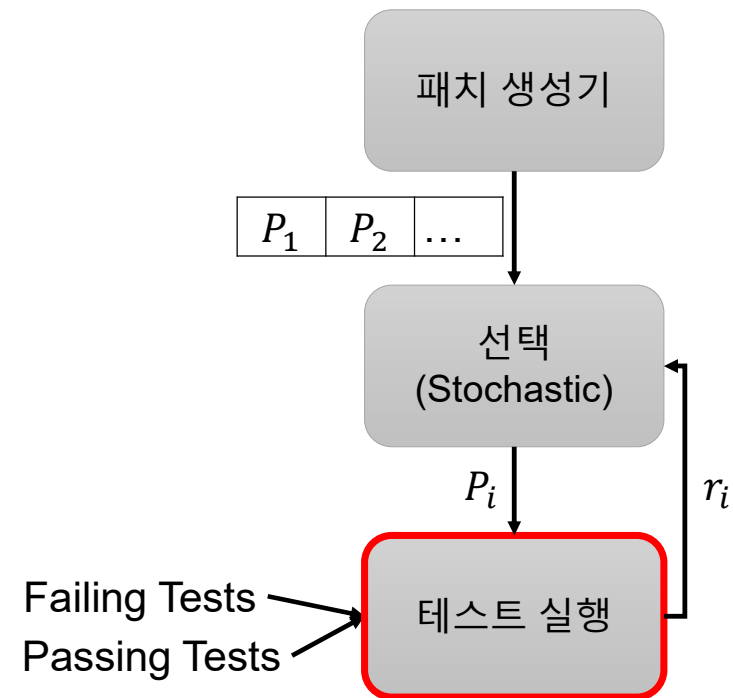


# Program/Test Execution

- Fuzzing

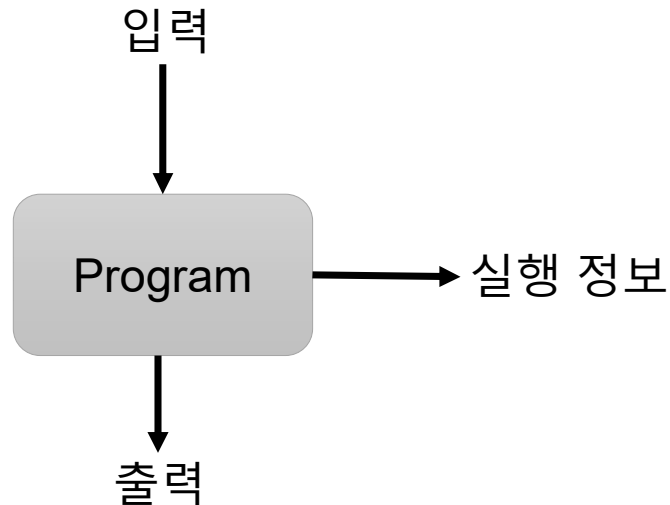


- Our algorithm

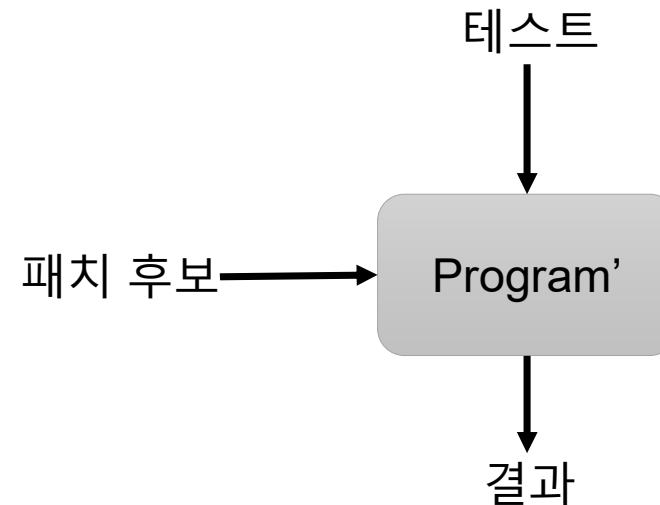


# Program/Test Execution

- 프로그램 실행 in Fuzzing
  - 선택/mutation한 입력으로 실행
  - 출력: 실행 정보
    - 커버리지, ...

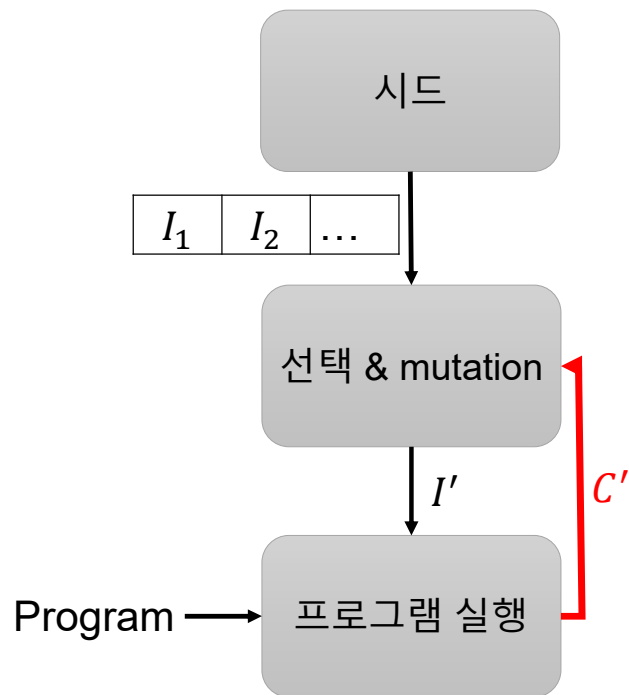


- 테스트 실행 in Our algorithm
  - 선택한 패치 후보로 실행
  - 결과: Pass/Fail

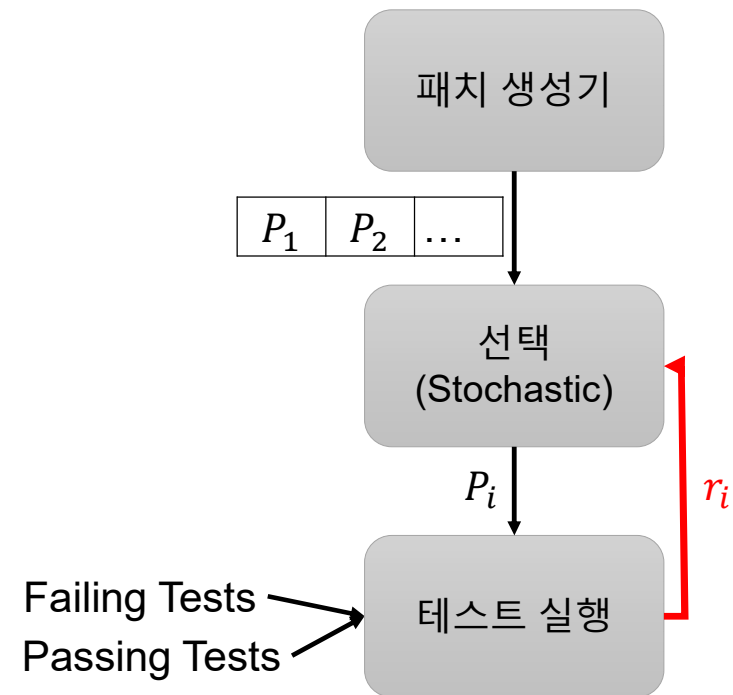


# Update Search Space

- Fuzzing

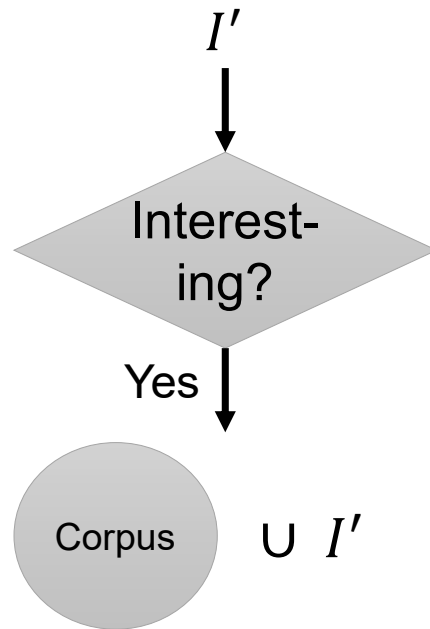


- Our algorithm

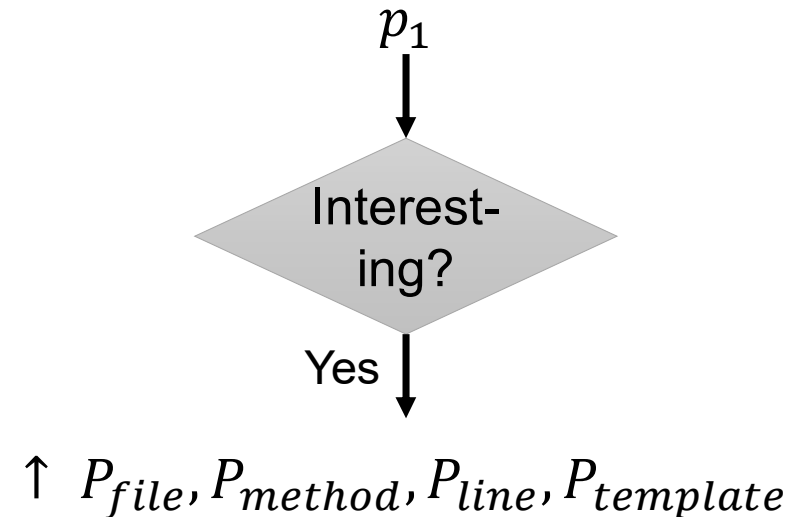


# Update Search Space

- 입력 corpus 업데이트 in Fuzzing
  - Interesting Input이면 입력 corpus에 추가

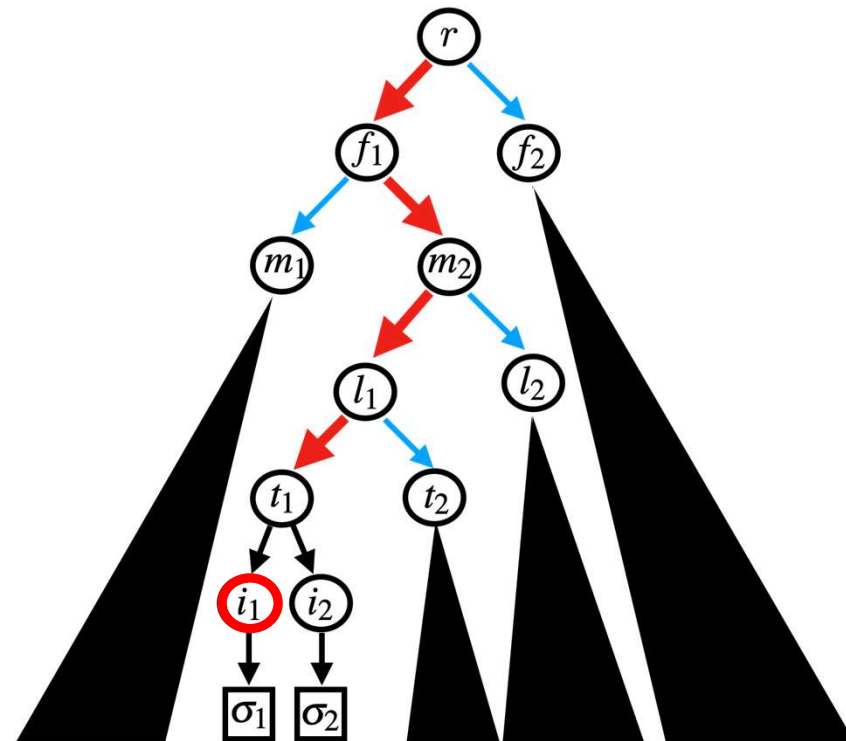


- 패치 트리 업데이트 in Our Algorithm
  - Interesting Patch면 패치 트리의 파일, 메소드, 라인, 템플릿 노드의 확률 증가



# Update Search Space

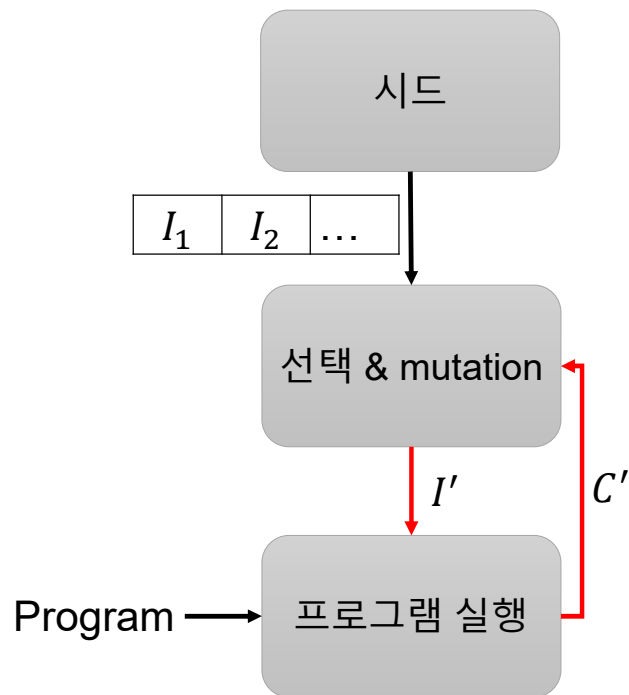
- 예:  $i_1$  이 Interesting Patch였다면
  - $f_1, m_2, l_1, t_1$ 의 확률 증가 (→)
  - 다른 노드도 확률 존재 (→)



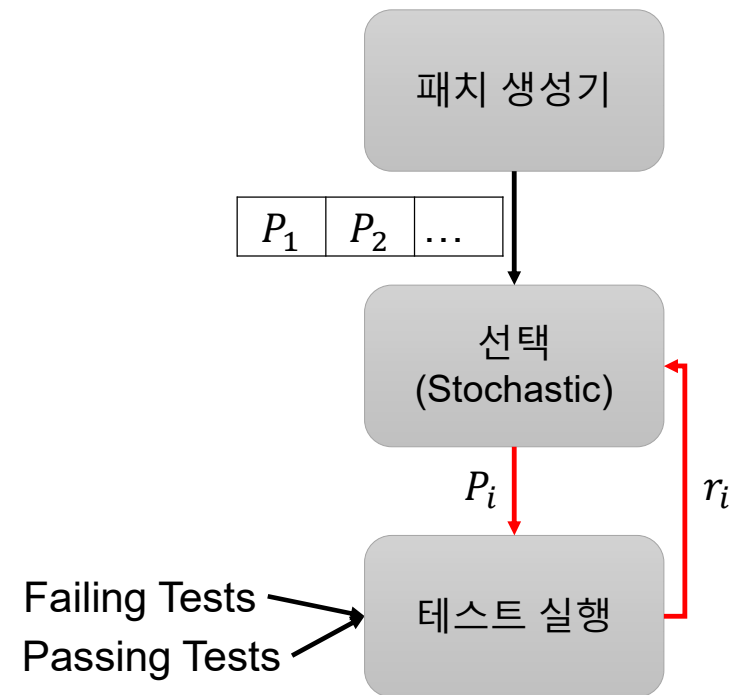


# Termination Condition

- Fuzzing
  - Time-out



- Our algorithm
  - Time-out



# Our Algorithm

- 패치 후보 효율적 탐색
  - 패치를 더 빨리, 더 많이 찾는 알고리즘
- Fuzzing의 알고리즘
  - Stochastic하게 패치 후보 탐색
  - 테스트 실행 후 탐색 공간 업데이트
    - Interesting Patch 주변 후보들의 확률 증가
- Our Algorithm: Casino

# Evaluation

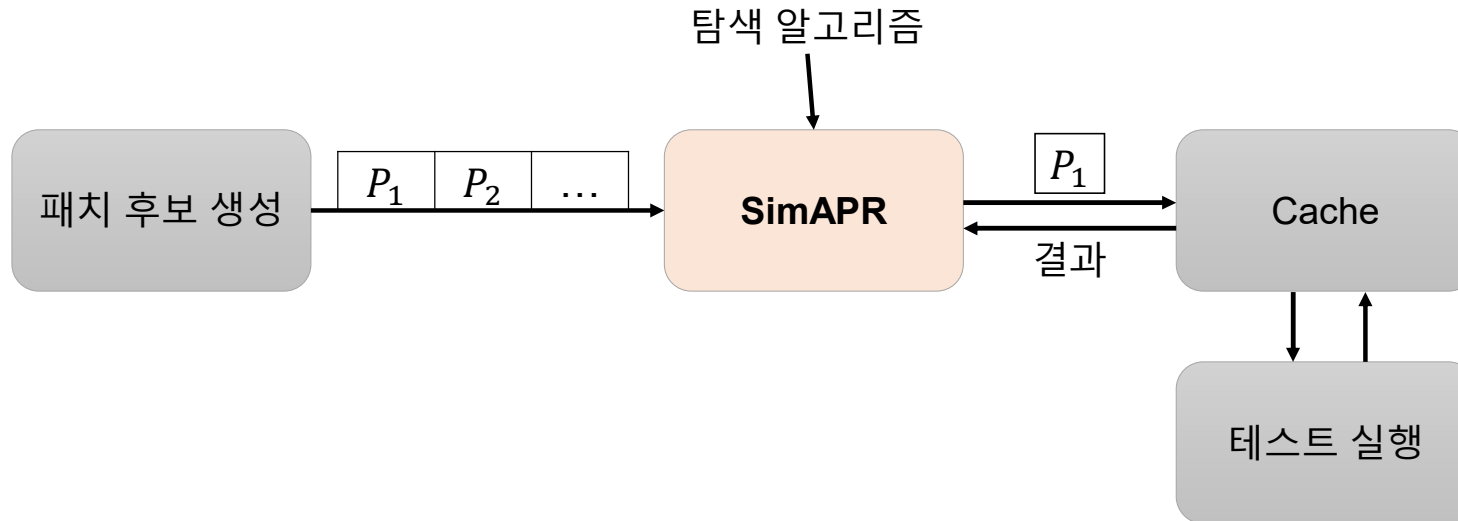
- 벤치마크: Defects4j v1.2.0
  - 6개 Java 오픈소스 소프트웨어, 395개 버그
- 6개 APR도구
  - 템플릿 기반: TBar, Avatar, kPar, Fixminer
  - 학습 기반: AlphaRepair, Recoder
- 4개 탐색 알고리즘
  - 원래 APR도구의 알고리즘, GenProg
  - SeAPR: ICSE'22에 발표된 패치 탐색 알고리즘
  - Casino: Our Algorithm

# Evaluation

- 원래 알고리즘, SeAPR: 각 1번씩
- Casino, GenProg: 각 50번씩
- Time-out: 각 5시간
- Challenge: 실험 시간
  - 5시간 x (50 + 50 + 1 + 1) x 6개 APR x 395개 버그 =  
1,208,700시간 ( $\approx$  137.98년)

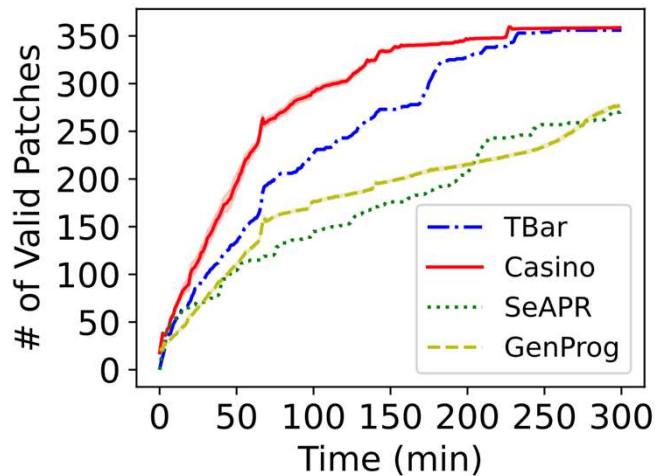
# Implementation

- SimAPR
  - 패치 탐색 알고리즘을 시뮬레이션
  - 각 패치 후보의 테스트 실행 결과 Cache
    - 다음 알고리즘에서 cache된 결과 활용, 테스트 실행 생략
  - <https://github.com/UNIST-LOFT/SimAPR>

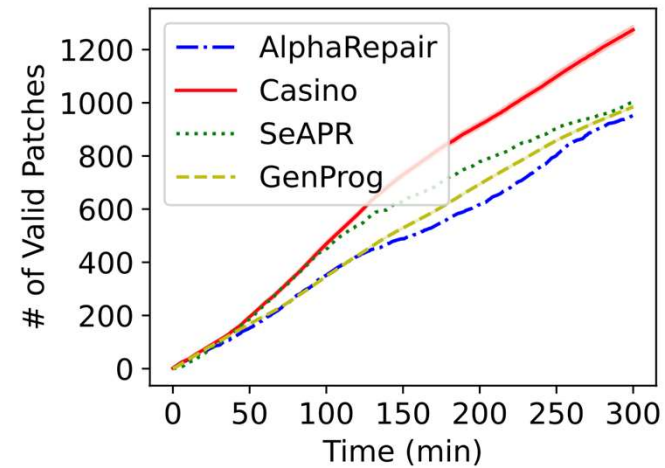


# Results

- Valid Patch 탐색 성능
  - Valid Patch: Failing과 Passing Tests 모두 pass
- Casino, GenProg: 95% 신뢰구간



TBar의 결과



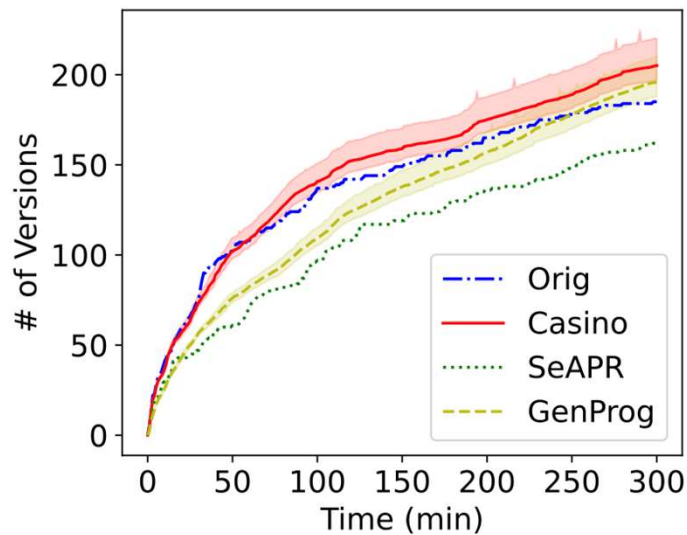
AlphaRepair의 결과

# Results

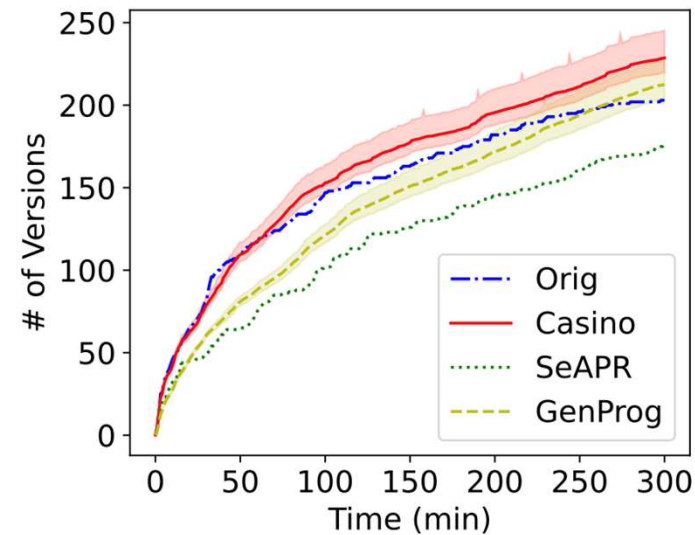
- 정답 패치 탐색 성능
  - 정답 패치: 개발자가 제공한 패치와 의미가 같은 패치
- 2,775 Valid Patch들
  - 직접 눈으로 정답인지 확인 불가
- DiffTGen: Differential Testing 도구
  - 개발자 패치와 semantic 비교
  - Acceptable Patch: 개발자 패치와 semantic이 같은 패치

# Results

- Acceptable Patch 탐색 성능
- Valid Patch에서 N등 안에 Acceptable Patch가 있는 버그의 개수



**(a) Top-1**



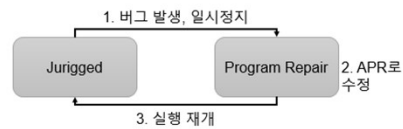
**(b) Top-5**



# Conclusion

## Runtime APR

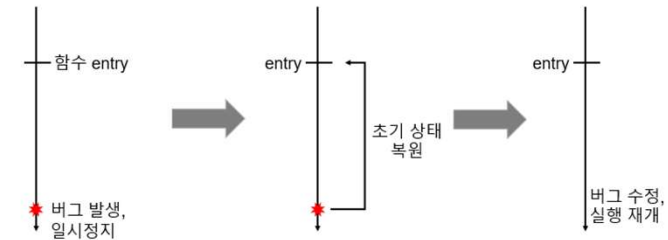
- Jurigged에서 Crash가 발생했을 때
  - 자동 프로그램 수정 (APR)
- 프로그램 종료 없이 런타임에 Hot-Patching



13

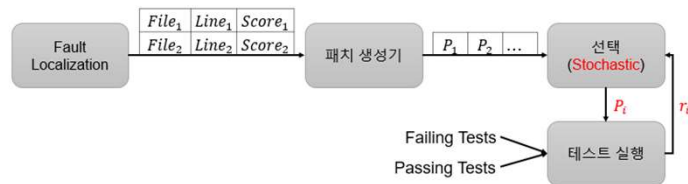
## Runtime APR

- Overview
  - 버그 관찰, 상태 복원, 버그 수정



14

## APR from Fuzzing Perspective

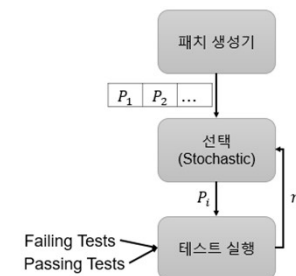
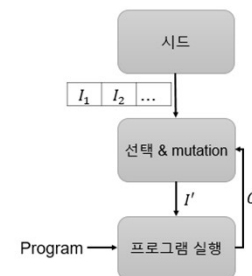


- Fuzzing 알고리즘을 활용한 APR
  - 패치 탐색 시 Stochastic하게 후보 선택
  - 실행 결과를 이용해 탐색 공간 업데이트

36

## APR from Fuzzing Perspective

- Fuzzing
- Our algorithm



37