

# 테스트 만족 가능성 분석을 통한 패치 공간 프루닝 기술

홍성준, 이준희, 오학주

고려대학교 소프트웨어 분석 연구실

2023년 7월 6일 @ ERC 여름 워크샵

# Closure 자바스크립트 컴파일러 버그 사례

## no warnings when @private prop is redeclared

Nicholas.J.Santos@2010-09-27T06:17:40.000Z

```
<b>What steps will reproduce the problem?</b>
/** @constructor */ function Foo() { /** @private */ this.x_ = 3; }

then, in a separate file:
/** @constructor
 * @extends {Foo} */ function SubFoo() { /** @private */ this.x_ = 3; }

then, compile with --jscomp_error=visibility

Expected: You should get an error.
Actual: No error.
```

실제로는 컴파일 오류가 있는 JavaScript 코드인데,  
Closure는 오류를 안 냅니다.



```
test(“/** @constructor */ function Foo() { /** @private */this.bar_ =
3; }/**@constructor \n * @extends {Foo} */ function SubFoo() { /** @private
*/this.bar_ = 3; };”, PRIVATE_OVERRIDE);
```

# Closure 자바스크립트 컴파일러 버그 사례

**Emit a warning when a private property overrides another private property, with both defined in the ctor.**  
Fixes issue 254

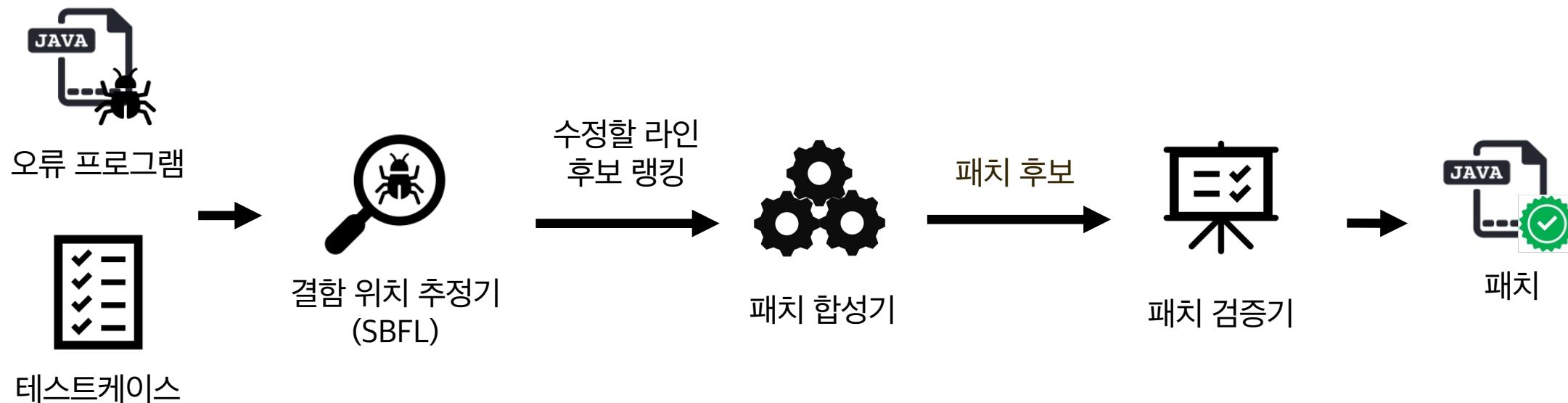
master  
webpack-v20180702 ... closure-compiler-maven-v20140407

nicksantos@google.com committed on May 13, 2011

오류 수정까지 총 8개월 소요

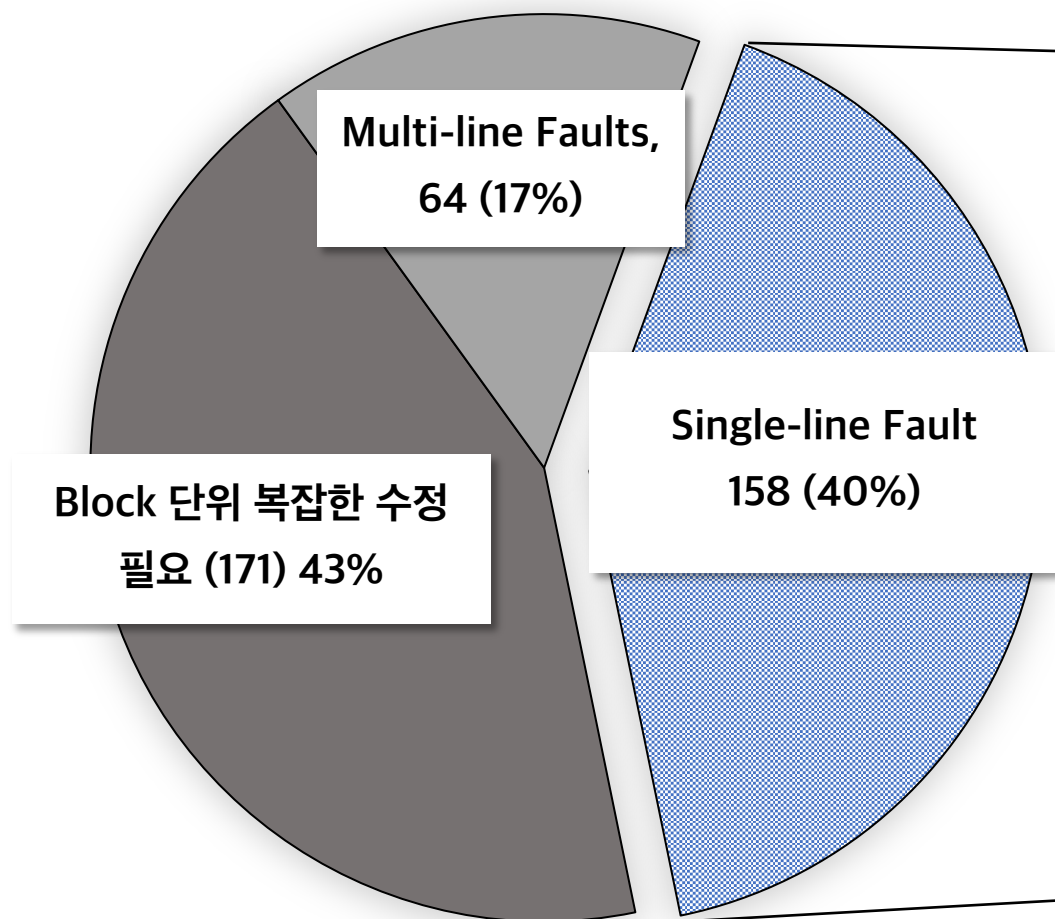
```
if (objectType != null) {  
- boolean isOverride = t.inGlobalScope() &&  
+ boolean isOverride = parent.getJSDocInfo() != null &&  
  parent.getType() == Token.ASSIGN &&  
  parent.getFirstChild() == getprop;  
}
```

# 테스트 기반 오류 자동 수정 기술



# 현재 수준: 테스트를 통과하는 패치 합성조차 어려움

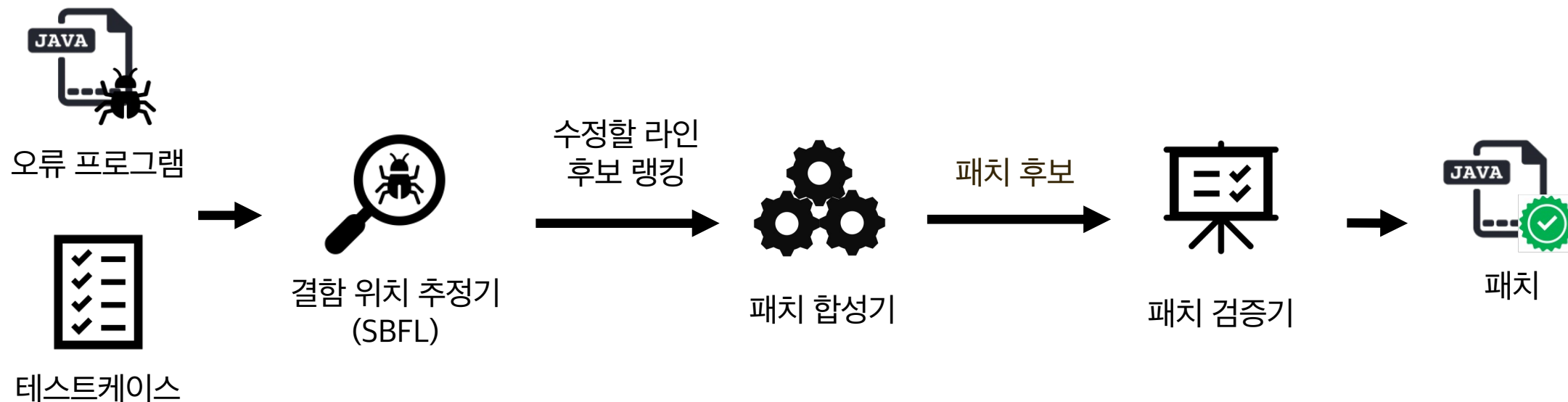
Defects4J 벤치마크 393개 버그



TBar [FSE'19] 패치 결과 분석

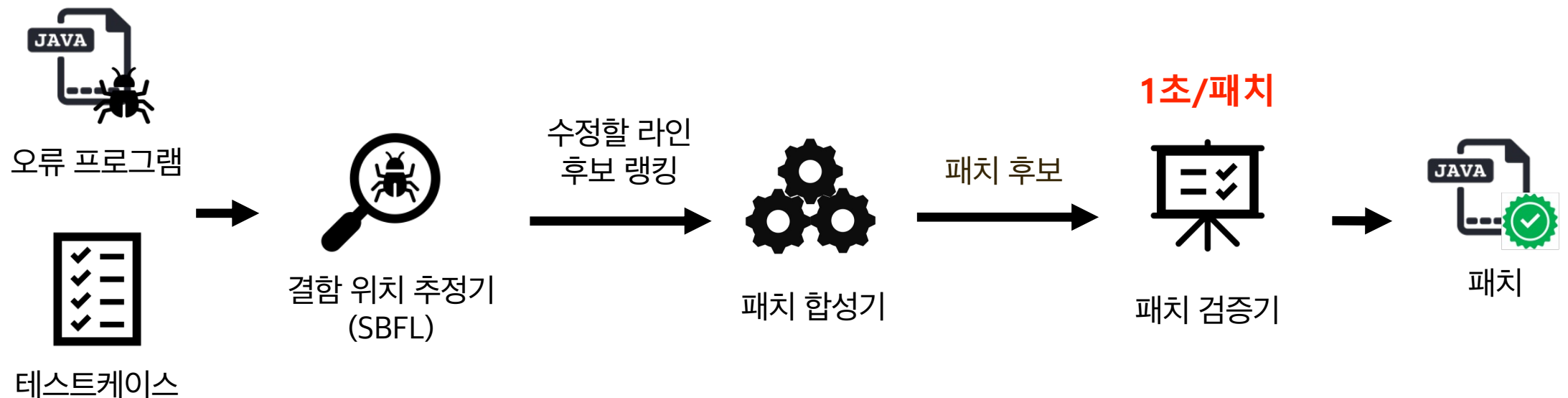
테스트를 통과하는 패치가 없는 버그의 수 99 (63%)	Incorrect 19
	Correct 40

# 원인: 매우 넓은 패치 공간 × 비싼 검증 시간



패치 공간 = 수정 위치 (라인) × 수정 방법 (템플릿) × 수정 재료 (표현식)

# 원인: 매우 넓은 패치 공간 × 비싼 검증 시간



SBFL 기준  
평균 수십~수백 라인

한 라인을 수정하는 방법이 매우 다양  
e.g., 표현식 바꾸기, 조건문 삽입, ...

복잡한 표현식 합성 필요  
e.g., 메소드 호출  $\square.\square(\square^*)$

패치 공간 = 수정 위치 (라인) × 수정 방법 (템플릿) × 수정 재료 (표현식)

# 기존 기술: 휴리스틱 기반의 제한적인 패치 공간 디자인

TBar [FSE'19]의 패치 템플릿 예시

```
FP2.2: + if (exp == null) return DEFAULT_VALUE;
      + ...exp...;
FP2.3: + if (exp == null) exp = exp1;
      + ...exp...;
FP2.4: + if (exp == null) continue;
      + ...exp...;
FP2.5: + if (exp == null)
      +   throw new IllegalArgumentException(...);
      + ...exp...;
```

```
FP4.2: + return DEFAULT_VALUE;
FP4.3: + try {
      +   statement; .....
      + } catch (Exception e) { ... }
FP4.4: + if (conditional_exp) {
      +   statement; .....
      + }
```

널-체크 조건식만  
합성 가능

?

기존의 명령문을  
감싸기만 가능

TBar로는 합성이 불가능한 패치 예시

99	99	return true;
100	100	}
101	101	
102	+	if (n.isDelProp()) {
103	+	return true;
104	+	}
102	105	

src/com/google/javascript/jscomp/DisambiguateProperties.java [CHANGE](#)

492	492	child != null;
493	493	child = child.getNext();
494	494	// Maybe STRING, GET, SET
495	+	if (child.isQuotedString()) {
496	+	continue;
497	+	}
495	498	



# 딥러닝 기반 APR 기술: 라인 별 패치 수를 고정

## AlphaRepair [FSE'22]

수정 할 라인    ×    라인 별 패치 수  
최대 40 고정        500

## TARE [ICSE'23]

3) *Patch Validation and Correctness*: In our experiment, Tare generates the patches based on the result of the fault localization technique. For each suspicious faulty statement, Tare adopts **beam** search strategy with size 100 to generate candidate patches. Thus, we **generate 100 patches** for each

## TENURE [ICSE'23]

as the **beam** size to keep multiple patch candidates [18]–[20], [22]. In this paper, **we use 500** in both localization settings to balance the effectiveness and efficiency of the repair process. Following the previous work [7], [20], [21], we set the running

Closure-71

```
(-) t.inGlobalScope();  
(+) parent.getJSDocInfo() != null
```

수정할 라인 후보가 68위

Closure-10

```
(-) if (fnType != null) {  
(+) if (fnType != null &&  
      fnType.hasInstanceType()) {
```

라인 별 패치 수를 늘려야만 탐색 가능

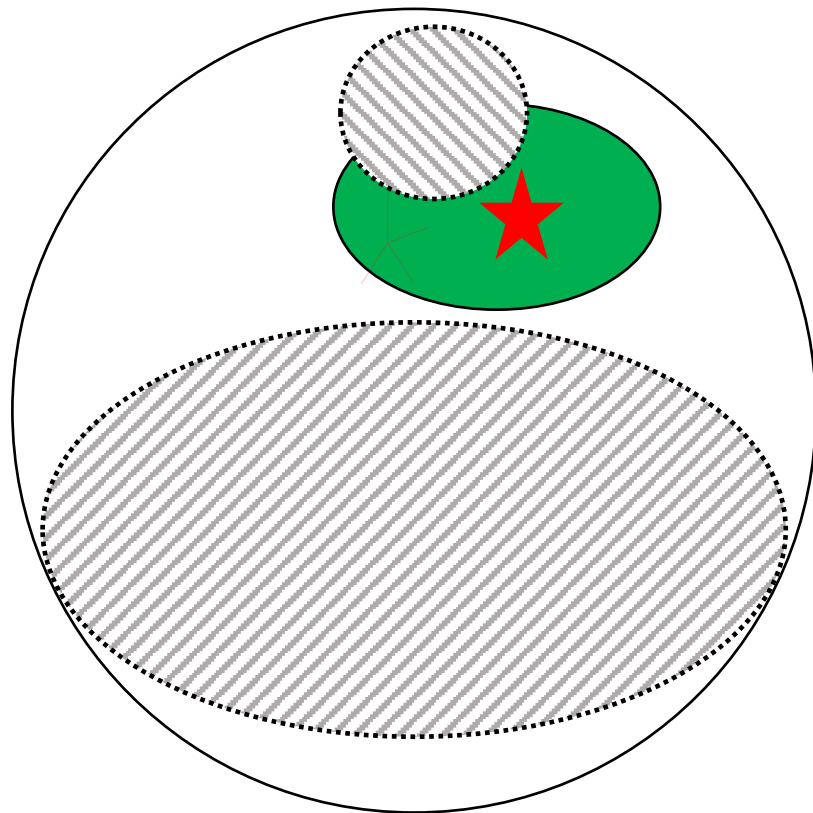
# 목표: 불필요한 패치 공간을 안전하게 잘라내기

★ : 정답 패치 공간

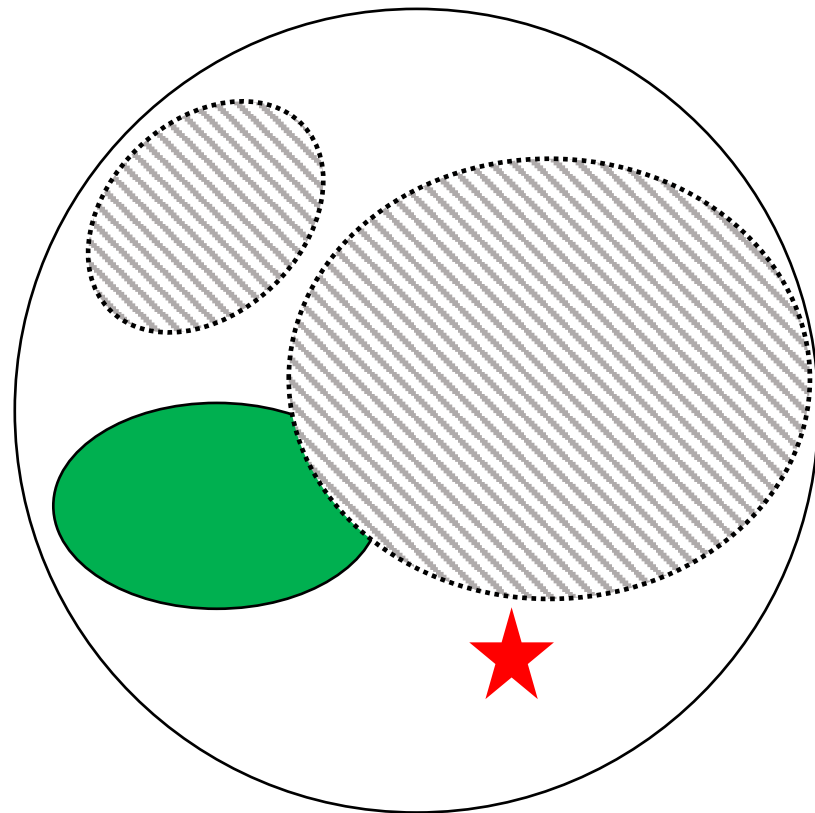
● : 기존 기술의  
탐색 공간

⊙ : 우리 기술로  
프루닝 된  
탐색 공간

버그 1



버그 2



# 관찰: 테스트를 통과할 가망 없는 패치 템플릿들

```
void checkVisibility(Node t, Node p) {  
  (-)  boolean isOverride = t.inGlobalScope() && parent.getType...;  
  (+)  boolean isOverride = parent.getJSDocInfo() && parent.getType...;  
  if (isOverride)  
    compiler.report(PRIVATE_OVERRIDE);  
  ...  
}
```

바뀌어야 하는 값

테스트를 통과하기 위해 반드시 실행해야 하는 명령문

- 관련 없는 변수를 바꾸는 패치: `JSDocInfo docinfo =` □`;`
- 예외 처리를 삽입하는 패치: `if (`□`) throw` □`;`
- 조건문 값에 영향을 주는 패치: `isOverride =` □ == □ `&& parent.getType...`

# 관찰: 테스트를 통과할 가망 없는 패치 템플릿들

```
void checkVisibility(Node t, Node p) {
```

```
(-)    boolean isOverride = t.inGlobalScope() && parent.getType...;
```

```
(+)    boolean isOverride = parent.getJSDocInfo() && parent.getType...;
```

```
    if (isOverride)
```

```
        compiler.report(PRIVATE_OVERRIDE);
```

```
    ...
```

```
}
```

바꿔야 하는 값

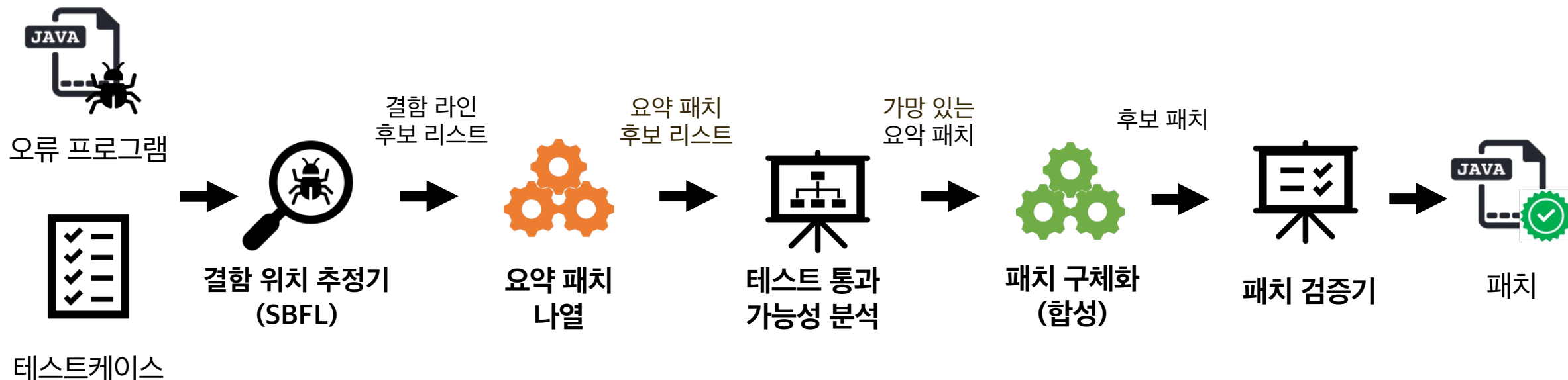
테스트를 통과하기 위해  
반드시 실행해야 하는 명령문

• ~~관련 없는 변수를 바꾸는 패치: JSDocInfo docinfo = ☐.~~

• ~~예외 처리를 삽입하는 패치: if (☐) throw ☐.~~

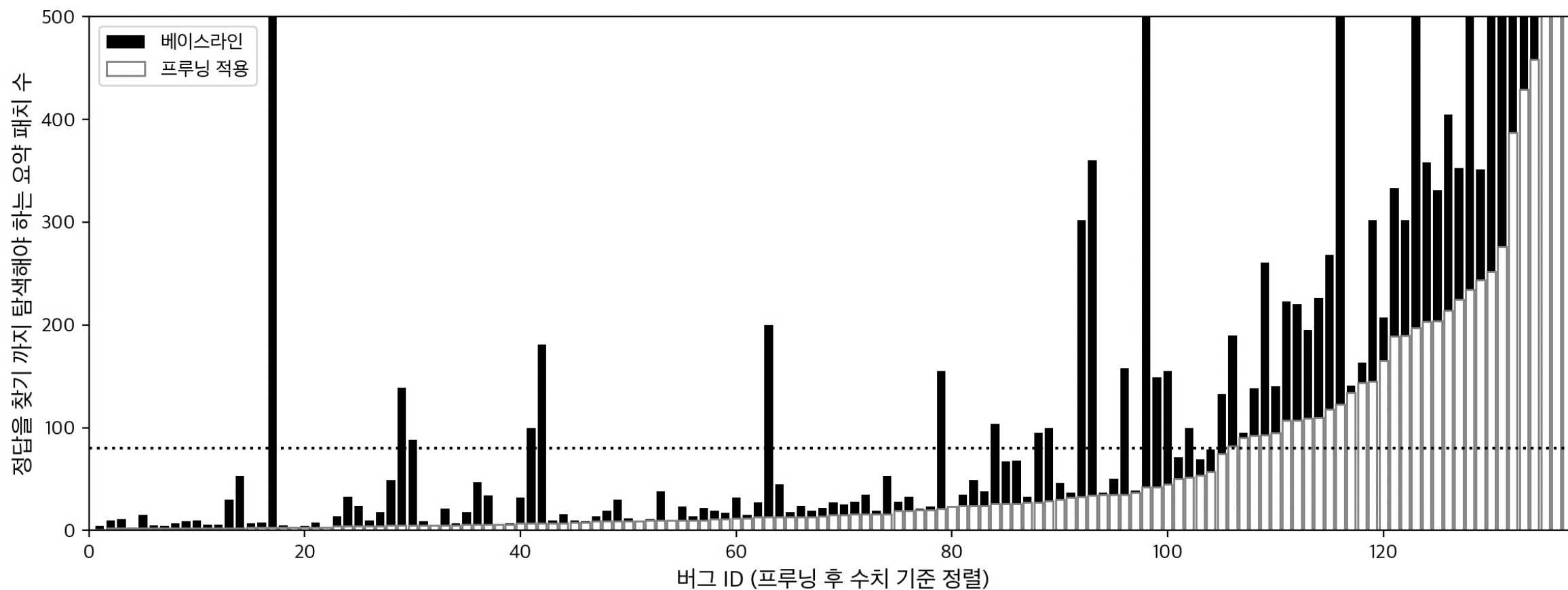
• 조건문 값에 영향을 주는 패치: isOverride = ☐ == ☐ && parent.getType...;

# 우리 전략: 가망 없는 패치 공간을 미리 쳐내기



$x = f(e);$	{	<del><math>x = f(e);</math></del> $x = f(\square);$	❌	{	$parent.getJsDoc() \neq null$	✅
		$x.f = \square;$	❌		$t.getNodes() \neq null$	❌
		<del><math>x = f(e);</math></del> $x = \square;$	✅		$!t.inGlobalScope()$	❌
		$if (\square) \{ Stmt; \}$	❌		$t.inGlobalScope() \ \&\& \ parent...$	❌
		$if (\square) \text{ return } \square;$	❌		$true$	✅
		$if (\square) \text{ throw } \square;$	❌			

# 초기 성능: 패치 공간 프루닝 효과



# 핵심 기술: 요약 패치의 테스트 만족 가능성 판단

Failing  
Test  
실행 경로

```
x = 2023;
```

```
if (difficult(y))
```

```
    x = 0;
```

```
assert(x == 2023);
```

# 핵심 기술: 요약 패치의 테스트 만족 가능성 판단

Failing  
Test  
실행 경로

```
x = 2023;
```

```
if (difficult(y))
```

```
    x = 0;
```

```
    y = □;
```

```
assert(x == 2023);
```



# Challenge: 정적 분석의 부정확도

Failing  
Test  
실행 경로

`x = 2023;`

`if (difficult(y))`

`x = 0;`

`y = □;`

`assert(x == 2023);`

일반적인 정적 분석 결과

Loc	Value
x	$[0, 0] \sqcup [2023, 2023]$
y	$\top$



$2023 \in [0, 2023]$

# 아이디어: 테스트 실행 정보를 활용하기

Failing  
Test  
실행 경로

```
x = 2023;
```

```
if (difficult(y))
```

true

```
x = 0;
```

```
y = □;
```

```
assert(x == 2023);
```

테스트 실행 정보로 가이드 한  
정적 분석 결과

Loc	Value
x	<b>[0, 0]</b> $\sqcup$ <del>[2023, 2023]</del>
y	T



2023  $\notin$  [0, 0]

# 프로그램이 바뀐 이후에도 실행 정보를 사용 가능할까?

```
x = 2023;
```

```
y = □;
```

```
if (difficult(y))
```

```
    x = 0;
```

```
assert(x == 2023);
```

```
x = 2023;
```

```
x = □;
```

```
if (difficult(y))
```

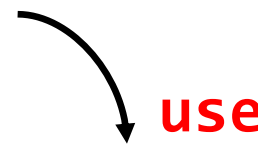
```
    x = 0;
```

```
assert(x == 2023);
```

# 프로그램이 바뀐 이후에도 실행 정보를 사용 가능할까?

```
x = 2023;
```

```
def  
y = □;  
if (difficult(y))
```

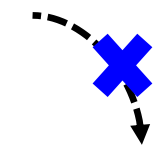
A curved arrow points from the variable 'y' in the assignment 'y = □;' to the argument 'y' in the function call 'difficult(y)'. The word 'def' is written in red above the arrow, and the word 'use' is written in red below the arrow.

```
x = 0;
```

```
assert(x == 2023);
```

```
x = 2023;
```

```
x = □;  
if (difficult(y))
```

A curved arrow points from the variable 'x' in the assignment 'x = □;' to the argument 'y' in the function call 'difficult(y)'. A blue 'X' is placed over the arrow, indicating a mismatch or error.

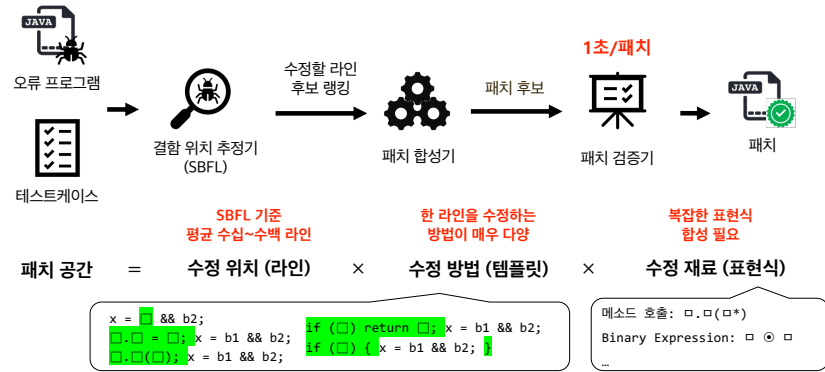
```
x = 0;
```

```
assert(x == 2023);
```

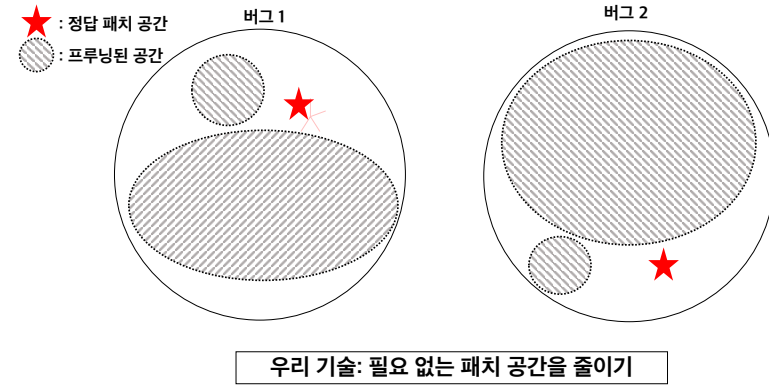
패치와 대상 표현식 사이의 **의존성**이 없다면 가능

# Summary

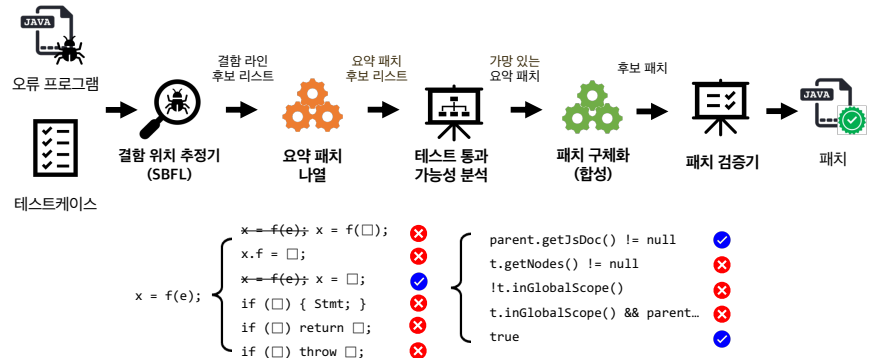
원인: 매우 넓은 패치 공간 × 비싼 검증 시간



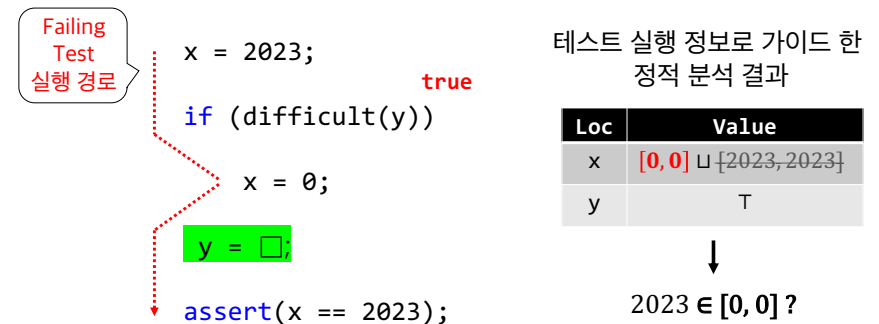
목표: 패치 공간을 안전하고 효과적으로 자르기



우리 전략: 가망 없는 패치 공간을 미리 쳐내기



아이디어: 테스트 실행 정보를 활용하기



감사합니다

# 의존성 파악 역시 실행 정보를 활용하면 더 정확

Failing  
Test  
실행 경로

```
x = 2023;
```

```
y = □;
```

```
if (difficult(x))
```

```
    y = 0;
```

```
if (difficult(y))
```

```
    x = 0;
```

```
assert(x == 2023);
```

테스트 실행 정보를  
사용할 수 있을까?

# 의존성 파악 역시 실행 정보를 활용하면 더 정확

Failing  
Test  
실행 경로

```
x = 2023;
```

```
y = □; def
```

```
if (difficult(x))
```

```
    y = 0;
```

```
if (difficult(y))
```

```
    x = 0;
```

```
assert(x == 2023);
```

**use**

일반적인 데이터/제어흐름-의존성  
⇒ 영향 있음

# 의존성 파악 역시 실행 정보를 활용하면 더 정확

Failing  
Test  
실행 경로

```
x = 2023;
```

```
y = □;
```

```
if (difficult(x))
```

```
    y = 0;
```

```
if (difficult(y))
```

```
    x = 0;
```

```
assert(x == 2023);
```

테스트 실행 정보를 고려하면?  
⇒ y = 0; 명령문이 항상 실행  
⇒ 패치와의 의존성 없음