

Lecture I 2: 프로그램 분석 개괄

오학주
고려대학교 컴퓨터학과



소프트웨어 오류 문제

- 소프트웨어 오류는 사회 모든 영역에서 발생하는 주제



금융거래SW(2012)

자율주행SW(2017)

의료SW(2018)

블록체인SW(2020)

- 소프트웨어 결함으로 인한 사회경제적 비용은 연 1.7조 달러로 추정



606
software fails



\$1.7
trillion



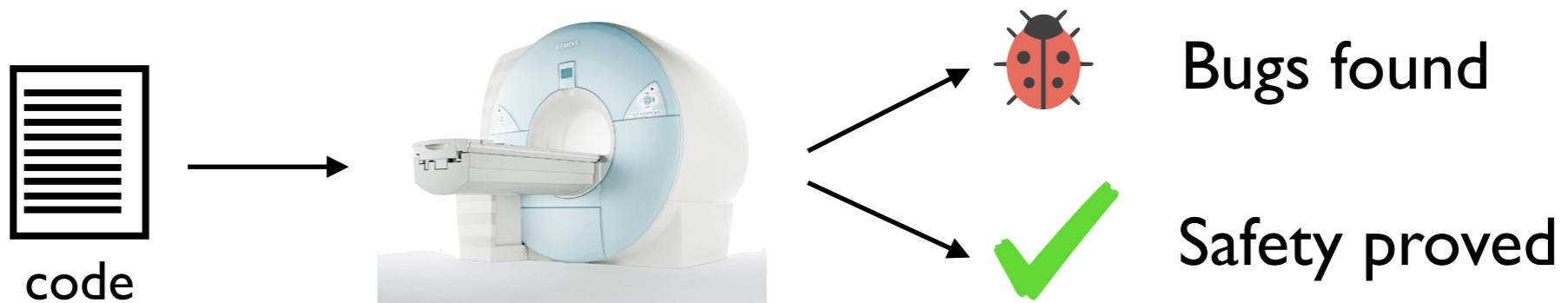
3.6 billion
affected users



268 years
in downtime

Software fail watch (5th edition). 2017

프로그램 분석



- 소프트웨어의 실행 성질을 엄밀히 확인하는 기술
 - 정적 분석: 실행 전 확인 (요약 해석, 모델 체킹 등)
 - 동적 분석: 실행 중 확인 (퍼징, 기호 실행 등)
- 소프트웨어 산업에서 적극적으로 활용되기 시작



Infer

COVERITY
BY SYNOPSYS

PolySpace
TECHNOLOGIES

Clang Static
Analyzer

sonarQube

Sparrow
The Early Bird

Dagstuhl Seminar 23131

Software Bug Detection: Challenges and Synergies

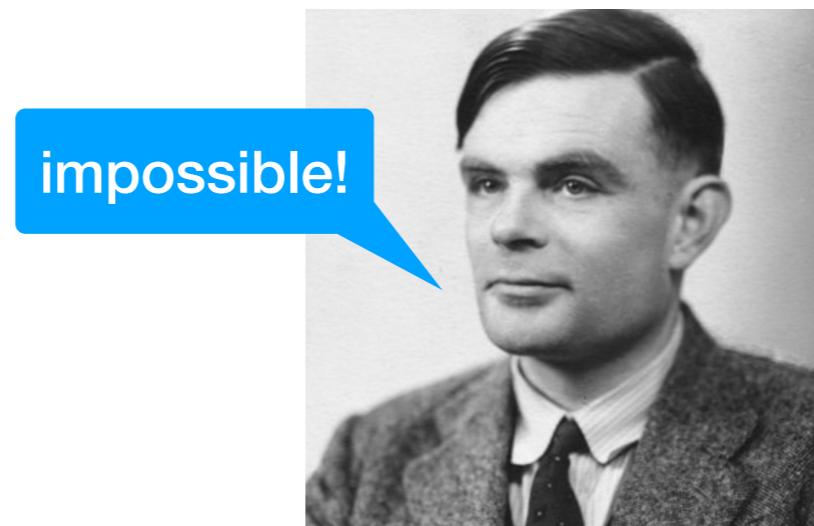
(Mar 26 – Mar 31, 2023)



© SCHLOSS DAGSTUHL – LZI GMBH
licensed under Creative Commons License CC BY-NC-ND

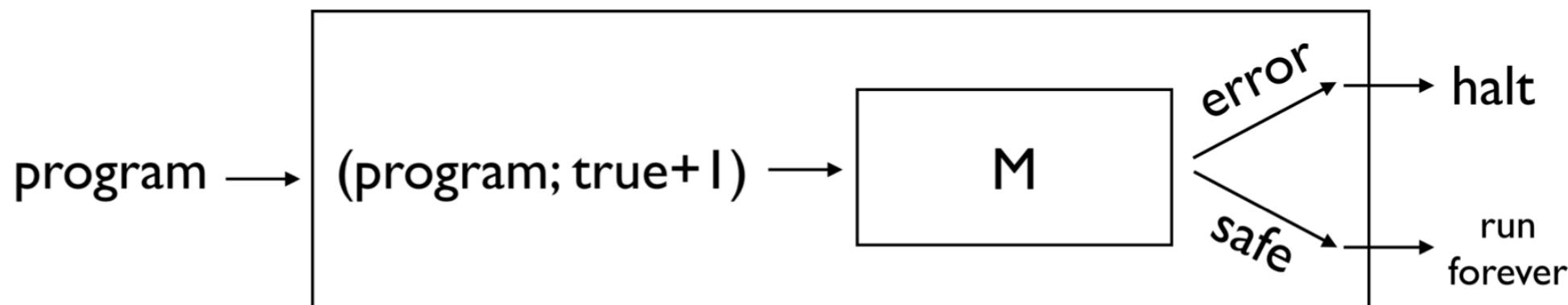
완벽한 프로그램 분석은 불가능

- Halting problem: 주어진 프로그램이 항상 종료하는가?

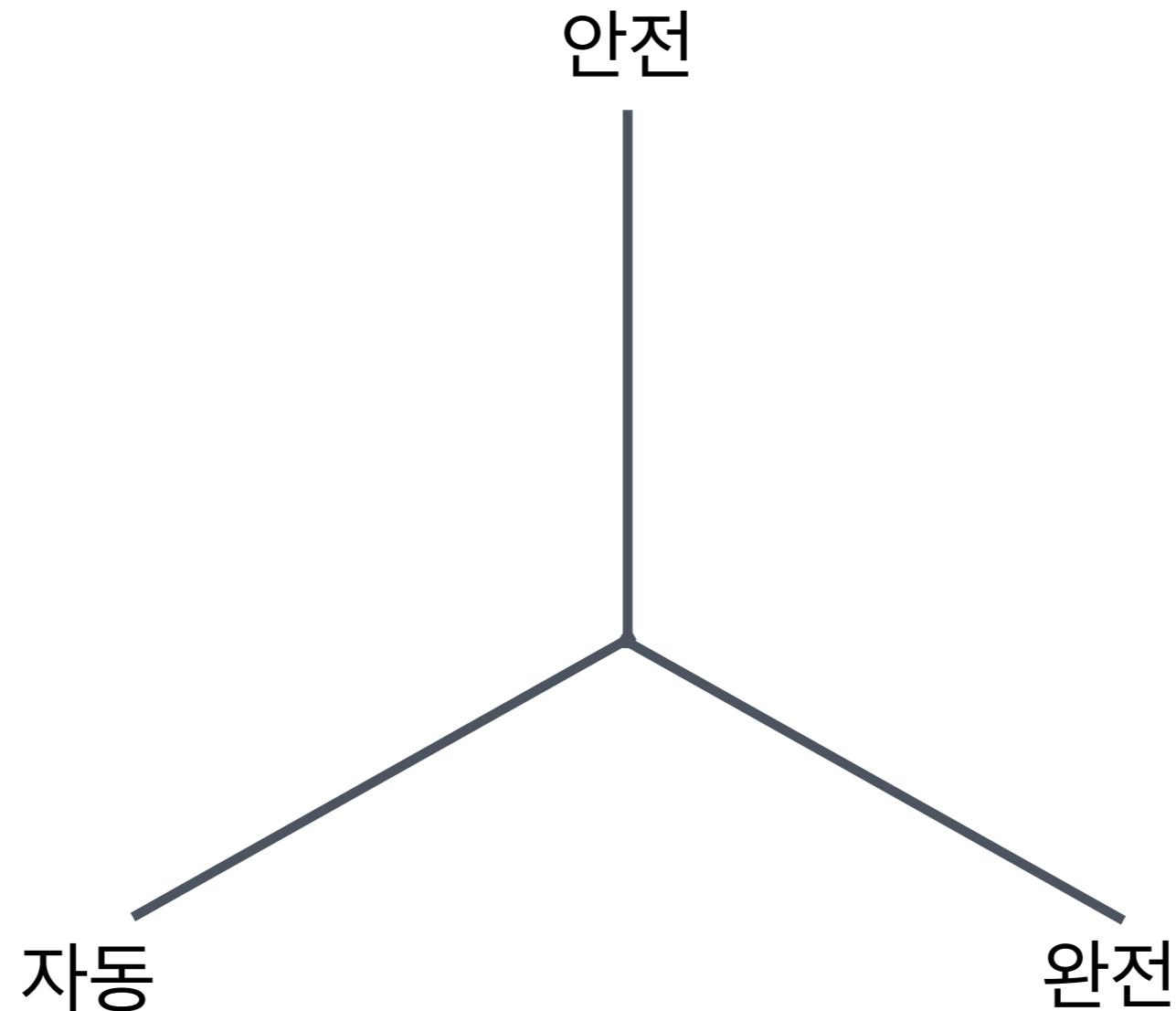


Alan Turing

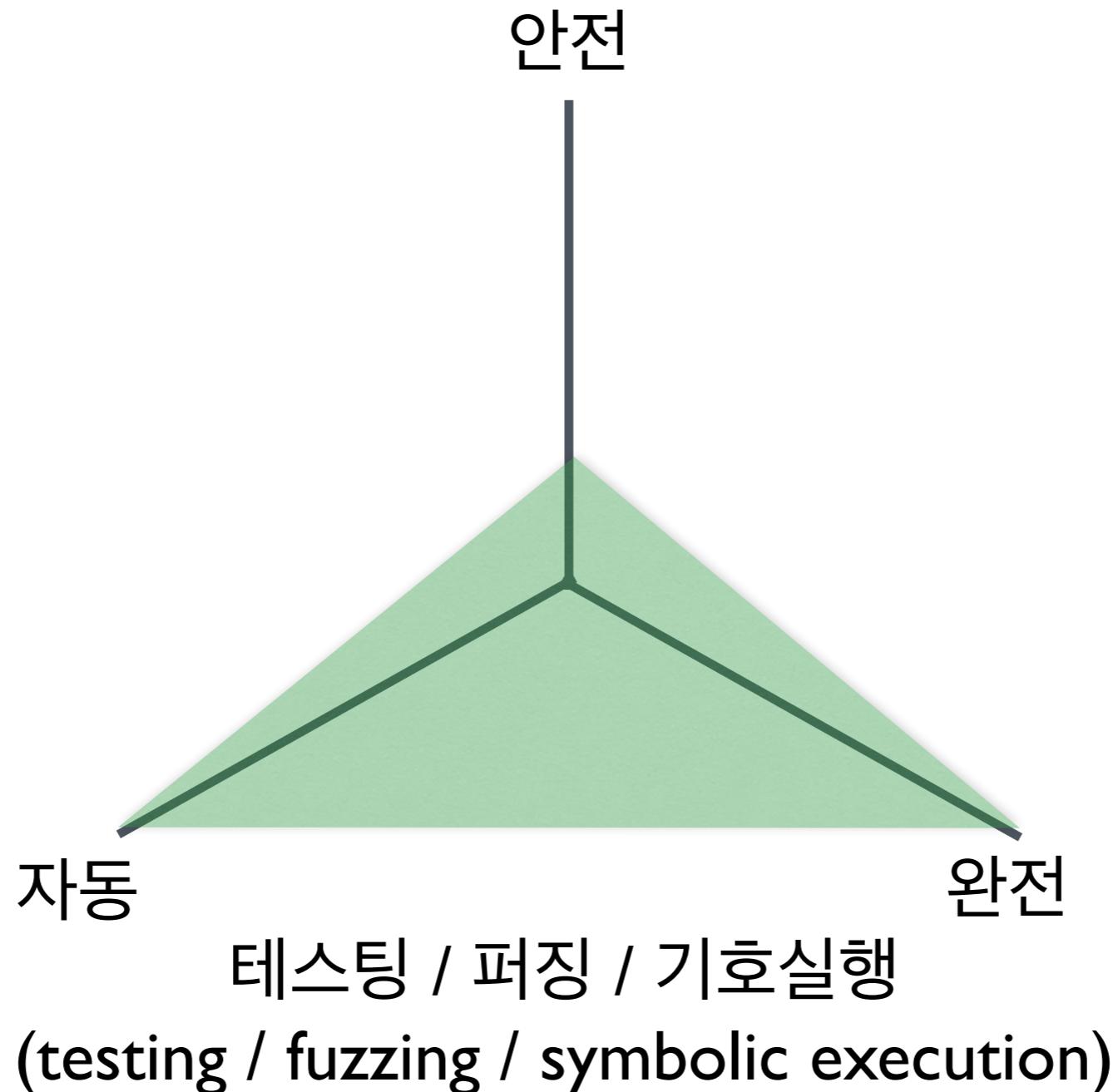
- 완벽한 프로그램 분석기 M 이 존재한다면 Halting problem이 풀린다.



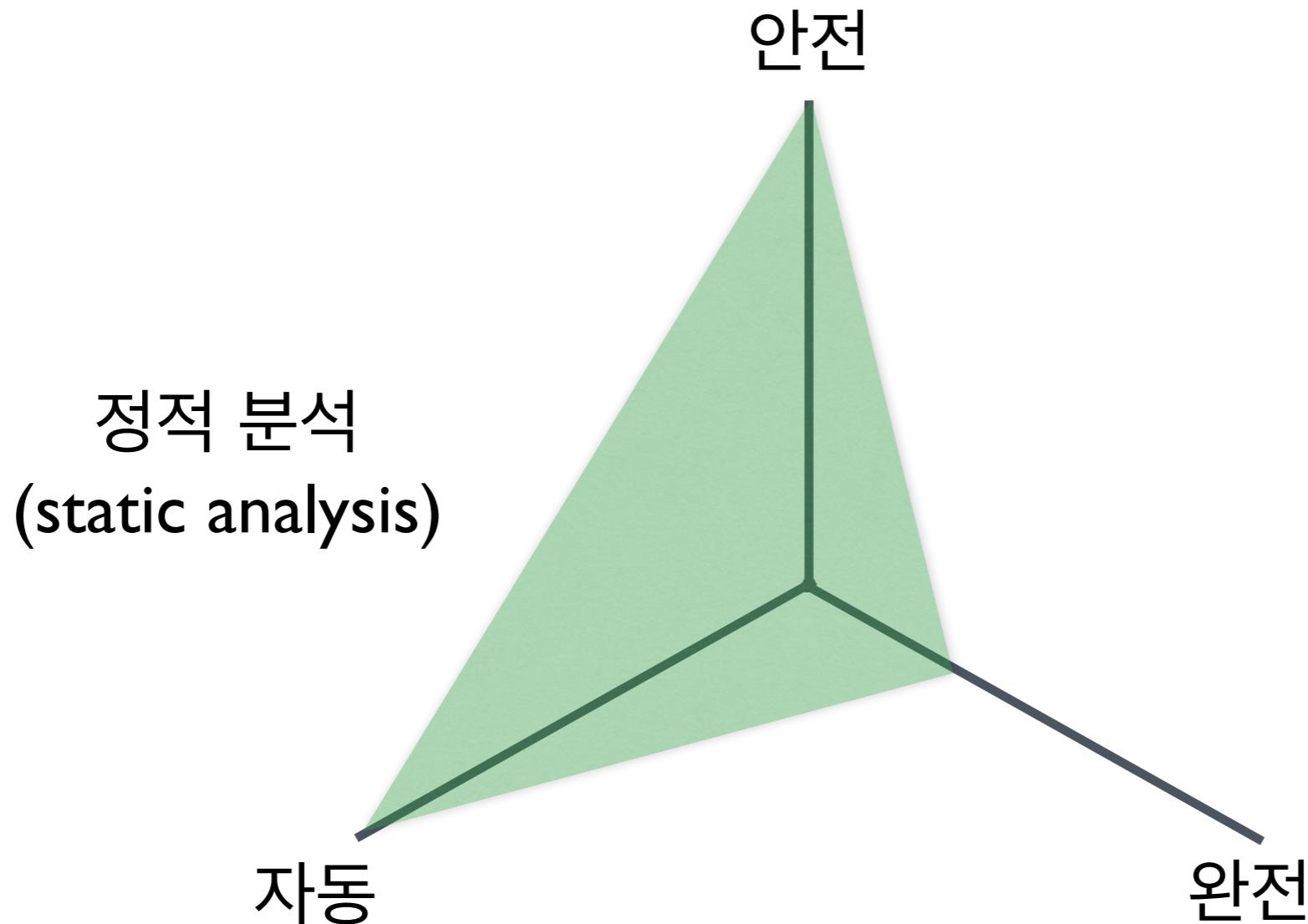
프로그램 분석 기법



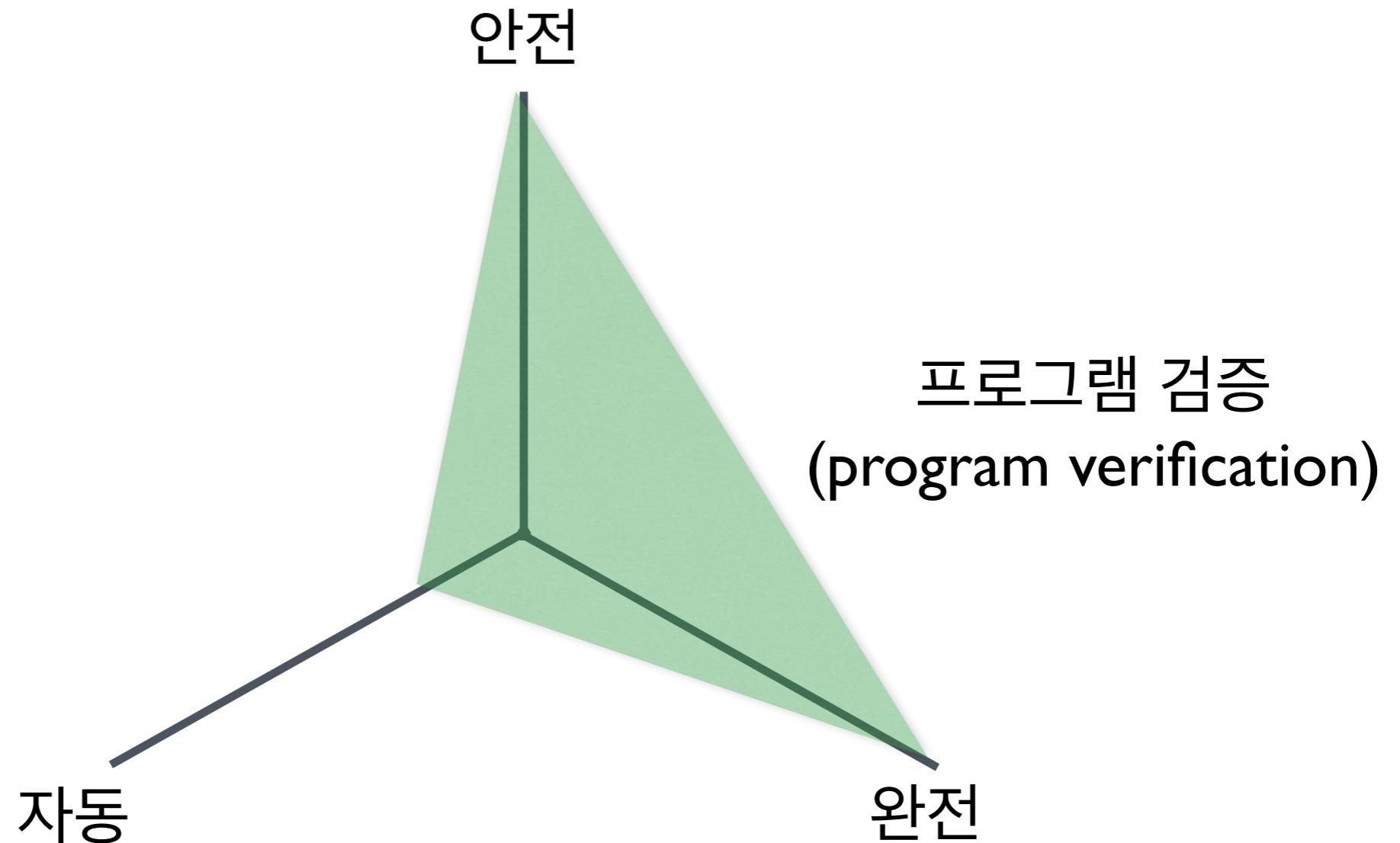
프로그램 분석 기법



프로그램 분석 기법

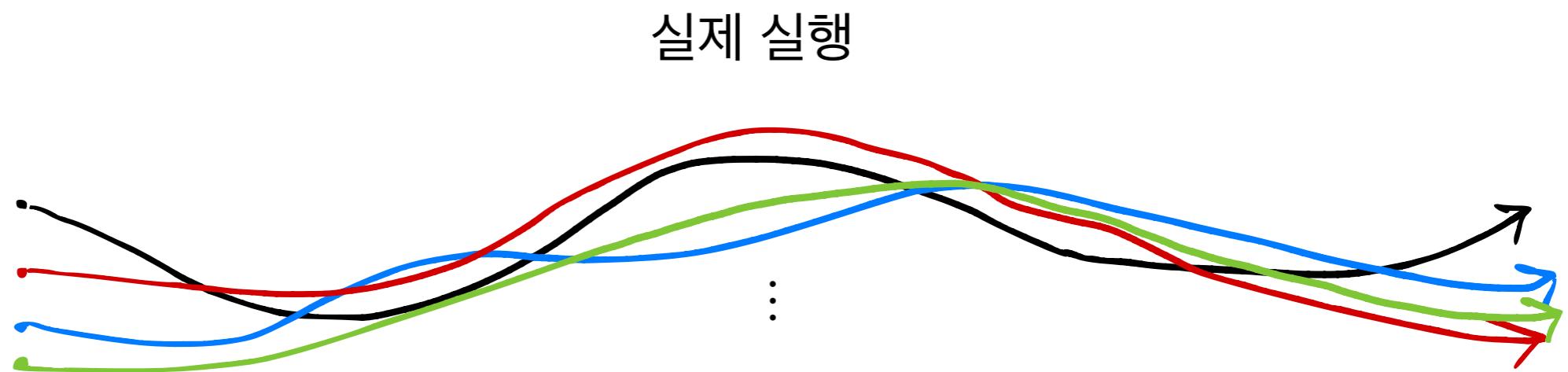


프로그램 분석 기법



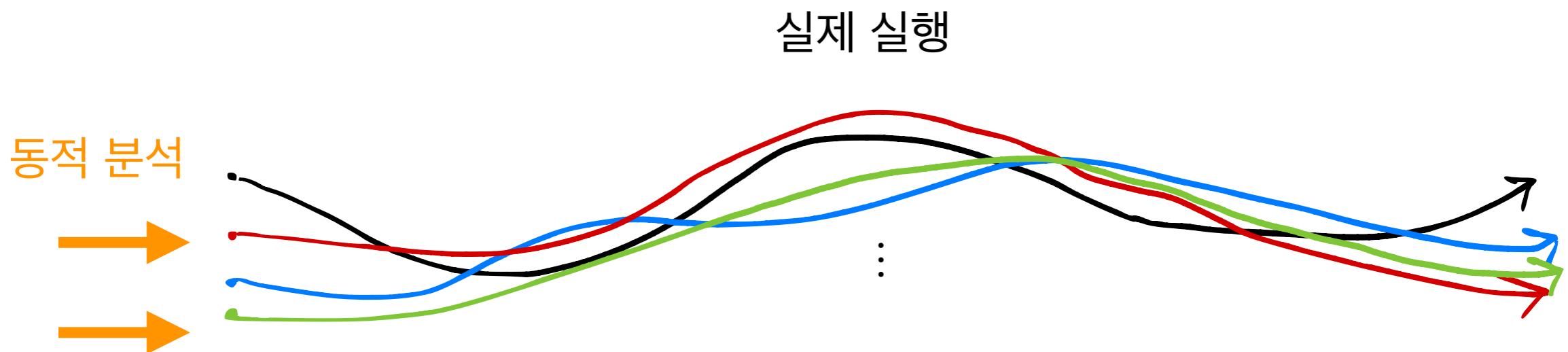
프로그램 분석 원리

- 근사/요약 (approximation/abstraction)
 - 동적 분석: under-approximation
 - 정적 분석: over-approximation



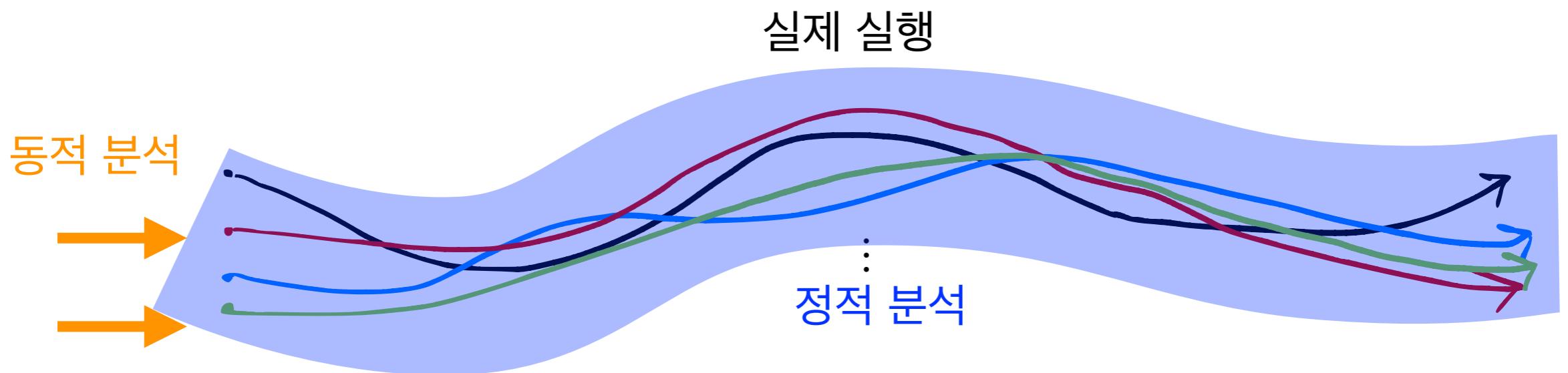
프로그램 분석 원리

- 근사/요약 (approximation/abstraction)
 - 동적 분석: under-approximation
 - 정적 분석: over-approximation



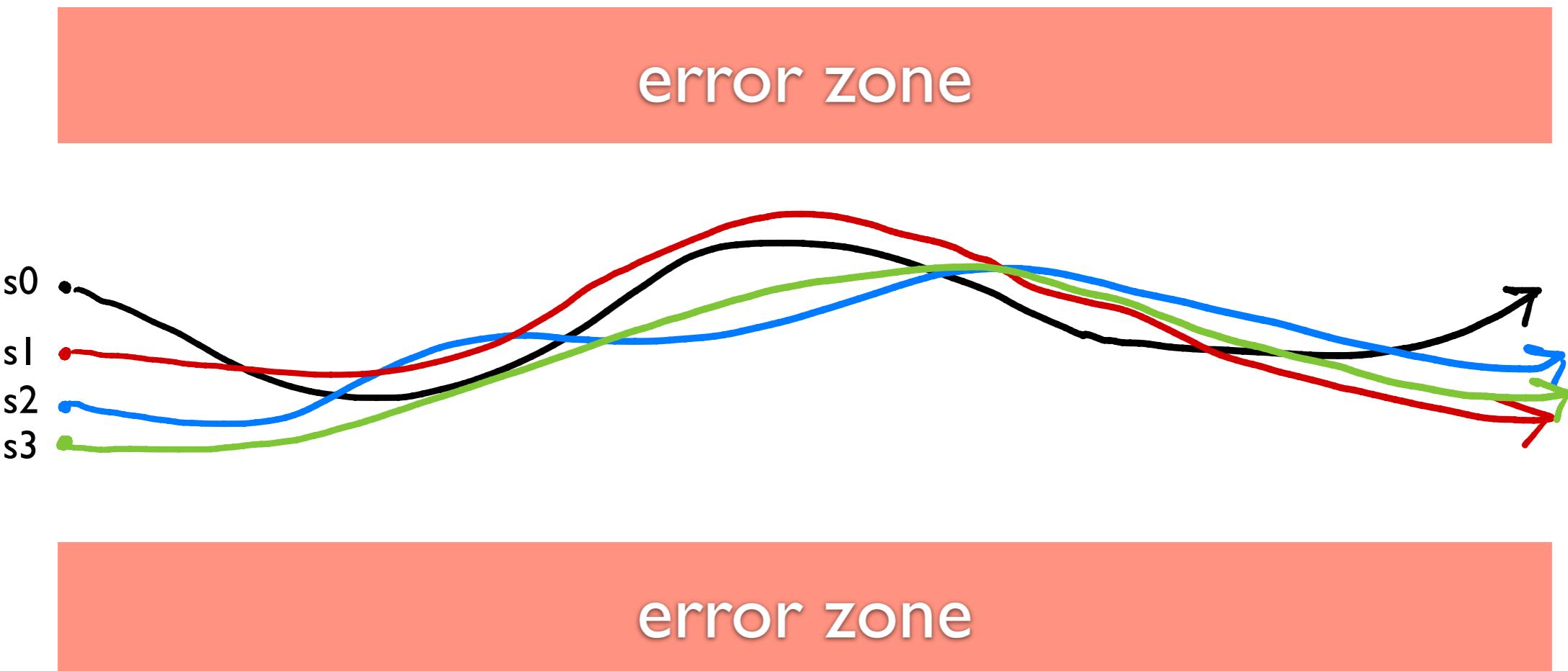
프로그램 분석 원리

- 근사/요약 (approximation/abstraction)
 - 동적 분석: under-approximation
 - 정적 분석: over-approximation



테스팅 / 퍼징 원리

- 프로그램의 개별 실행 경로들을 일일이 추적



테스팅 / 페징 원리

```
int double (int v) {  
    return 2*v;  
}
```

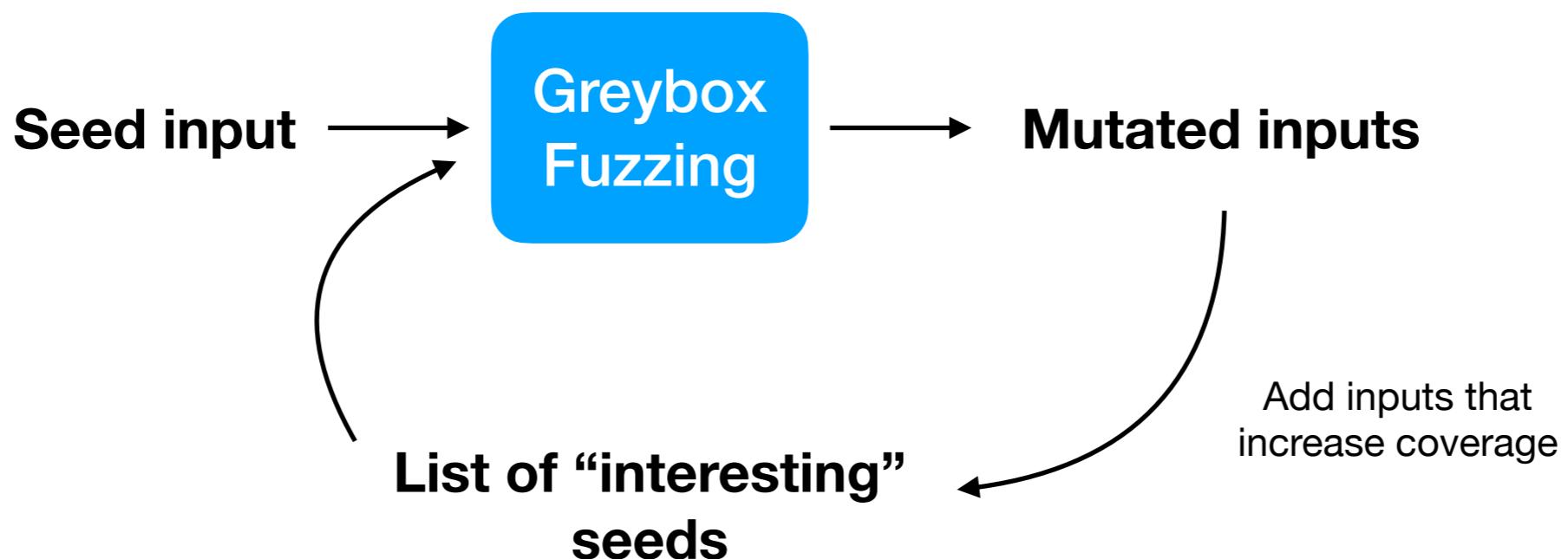
I. Error-triggering test?

```
void testme(int x, int y) {  
  
    z := double (y);  
  
    if (z==x) {  
        if (x>y+10) {  
            Error;  
        }  
    }  
}
```

2. Probability of the error?
(assume $0 \leq x,y \leq 10,000$)

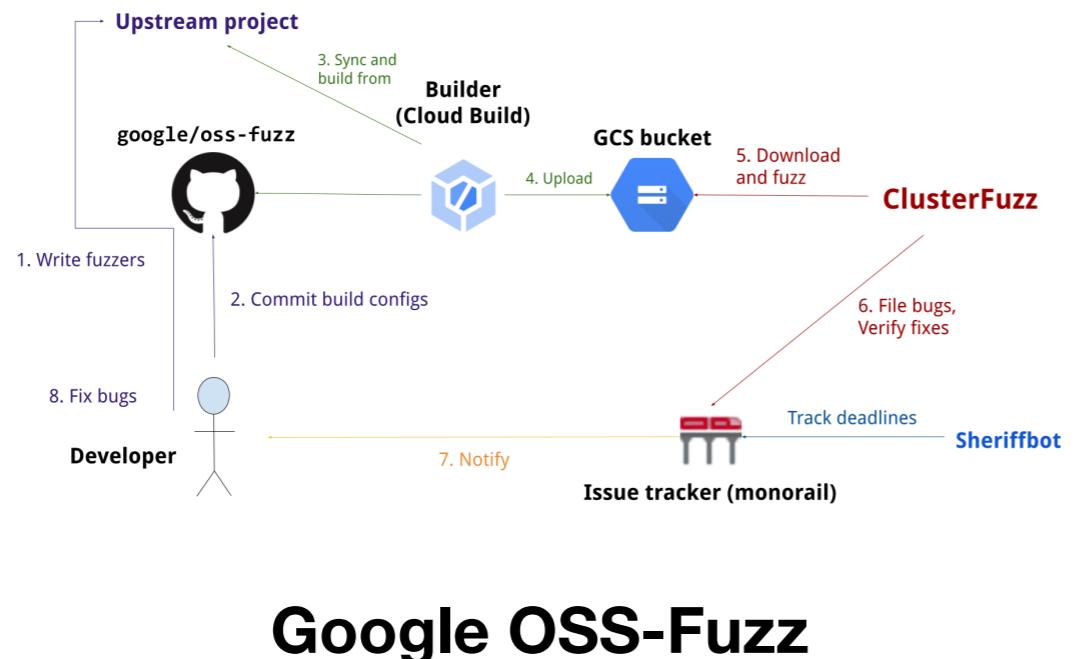
cf) 퍼징 방식

- Blackbox fuzzing
- Whitebox fuzzing
- Greybox fuzzing



산업체 적용 사례

- AFL (<https://github.com/google/AFL>)
- OSS-Fuzz (<https://github.com/google/oss-fuzz>)



DOI:10.1145/3363824
Reviewing software testing techniques for finding security vulnerabilities.

BY PATRICE GODEFROID

Fuzzing: Hack, Art, and Science

FUZZING, OR FUZZ TESTING, is the process of finding security vulnerabilities in input-parsing code by repeatedly testing the parser with modified, or fuzzed, inputs.³⁵ Since the early 2000s, fuzzing has become a mainstream practice in assessing software security. Thousands of security vulnerabilities have been found while fuzzing all kinds of software applications for processing documents, images, sounds, videos, network packets, Web pages, among others. These applications must deal with untrusted inputs

encoded in complex data formats. For example, the Microsoft Windows operating system supports over 360 file formats and includes millions of lines of code just to handle all of these.

Most of the code to process such files and packets evolved over the last 20+ years. It is large, complex, and written in C/C++ for performance reasons. If an attacker could trigger a buffer-overflow bug in one of these applications, s/he could corrupt the memory of the application and possibly hijack its execution to run malicious code (elevation-of-privilege attack), or steal internal information (information-disclosure attack), or simply crash the application (denial-of-service attack).⁹ Such attacks might be launched by tricking the victim into opening a single malicious document, image, or Web page. If you are reading this article on an electronic device, you are using a PDF and JPEG parser in order to see Figure 1.

Buffer-overflows are examples of security vulnerabilities: they are programming errors, or bugs, and typically triggered only in specific hard-to-find corner cases. In contrast, an exploit is a piece of code which triggers a security vulnerability and then takes advantage of it for malicious purposes. When exploitable, a security vulnerability is like an unintended backdoor in a software application that lets an attacker enter the victim's device.

There are approximately three main ways to detect security vulnerabilities in software.

Static program analyzers are tools that automatically inspect code and

» key insights

- Fuzzing means automatic test generation and execution with the goal of finding security vulnerabilities.
- Over the last two decades, fuzzing has become a mainstay in software security. Thousands of security vulnerabilities in all kinds of software have been found using fuzzing.
- If you develop software that may process untrusted inputs and have never used fuzzing, you probably should.

Microsoft

기호 실행 원리

- 동일한 실행 경로를 가지는 입력들을 한번에 실행



기호 실행 원리

1

x: α , y: β
pc: true

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {
```

1

```
    z := double (y);
```

2

```
    if (z==x) {
```

3

```
        if (x>y+10) {
```

4 Error;

```
    } else { 5 ...}
```

```
}
```

6 }

기호 실행 원리

```
int double (int v) {  
    return 2*v;  
}  
  
void testme(int x, int y) {  
1   z := double (y);  
2   if (z==x) {  
3       if (x>y+10) {  
4           Error;  
    } else { 5 ...}  
6 }
```

1 $x: \alpha, y: \beta$
 pc: true

2 $x: \alpha, y: \beta, z: 2\beta$
 pc: true

기호 실행 원리

```
int double (int v) {  
    return 2*v;  
}  
  
void testme(int x, int y) {  
1   z := double (y);  
2   if (z==x) {  
3       if (x>y+10) {  
4           Error;  
    } else { 5 ...}  
6 }
```

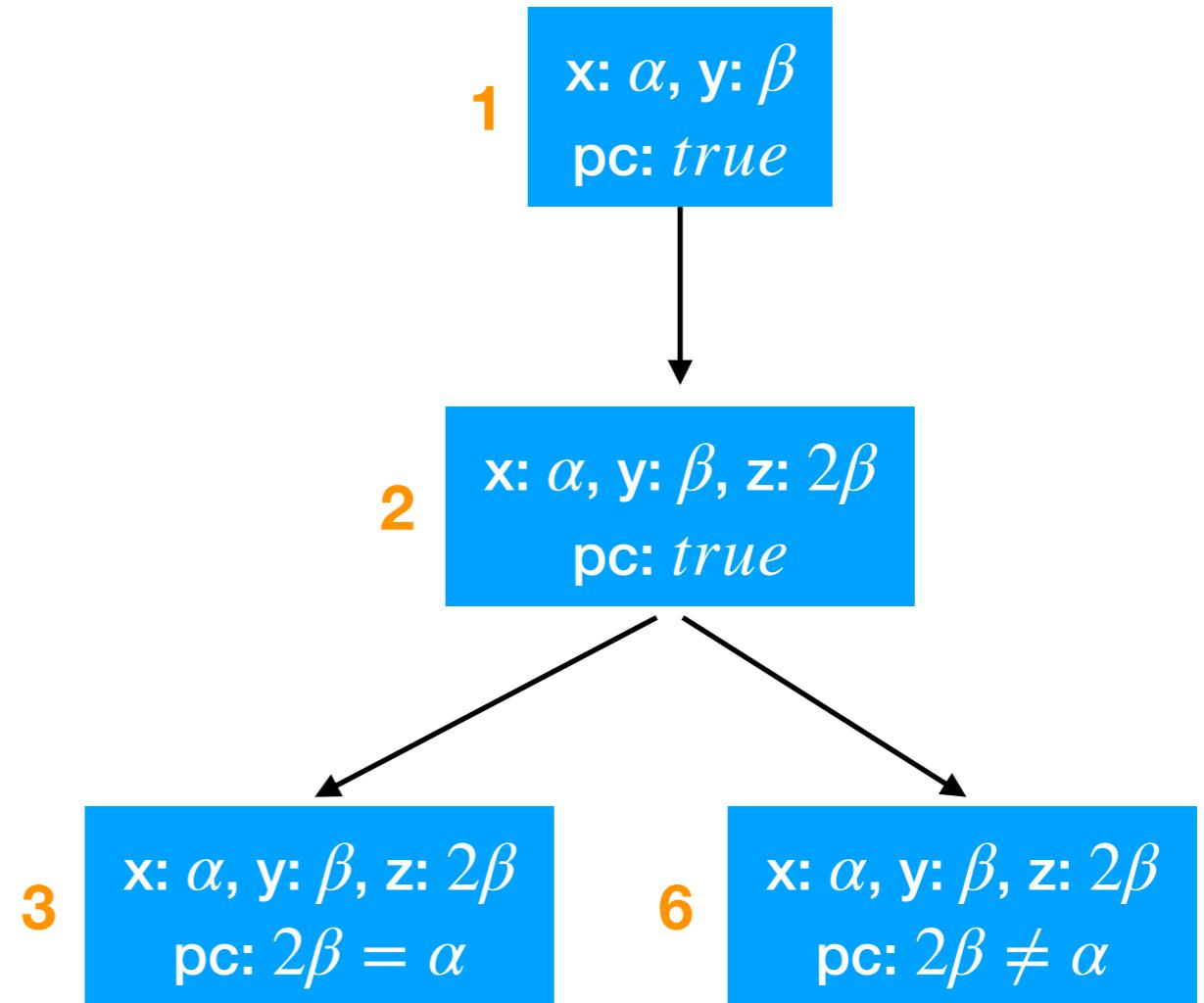
1 $x: \alpha, y: \beta$
pc: true

2 $x: \alpha, y: \beta, z: 2\beta$
pc: true

3 $x: \alpha, y: \beta, z: 2\beta$
pc: $2\beta = \alpha$

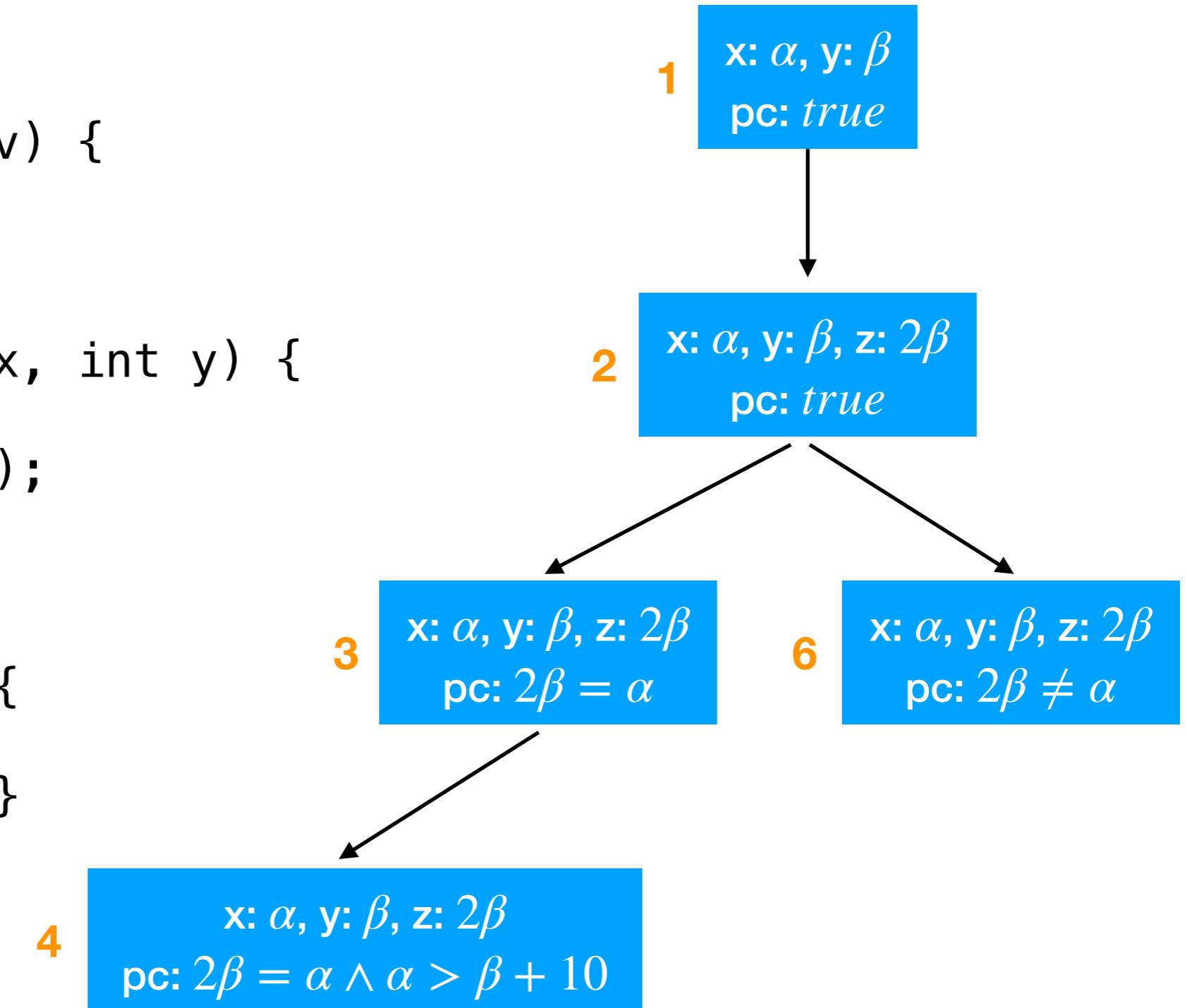
기호 실행 원리

```
int double (int v) {  
    return 2*v;  
}  
  
void testme(int x, int y) {  
    1   z := double (y);  
    2   if (z==x) {  
        3     if (x>y+10) {  
            4     Error;  
        } else { 5 ...}  
    }  
    6 }
```



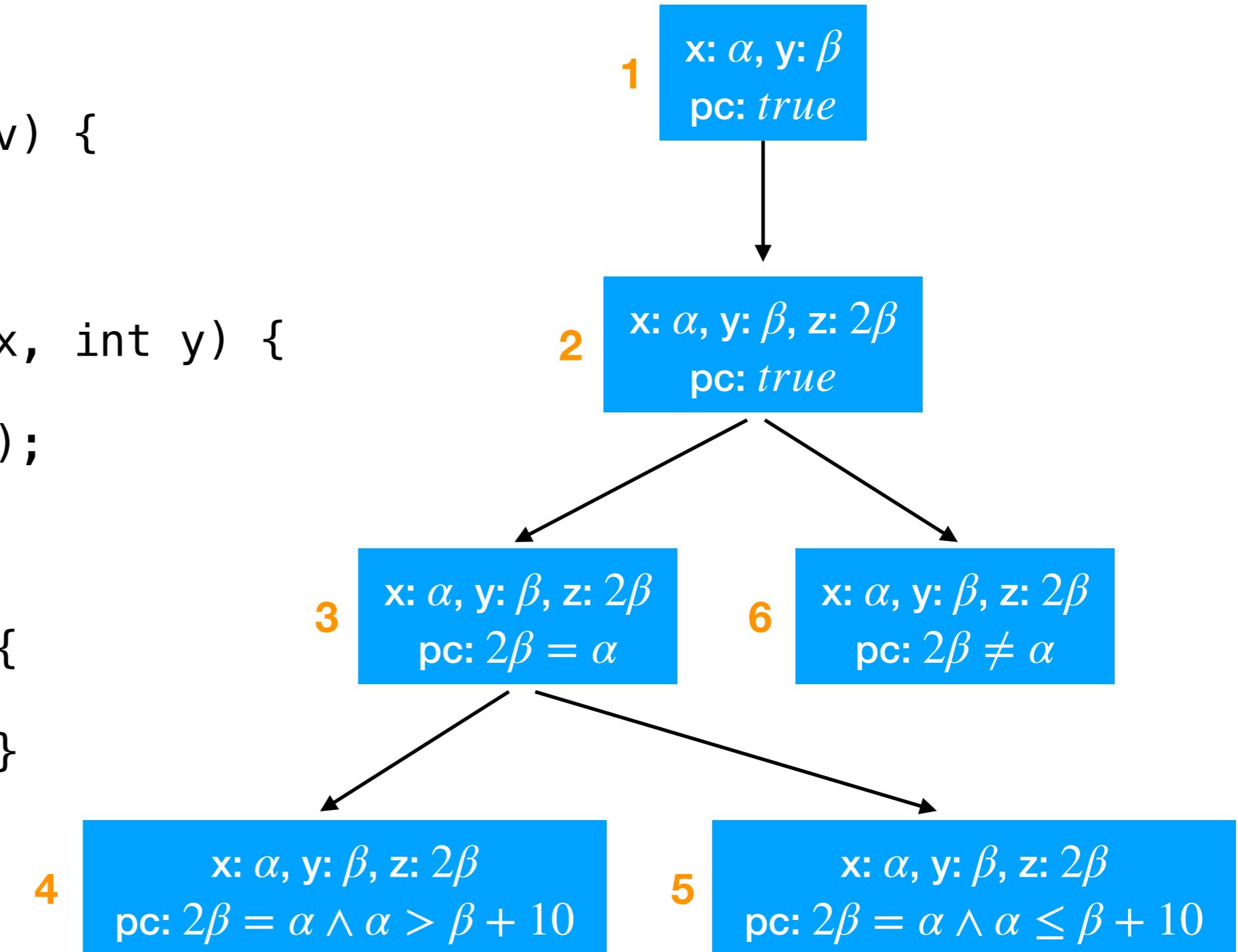
기호 실행 원리

```
int double (int v) {  
    return 2*v;  
}  
  
void testme(int x, int y) {  
    1   z := double (y);  
    2   if (z==x) {  
        3     if (x>y+10) {  
            4     Error;  
        } else { 5 ...}  
    }  
    6 }
```



기호 실행 원리

```
int double (int v) {  
    return 2*v;  
}  
  
void testme(int x, int y) {  
    1   z := double (y);  
    2   if (z==x) {  
        3     if (x>y+10) {  
            4     Error;  
        } else { 5 ...}  
    }  
    6 }
```



cf) 콘콜릭 테스팅 (Concolic Testing)

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {  
    ←—————  
    z := double (y);  
  
    if (z==x) {  
  
        if (x>y+10) {  
            Error;  
        }  
    }  
}
```

Concrete
State

x=22, y=7

Symbolic
State

x=a, y=β

true

1st iteration

cf) 콘콜릭 테스팅 (Concolic Testing)

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {  
  
    z := double (y);  
    ←—————  
    if (z==x) {  
  
        if (x>y+10) {  
            Error;  
        }  
    }  
}
```

Concrete
State

x=22, y=7,
z=14

Symbolic
State

x=a, y=β,z=2*β
true

1st iteration

cf) 콘콜릭 테스팅 (Concolic Testing)

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {  
  
    z := double (y);  
  
    if (z==x) {  
  
        if (x>y+10) {  
            Error;  
        }  
    }  
}
```

Concrete
State

x=22, y=7,
z=14

Symbolic
State

x=a, y=β, z=2*β
2*β ≠ a

1st iteration

cf) 콘콜릭 테스팅 (Concolic Testing)

```
int double (int v) {  
    return 2*v;  
}  
  
void testme(int x, int y) {  
    z := double (y);  
    if (z==x) {  
        if (x>y+10) {  
            Error;  
        }  
    }  
}
```

1st iteration

Concrete State	Symbolic State
$x=22, y=7, z=14$	Solve: $2^*\beta = a$ Solution: $a=2, \beta=1$ $x=a, y=\beta, z=2^*\beta$ $2^*\beta \neq a$

cf) 콘콜릭 테스팅 (Concolic Testing)

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {  
    ←—————  
    z := double (y);  
  
    if (z==x) {  
  
        if (x>y+10) {  
            Error;  
        }  
    }  
}
```

Concrete
State

x=2, y=1

Symbolic
State

x=a, y=β

true

2nd iteration

cf) 콘콜릭 테스팅 (Concolic Testing)

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {  
  
    z := double (y);  
    ←—————  
    if (z==x) {  
  
        if (x>y+10) {  
            Error;  
        }  
    }  
}
```

Concrete
State

x=2, y=1,
z=2

Symbolic
State

x=a, y=β,z=2*β
true

2nd iteration

cf) 콘콜릭 테스팅 (Concolic Testing)

```
int double (int v) {  
    return 2*v;  
}  
  
void testme(int x, int y) {  
  
    z := double (y);  
  
    if (z==x) {  
        ←—————  
        if (x>y+10) {  
            Error;  
        }  
    }  
}
```

Concrete
State

x=2, y=1,
z=2

Symbolic
State

x=a, y=β, z=2*β
2*β = a

2nd iteration

cf) 콘콜릭 테스팅 (Concolic Testing)

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {  
  
    z := double (y);  
  
    if (z==x) {  
  
        if (x>y+10) {  
            Error;  
        }  
    }  
}
```

Concrete
State

x=2, y=1,
z=2

Symbolic
State

x=a, y= β , z= $2^*\beta$
 $2^*\beta = a \wedge$
 $a \leq \beta + 10$

2nd iteration

cf) 콘콜릭 테스팅 (Concolic Testing)

```
int double (int v) {  
    return 2*v;  
}  
  
void testme(int x, int y) {  
    z := double (y);  
    if (z==x) {  
        if (x>y+10) {  
            Error;  
        }  
    }  
}
```

Concrete
State

Solve: $2^*\beta = \alpha \wedge \alpha > \beta + 10$
Solution: $\alpha=30, \beta=15$

$x=2, y=1,$
 $z=2$

Symbolic
State

$x=\alpha, y=\beta, z=2^*\beta$
 $2^*\beta = \alpha \wedge$
 $\alpha \leq \beta + 10$

2nd iteration

cf) 콘콜릭 테스팅 (Concolic Testing)

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {  
    ←—————  
    z := double (y);  
  
    if (z==x) {  
  
        if (x>y+10) {  
            Error;  
        }  
    }  
}
```

Concrete
State

x=30, y=15

Symbolic
State

x=a, y=β

true

3rd iteration

cf) 콘콜릭 테스팅 (Concolic Testing)

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {  
  
    z := double (y);  
    ←—————  
    if (z==x) {  
  
        if (x>y+10) {  
            Error;  
        }  
    }  
}
```

Concrete
State

x=30, y=15,
z=30

Symbolic
State

x=a, y=β,z=2*β
true

3rd iteration

cf) 콘콜릭 테스팅 (Concolic Testing)

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {  
  
    z := double (y);  
  
    if (z==x) {  
        ←—————  
        if (x>y+10) {  
            Error;  
        }  
    }  
}
```

Concrete
State

x=30, y=15,
z=30

Symbolic
State

x=a, y=β, z=2*β
2*β = a

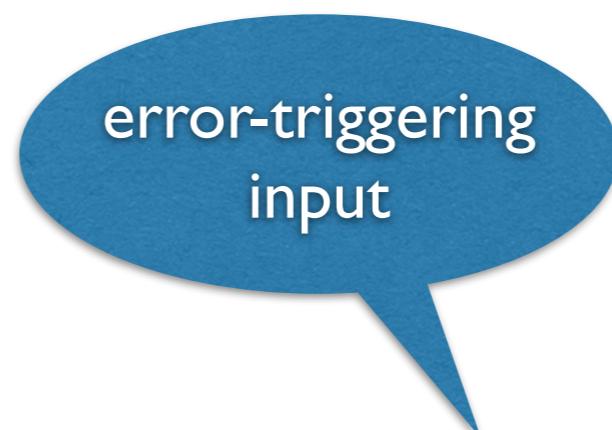
3rd iteration

cf) 콘콜릭 테스팅 (Concolic Testing)

```
int double (int v) {  
    return 2*v;  
}
```

```
void testme(int x, int y) {  
  
    z := double (y);  
  
    if (z==x) {  
  
        if (x>y+10) {  
            Error; ←  
        }  
    }  
}
```

Concrete
State



x=30, y=15,
z=30

Symbolic
State

x=a, y=β, z=2*β
 $2^*\beta = a \wedge$
 $a > \beta + 10$

3rd iteration

cf) Non-Linear Constraints

```
int double (int v) {  
    return v*v;  
}  
  
void testme(int x, int y) {  
    z := double (y);  
    if (z==x) {  
        if (x>y+10) {  
            Error;  
        }  
    }  
}
```

Concrete State	Symbolic State
$x=22, y=7, z=49$	Solve: $\beta^*\beta = a \wedge \beta=7$ Solution: $a=49, \beta=7$ $x=a, y=\beta, z=\beta^*\beta$ $\beta^*\beta \neq a$

기호 실행 적용 사례

Benchmarks	Versions	Error Types	Bug-Triggering Inputs
vim	8.1*	Non-termination	K1!100010010011110(
	5.7	Abnormal-termination	H:w>> ``\ [press ‘Enter’]
		Segmentation fault	=ipI\~-9~q0qw
gawk	4.2.1*	Memory-exhaustion	' +E_Q\$h+w\$8==++\$6E8# '
	3.0.3	Abnormal-termination	' f[][][][],[y]^/#['
		Non-termination	' \$g?E2^=-E-2"?^+\$=: /#/["'
grep	3.1*	Abnormal-termination	' \(\)\1*?*?\ \W*\1W* '
		Segmentation fault	' \(\)\1^*@*\? \1*\+*\? '
	2.2	Segmentation fault	" _^*9\ ^ \(\)\1*\\$"
sed		Non-termination	' \({****}\)*\+*\1*\+'
	1.17	Segmentation fault	'{ : }; :C;b'

See “Concolic Testing with Adaptively Changing Search Heuristics. FSE 2019”

```
vagrant@ubuntu-bionic:~/swtest$ grep --version
grep (GNU grep) 3.1
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

Written by Mike Haertel and others, see <<http://git.sv.gnu.org/cgit/grep.git/tree/AUTHORS>>.

```
vagrant@ubuntu-bionic:~/swtest$ time grep '\(\)\1*?*?\|\W*\1\W*'
grep: regexec.c:1344: pop_fail_stack: Assertion `num >= 0' failed.
Aborted (core dumped)
```

```
real 0m0.074s
user 0m0.001s
sys  0m0.000s
```

```
vagrant@ubuntu-bionic:~/swtest$ time grep '\(\)\1^*@\*?\|1*\+*\?'
Segmentation fault (core dumped)
```

```
real 0m10.975s
user 0m10.672s
sys  0m0.239s
```

산업체 적용 사례

DOI:10.1145/2093548.2093564

Article development led by **acmqueue**
queue.acm.org

SAGE has had a remarkable impact at Microsoft.

BY PATRICE GODEFROID, MICHAEL Y. LEVIN, AND DAVID MOLNAR

SAGE: Whitebox Fuzzing for Security Testing

MOST COMMUNICATIONS READERS might think of “program verification research” as mostly theoretical with little impact on the world at large. Think again. If you are reading these lines on a PC running some form of Windows (like over 93% of PC users—that is, more than one billion people), then you have been affected by this line of work—without knowing it, which is precisely the way we want it to be.

Every second Tuesday of every month, also known as “Patch Tuesday,” Microsoft releases a list of security bulletins and associated security patches to be deployed on hundreds of millions of machines worldwide. Each security bulletin costs Microsoft

and its users millions of dollars. If a monthly security update costs you \$0.001 (one tenth of one cent) in just electricity or loss of productivity, then this number multiplied by one billion people is \$1 million. Of course, if malware were spreading on your machine, possibly leaking some of your private data, then that might cost you much more than \$0.001. This is why we strongly encourage you to apply those pesky security updates.

Many security vulnerabilities are a result of programming errors in code for parsing files and packets that are transmitted over the Internet. For example, Microsoft Windows includes parsers for hundreds of file formats.

If you are reading this article on a computer, then the picture shown in Figure 1 is displayed on your screen after a jpg parser (typically part of your operating system) has read the image data, decoded it, created new data structures with the decoded data, and passed those to the graphics card in your computer. If the code implementing that jpg parser contains a bug such as a buffer overflow that can be triggered by a corrupted jpg image, then the execution of this jpg parser on your computer could potentially be hijacked to execute some other code, possibly malicious and hidden in the jpg data itself.

This is just one example of a possible security vulnerability and attack scenario. The security bugs discussed throughout the rest of this article are mostly buffer overflows.

Hunting for “Million-Dollar” Bugs
Today, hackers find security vulnerabilities in software products using two primary methods. The first is code inspection of binaries (with a good disassembler, binary code is like source code).

The second is *blackbox fuzzing*, a form of blackbox random testing, which randomly mutates well-formed program inputs and then tests the program with those modified inputs,³ hoping to trigger a bug such as a buf-

Symbolic Execution for Software Testing in Practice – Preliminary Assessment

Cristian Cadar
Imperial College London
c.cadar@imperial.ac.uk

Patrice Godefroid
Microsoft Research
pg@microsoft.com

Sarfraz Khurshid
U. Texas at Austin
khurshid@ece.utexas.edu

Corina S. Păsăreanu*
CMU/NASA Ames
corina.s.pasareanu@nasa.gov

Koushik Sen
U.C. Berkeley
ksen@eecs.berkeley.edu

Nikolai Tillmann
Microsoft Research
nikolait@microsoft.com

Willem Visser
Stellenbosch University
visserw@sun.ac.za

ABSTRACT

We present results for the “Impact Project Focus Area” on the topic of symbolic execution as used in software testing. Symbolic execution is a program analysis technique introduced in the 70s that has received renewed interest in recent years, due to algorithmic advances and increased availability of computational power and constraint solving technology. We review classical symbolic execution and some modern extensions such as generalized symbolic execution and dynamic test generation. We also give a preliminary assessment of the use in academia, research labs, and industry.

Categories and Subject Descriptors

D.2.5 [Testing and Debugging]: Symbolic execution

General Terms

Reliability

Keywords

Generalized symbolic execution, dynamic test generation

1. INTRODUCTION

The ACM-SIGSOFT Impact Project is documenting the impact that software engineering research has had on software development practice. In this paper, we present preliminary results for documenting the impact of research in symbolic execution for automated software testing. Symbolic execution is a program analysis technique that was introduced in the 70s [8, 15, 31, 35, 46], and that has found renewed interest in recent years [9, 12, 13, 28, 29, 32, 33, 40, 42, 43, 50–52, 56, 57].

*We thank Matt Dwyer for his advice

Symbolic execution is now the underlying technique of several popular testing tools, many of them open-source: NASA’s Symbolic (Java) PathFinder¹, UIUC’s CUTE and jCUTE², Stanford’s KLEE³, UC Berkeley’s CREST⁴ and BitBlaze⁵, etc. Symbolic execution tools are now used in industrial practice at Microsoft (Pex⁶, SAGE [29], YOGI⁷ and PREfix [10]), IBM (Apollo [2]), NASA and Fujitsu (Symbolic PathFinder), and also form a key part of the commercial testing tool suites from Parasoft and other companies [60].

Although we acknowledge that the impact of symbolic execution in software practice is still limited, we believe that the explosion of work in this area over the past years makes for an interesting story about the increasing impact of symbolic execution since it was first introduced in the 1970s. Note that this paper is not meant to provide a comprehensive survey of symbolic execution techniques; such surveys can be found elsewhere [19, 44, 49]. Instead, we focus here on a few modern symbolic execution techniques that have shown promise to impact software testing in practice.

Software testing is the most commonly used technique for validating the quality of software, but it is typically a mostly manual process that accounts for a large fraction of software development and maintenance. Symbolic execution is one of the many techniques that can be used to automate software testing by automatically generating test cases that achieve high coverage of program executions.

Symbolic execution is a program analysis technique that executes programs with symbolic rather than concrete inputs and maintains a *path condition* that is updated whenever a branch instruction is executed, to encode the constraints on the inputs that reach that program point. Test generation is performed by solving the collected constraints using a constraint solver. Symbolic execution can also be used for bug finding, where it checks for run-time errors or assertion violations and it generates test inputs that trigger those errors.

The original approaches to symbolic execution [8, 15, 31, 35,

¹<http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc>

²<http://osl.cs.uiuc.edu/~ksen/cute/>

³<http://klee.llvm.org/>

⁴<http://code.google.com/p/crest/>

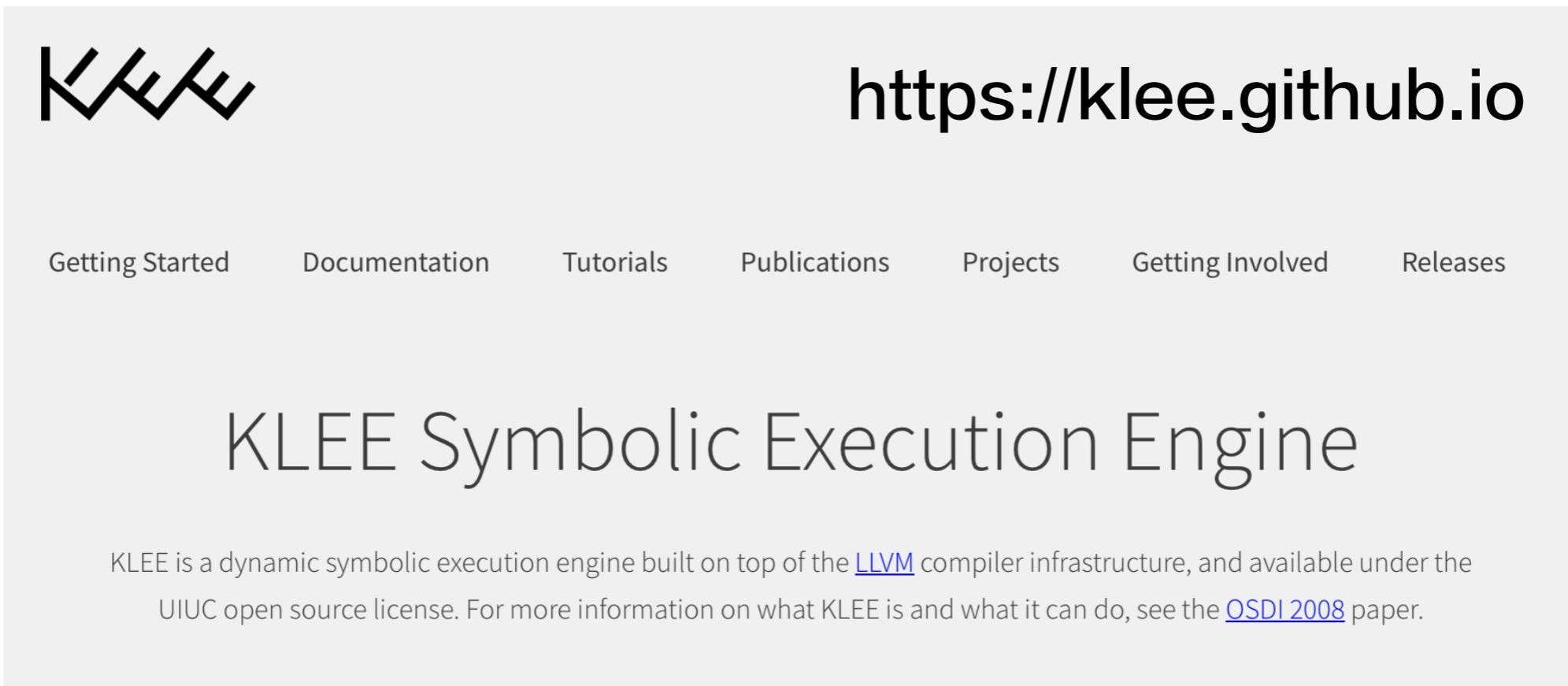
⁵<http://bitblaze.cs.berkeley.edu/>

⁶<http://research.microsoft.com/en-us/projects/pex/>

⁷<http://research.microsoft.com/en-us/projects/yogi/>

기호 실행기 데모: KLEE

- Open-source symbolic execution engine
- Demo: <http://klee.doc.ic.ac.uk>



The screenshot shows the GitHub repository page for KLEE. At the top left is the KLEE logo, which consists of four black diagonal lines forming a stylized 'K'. To the right of the logo is the URL <https://klee.github.io>. Below the URL is a navigation bar with links: Getting Started, Documentation, Tutorials, Publications, Projects, Getting Involved, and Releases. The main title "KLEE Symbolic Execution Engine" is centered below the navigation bar. A descriptive paragraph explains that KLEE is a dynamic symbolic execution engine built on top of the LLVM compiler infrastructure, available under the UIUC open source license, and provides a link to the OSDI 2008 paper.

KLEE is a dynamic symbolic execution engine built on top of the [LLVM](#) compiler infrastructure, and available under the UIUC open source license. For more information on what KLEE is and what it can do, see the [OSDI 2008](#) paper.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void crash() {
5     char* x = (char*)malloc(1);
6     free(x);
7     free(x);
8 }
9
10 void testme(int x, int y) {
11     int z;
12     z = y * 2;
13
14     if (z==x) {
15
16         if (x>y+10) {
17             crash();
18         }
19     }
20 }
21
22 int main(int argc, char** argv)
23 {
24     int a, b;
25     klee_make_symbolic(&a, sizeof(a), "a");
26     klee_make_symbolic(&b, sizeof(b), "b");
27     testme(a, b);
28     return 0;
29 }
```

KLEE RESULTS

OUTPUT

STATS

Job queued!
 Executing KLEE
 Executing KLEE
 Uploading KLEE output directory
 Done!

Ran command "/home/klee/klee_build/bin/klee /tmp/code/code.o".

KLEE: output directory is "/tmp/code/klee-out-0"
 KLEE: Using STP solver backend
 KLEE: WARNING ONCE: Alignment of memory from call "malloc" is not modelled. Using alignment of 8.
 KLEE: ERROR: /tmp/code/code.c:7: memory error: invalid pointer: free
 KLEE: NOTE: now ignoring this error at this location

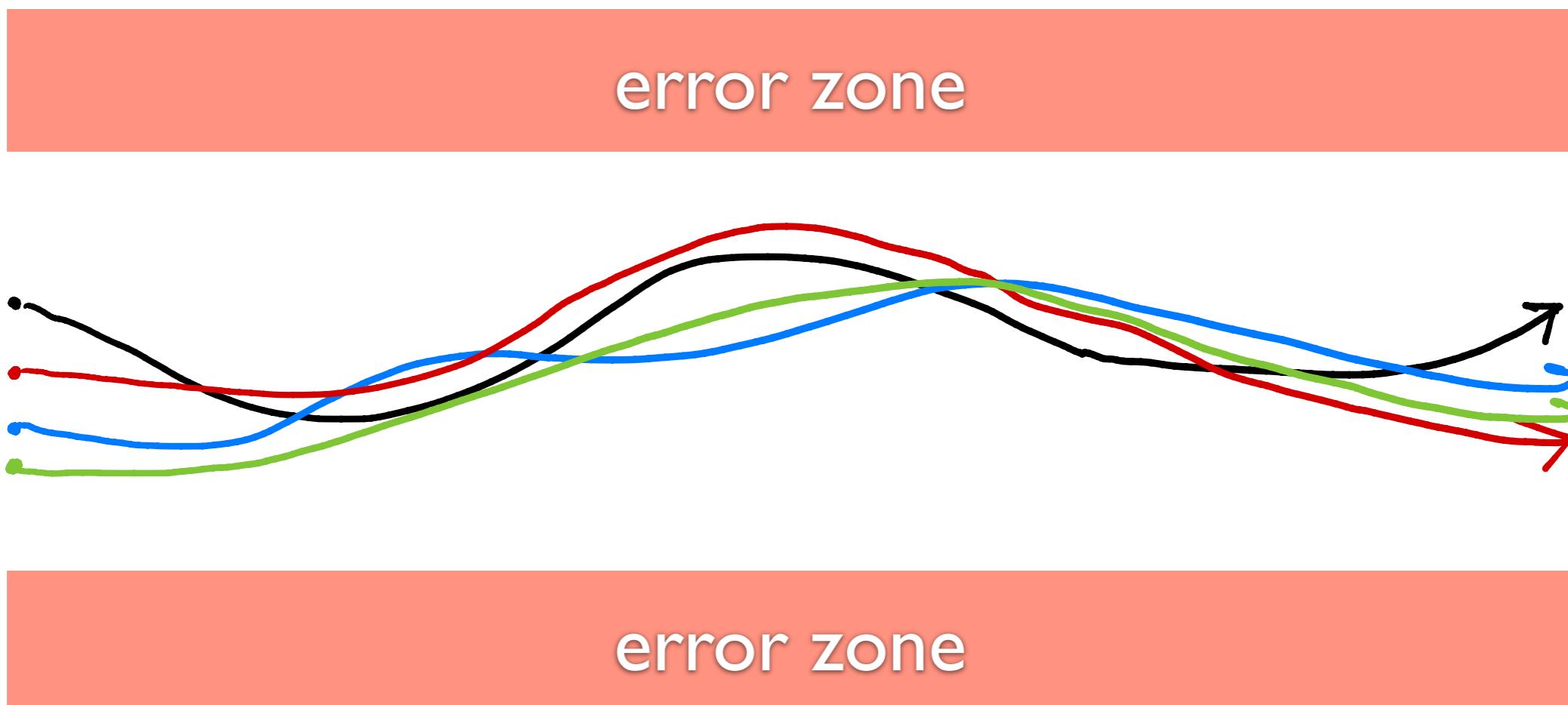
KLEE: done: total instructions = 51
 KLEE: done: completed paths = 3
 KLEE: done: generated tests = 3

Failed tests:

Memory error on line 7.
 free(x);

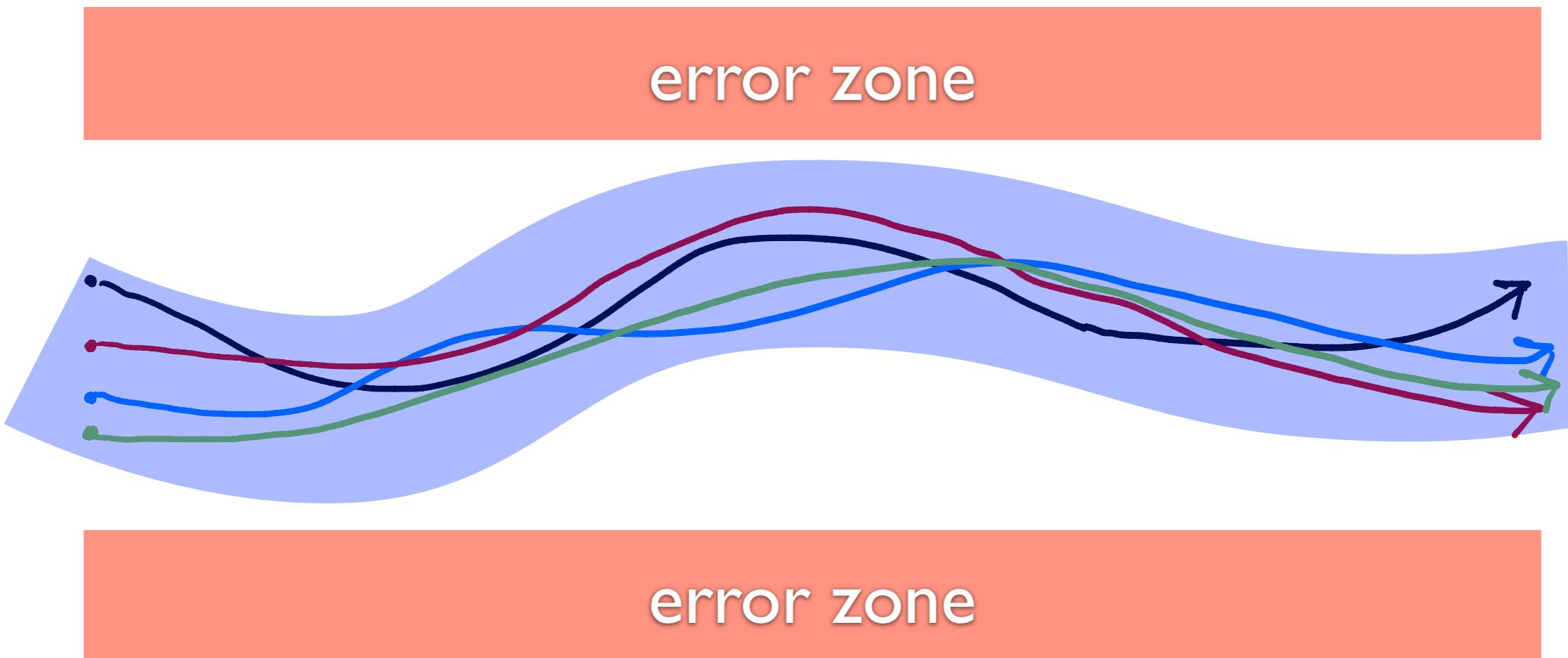
정적 분석 원리

- 프로그램 실행을 요약(abstraction)하여 실행



정적 분석 원리

- 프로그램 실행을 요약(abstraction)하여 실행



정적 분석 원리

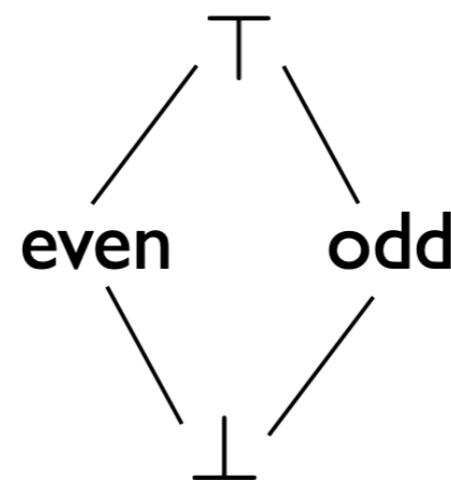
$$30 \times 12 + 11 \times 9 = ?$$

정적 분석 원리

```
void f (int x) {  
    y = x * 12 + 9 * 11;  
    assert (y % 2 == 1);  
}
```

요약 값 (Abstract Values)

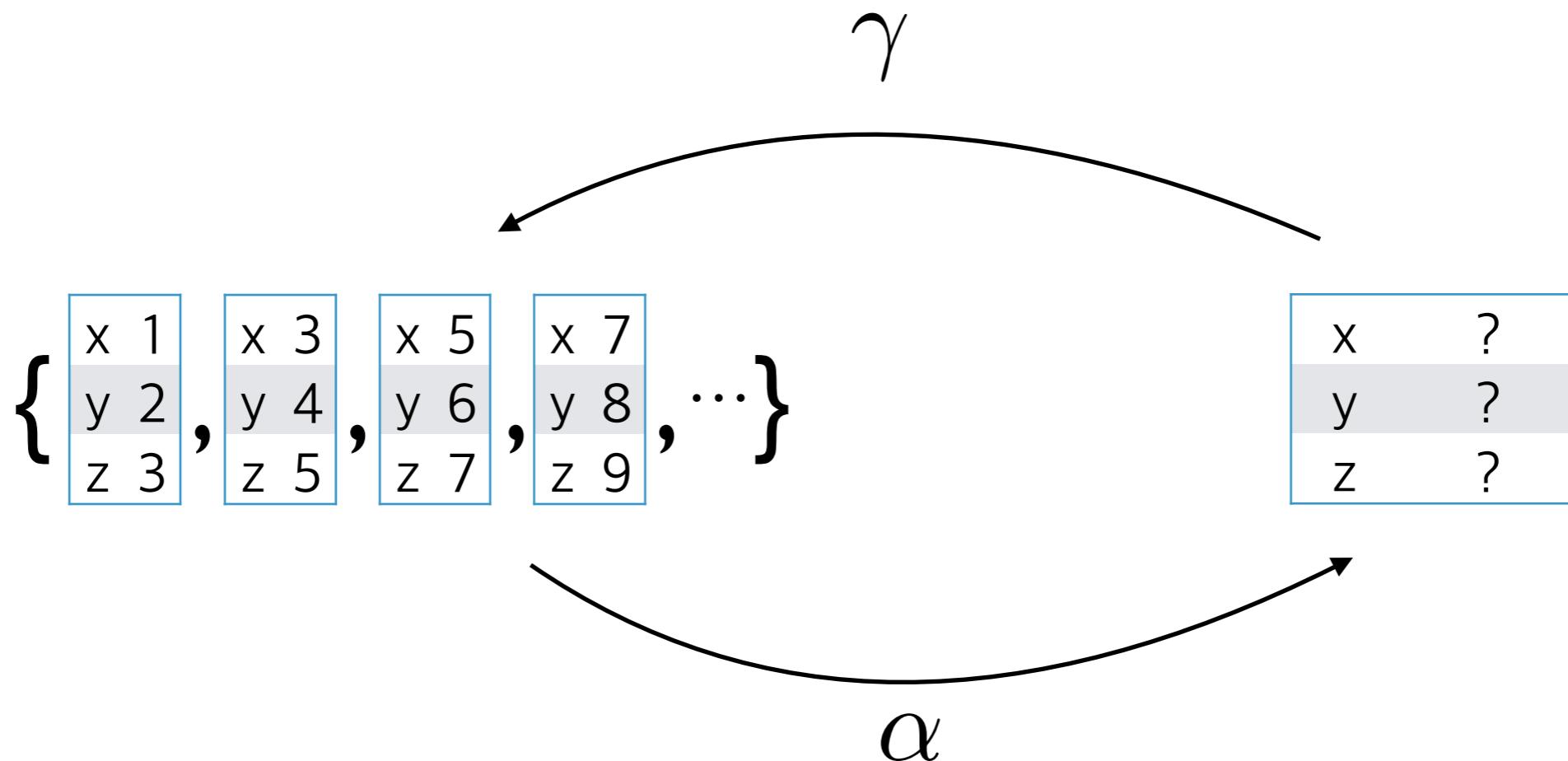
- Simple parity domain:



- Abstraction (α) and concretization (γ) functions:
- Join operation:

요약 상태 (Abstract States)

- Memory state based on simple parity domain:



- Join operation:

요약 의미 (Abstract Semantics)

- 홀짝 공간에서의 덧셈, 뺄셈, 곱셈:

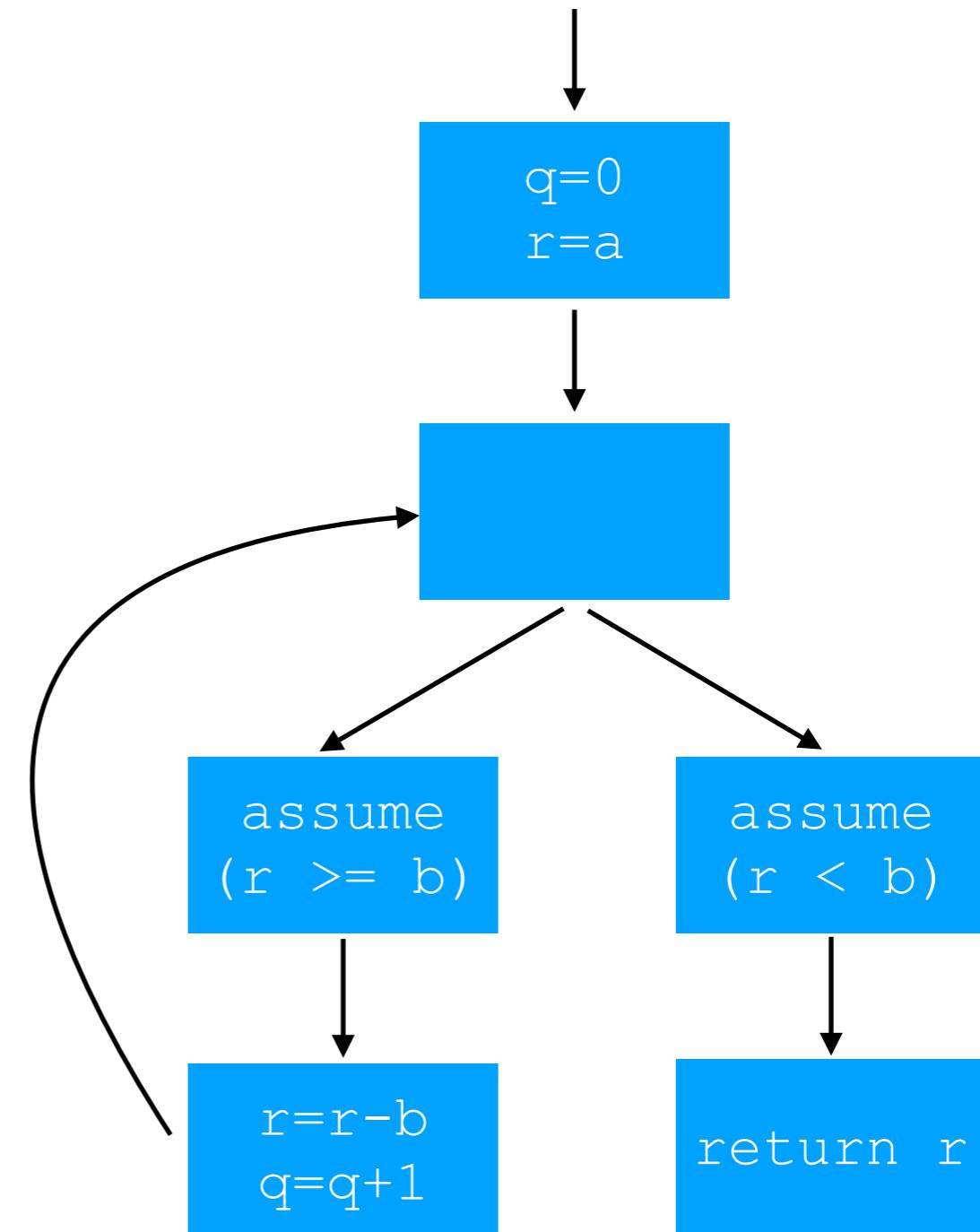
+	top	even	odd	bottom
top				
even				
odd				
bottom				

-	top	even	odd	bottom
top				
even				
odd				
bottom				

*	top	even	odd	bottom
top				
even				
odd				
bottom				

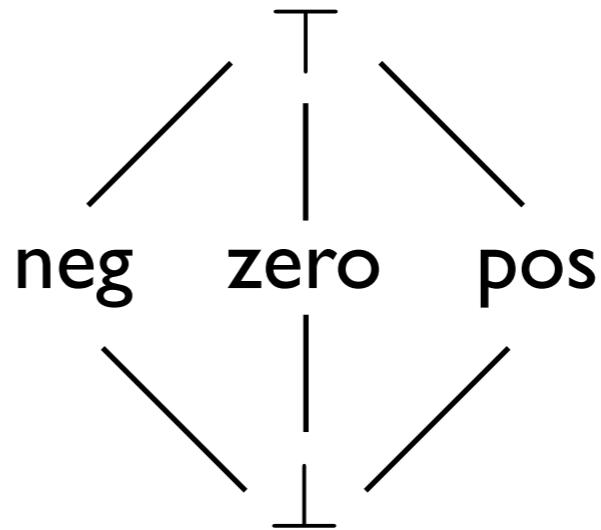
고정점 계산 (Fixpoint Computation)

```
// a >= 0, b >= 0
int mod (int a, int b) {
    int q = 0;
    int r = a;
    while (r >= b) {
        r = r - b;
        q = q + 1;
    }
    return r;
}
```



요약 값: 부호 분석

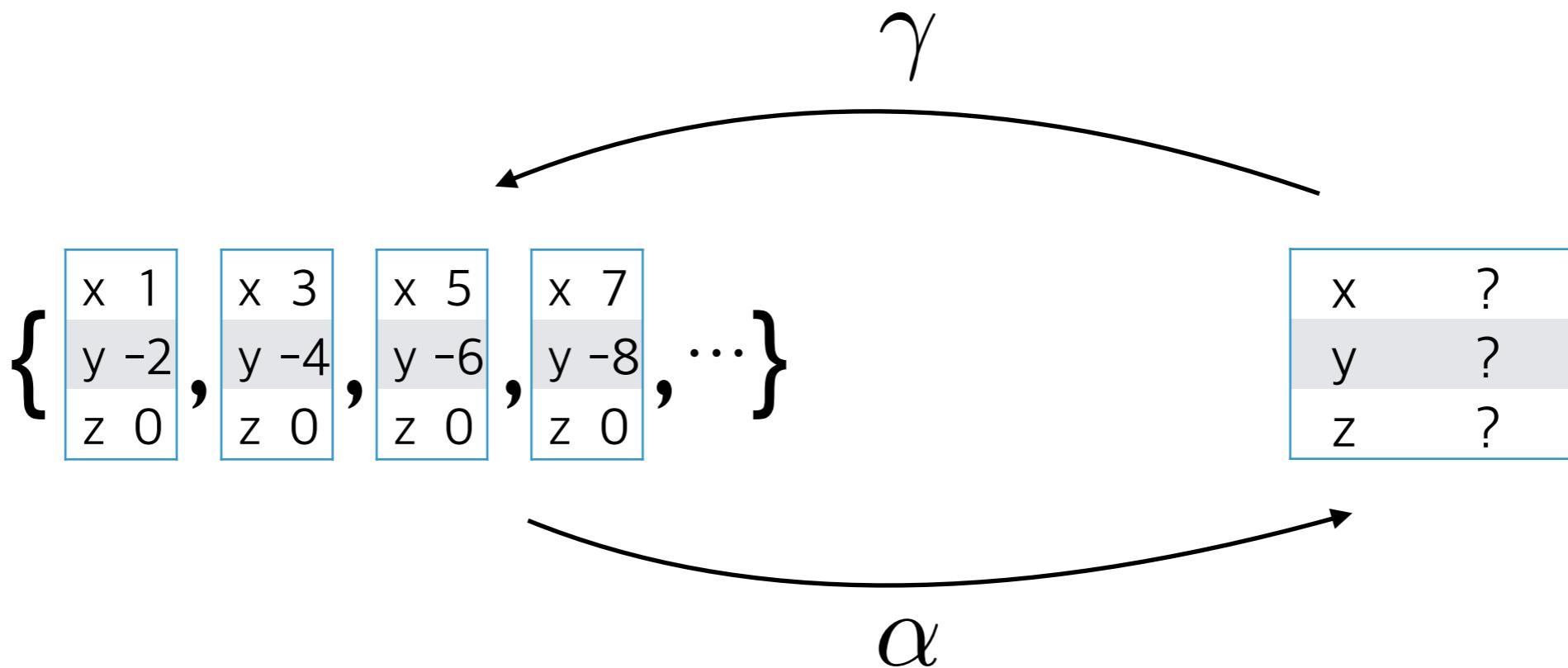
- Simple sign domain:



- Abstraction (α) and concretization (γ) functions:
- Join operation:

요약 상태: 부호 분석

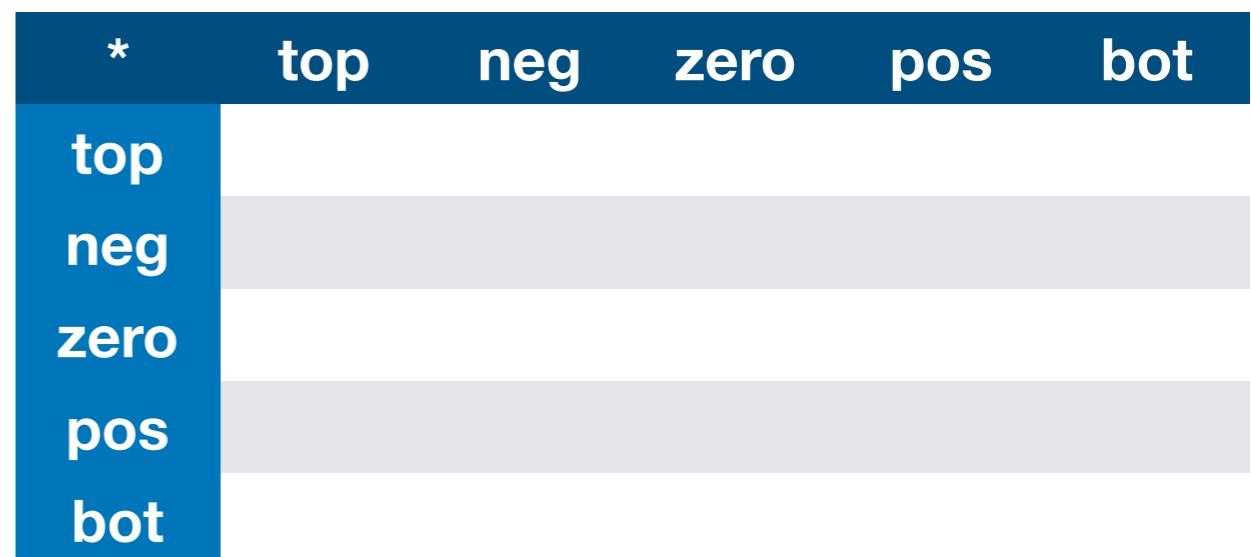
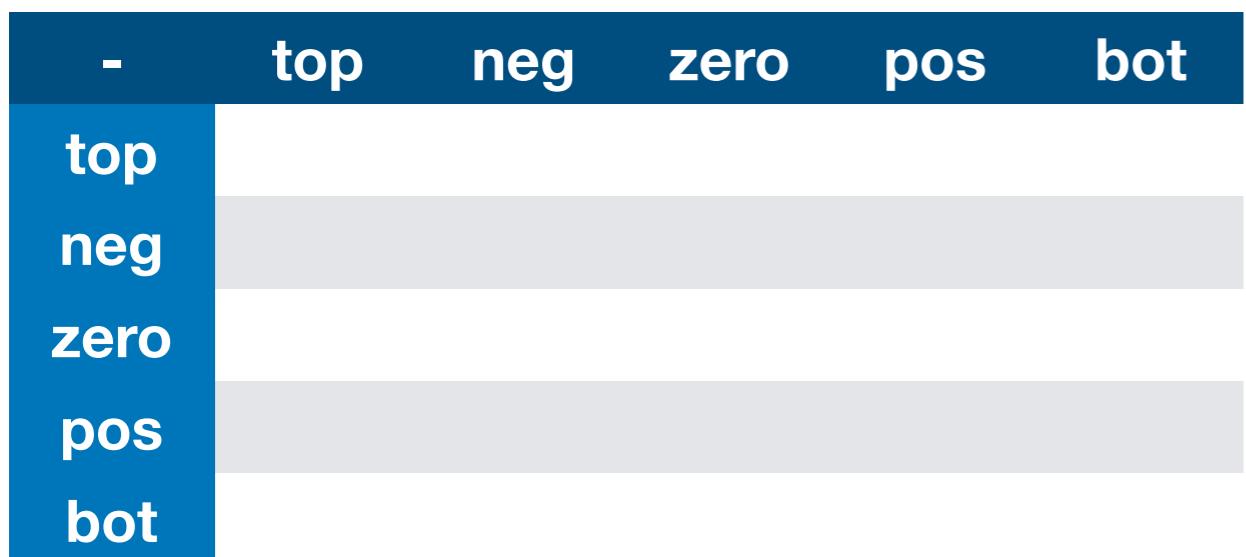
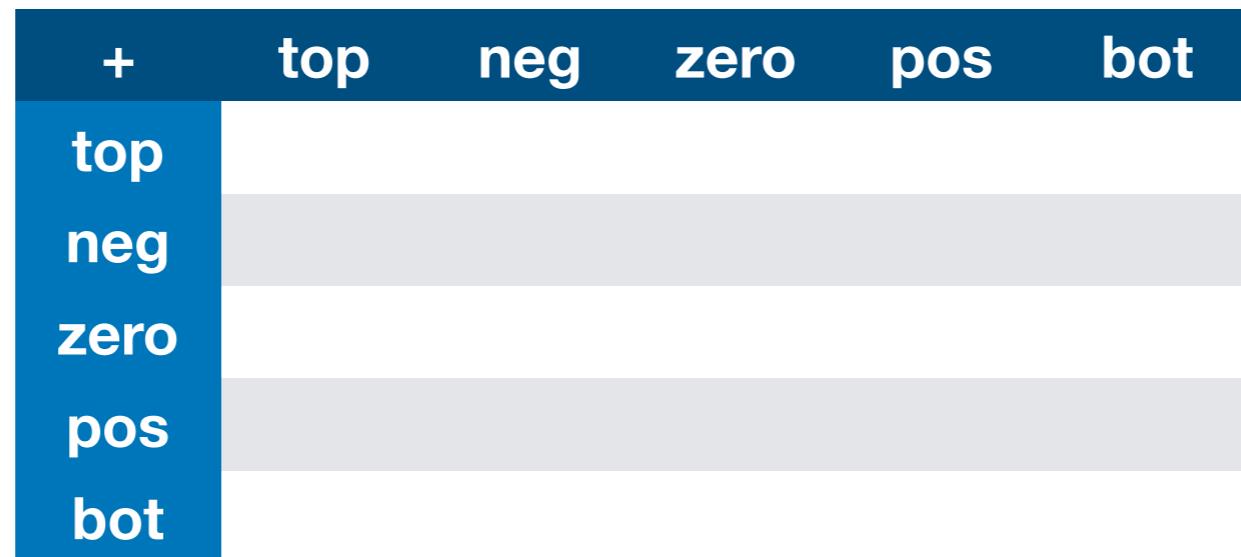
- Memory state abstraction:



- Join operation:

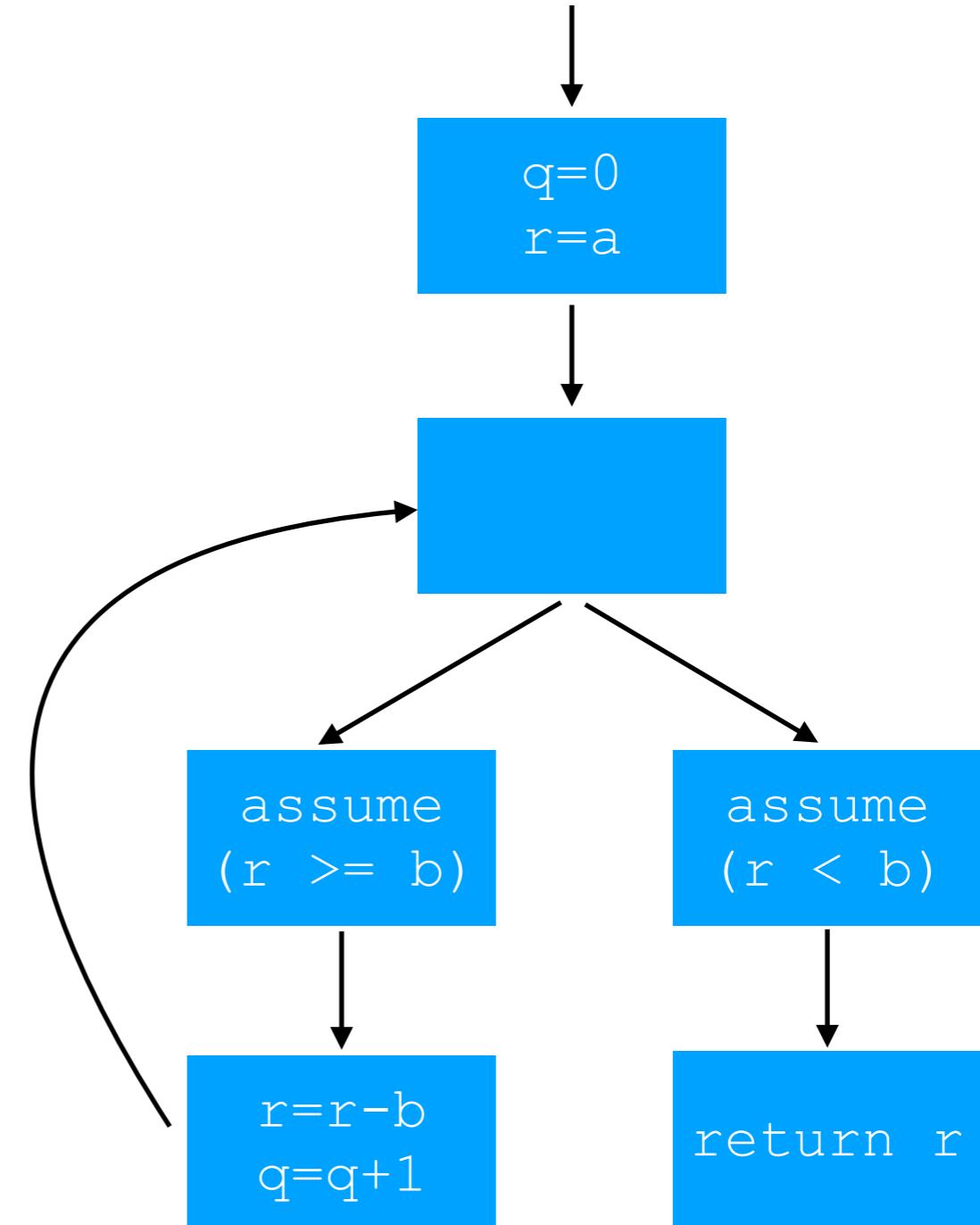
요약 의미: 부호 분석

- 덧셈, 뺄셈, 곱셈:

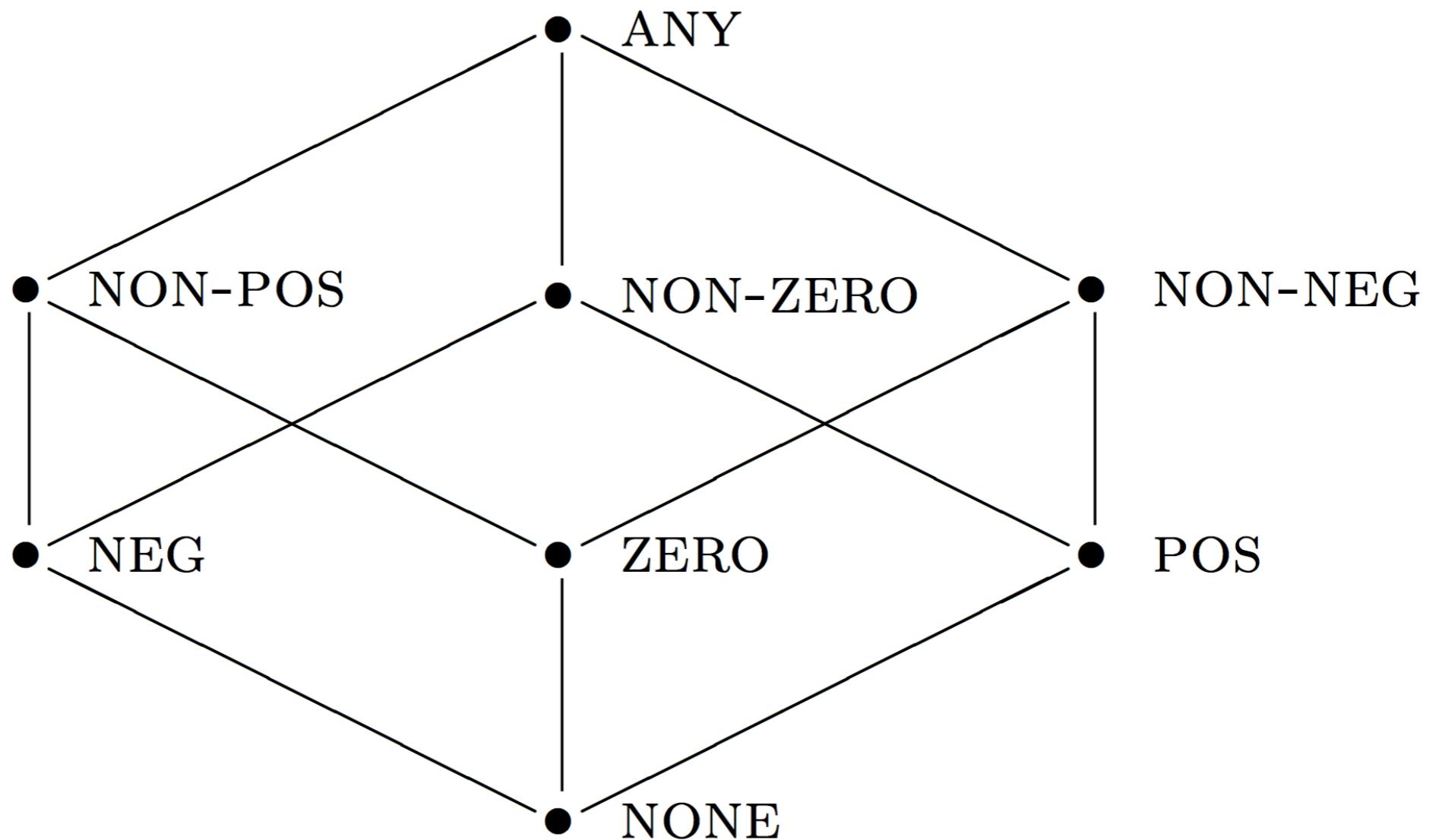


고정점 계산: 부호 분석

```
// a >= 0, b >= 0
int mod (int a, int b) {
    int q = 0;
    int r = a;
    while (r >= b) {
        r = r - b;
        q = q + 1;
    }
    return r;
}
```



부호 분석: 더 정교히

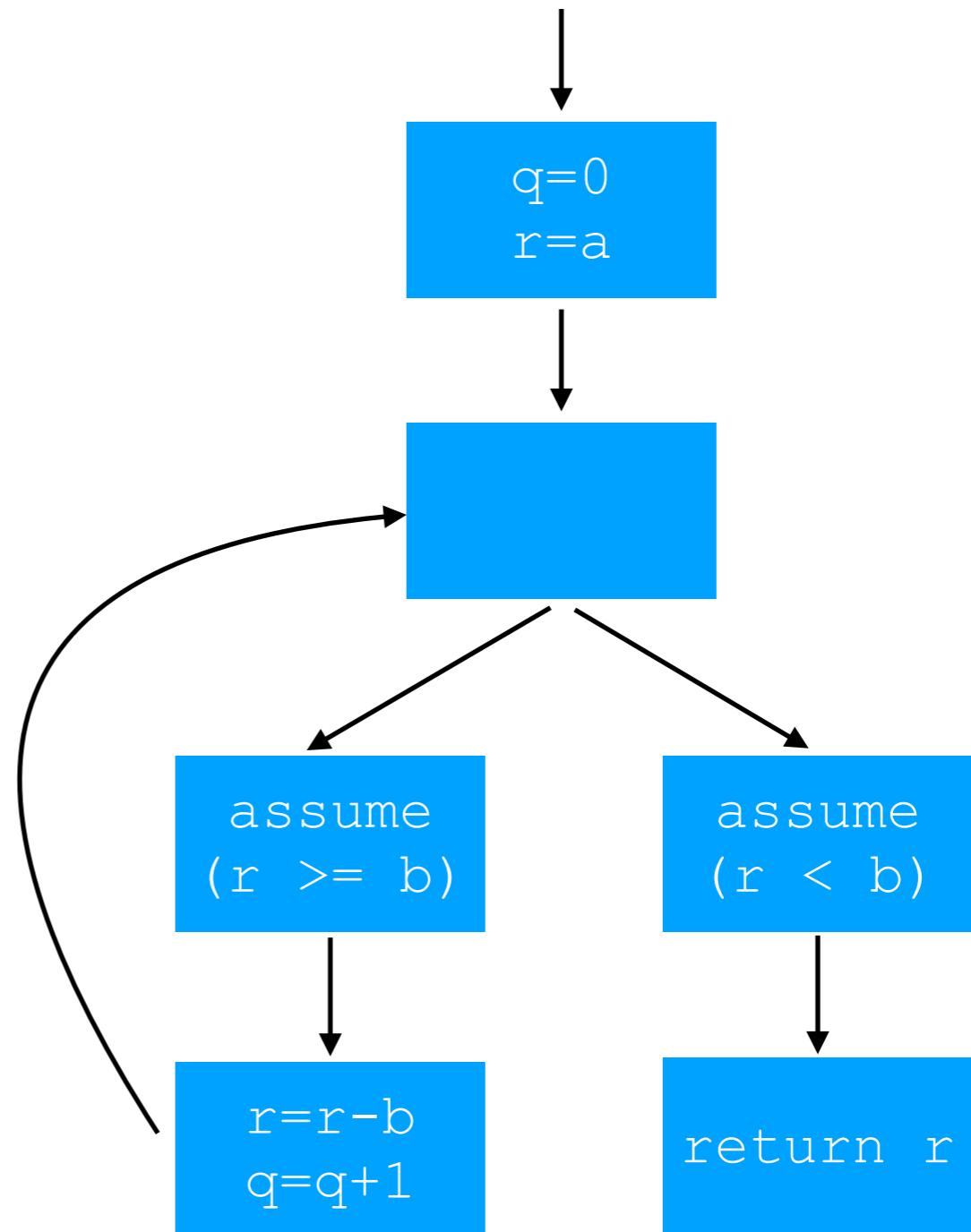


$+S$	NONE	NEG	ZERO	POS	NON-POS	NON-ZERO	NON-NEG	ANY
NONE	NONE	NONE	NONE	NONE	NONE	NONE	NONE	NONE
NEG	NONE	NEG	NEG	ANY	NEG	ANY	ANY	ANY
ZERO	NONE	POS	ZERO	POS	NON-POS	NON-ZERO	NON-NEG	ANY
POS	NONE	ANY	POS	POS	ANY	ANY	POS	ANY
NON-POS	NONE	NEG	NON-POS	ANY	NON-POS	ANY	ANY	ANY
NON-ZERO	NONE	ANY	NON-ZERO	ANY	ANY	ANY	ANY	ANY
NON-NEG	NONE	ANY	NON-NEG	POS	ANY	ANY	NON-NEG	ANY
ANY	NONE	ANY	ANY	ANY	ANY	ANY	ANY	ANY

$\star S$	NEG	ZERO	POS	$-S$	NEG	ZERO	POS
NEG	POS	ZERO	NEG	NEG	ANY	NEG	NEG
ZERO	ZERO	ZERO	ZERO	ZERO	POS	ZERO	NEG
POS	NEG	ZERO	POS	POS	POS	POS	ANY

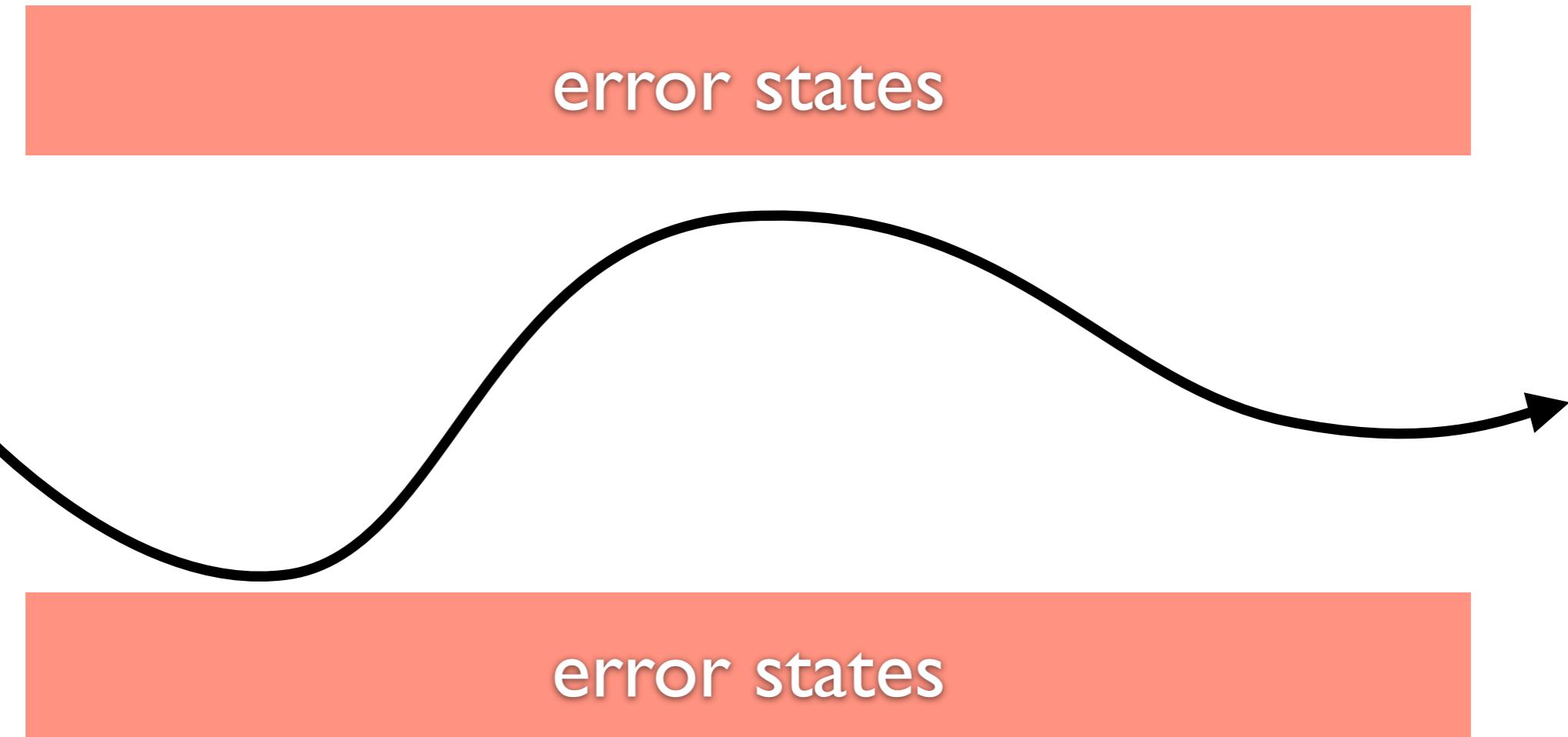
Exercise

```
// a >= 0, b >= 0
int mod (int a, int b) {
    int q = 0;
    int r = a;
    while (r >= b) {
        r = r - b;
        q = q + 1;
    }
    return r;
}
```



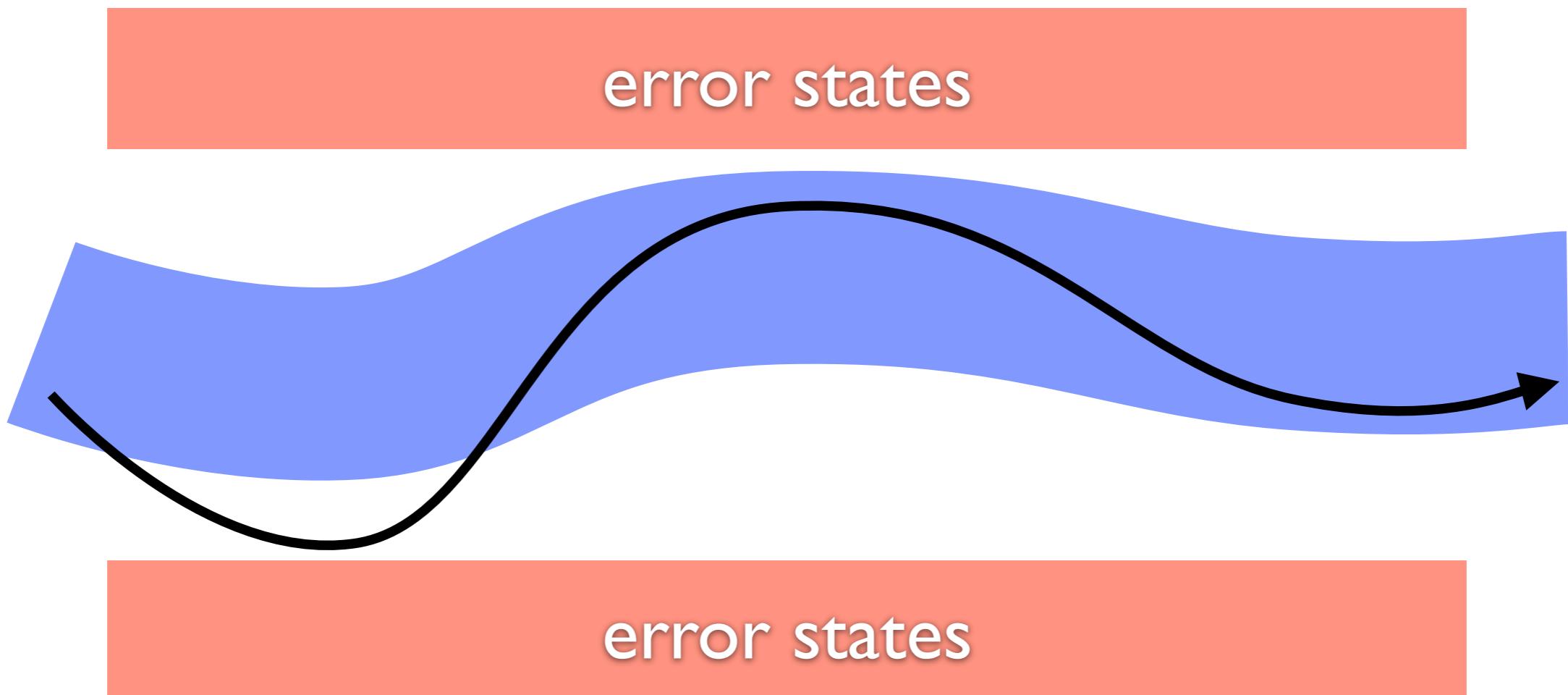
cf) 정적 버그 검출기

- Bug-finding static analysis



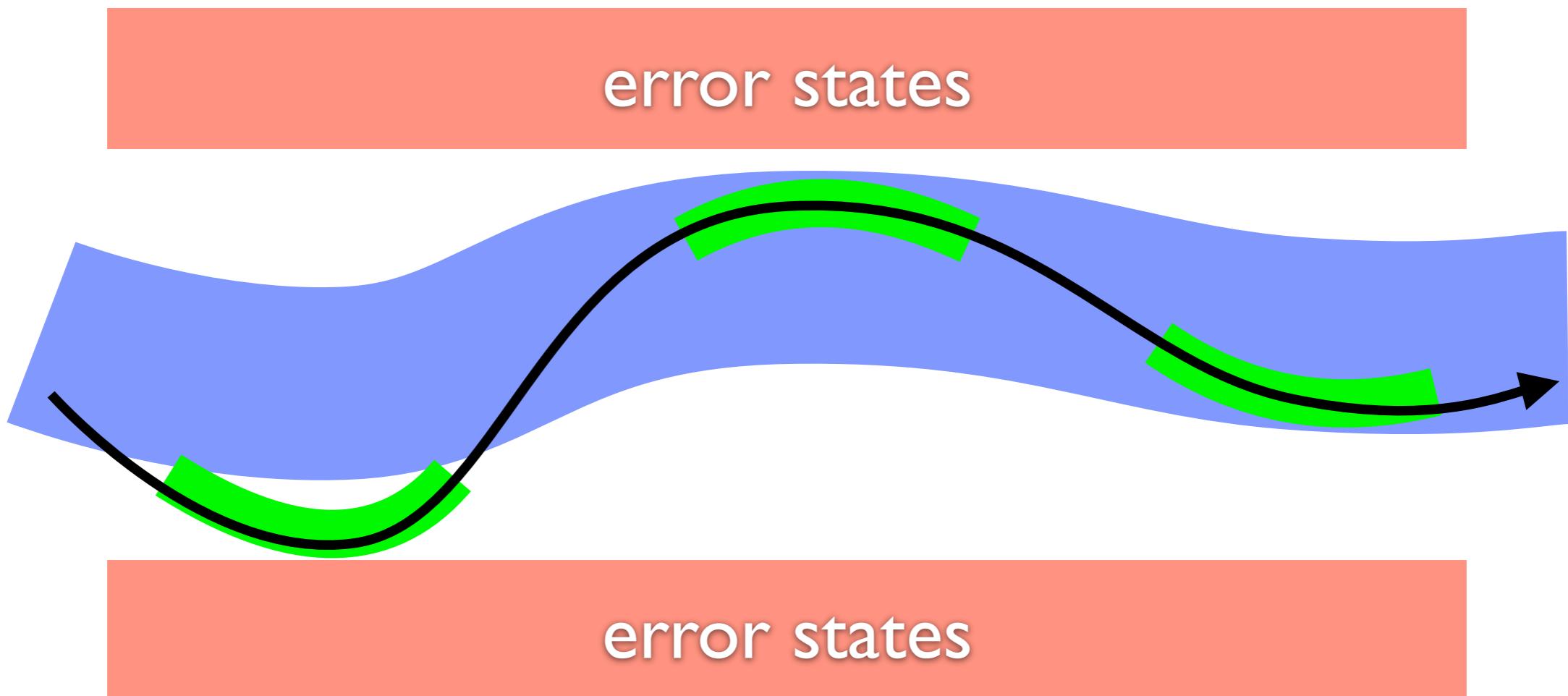
cf) 정적 버그 검출기

- Bug-finding static analysis



cf) 정적 버그 검출기

- Bug-finding static analysis



산업체 적용 사례

DOI:10.1145/3338112

Key lessons for designing static analyses tools deployed to find bugs in hundreds of millions of lines of code.

BY DINO DISTEFANO, MANUEL FÄHNDRICH,
FRANCESCO LOGOZZO, AND PETER W. O'HEARN

Scaling Static Analyses at Facebook



STATIC ANALYSIS TOOLS are programs that examine, and attempt to draw conclusions about, the source of other programs without running them. At Facebook, we have been investing in advanced static analysis tools that employ reasoning techniques similar to those from program verification. The tools we describe in this article (Infer and Zoncolan) target issues related to crashes and to the security of our services, they perform sometimes complex reasoning spanning many procedures or files, and they are integrated into engineering workflows in a way that attempts to bring value while minimizing friction.

These tools run on code modifications, participating as bots during the code review process. Infer targets our mobile apps as well as our backend C++ code, codebases with 10s of millions of lines; it has seen over 100 thousand reported issues fixed by developers before code reaches production. Zoncolan targets the 100-million lines of Hack code, and is additionally

integrated in the workflow used by security engineers. It has led to thousands of fixes of security and privacy bugs, outperforming any other detection method used at Facebook for such vulnerabilities. We will describe the human and technical challenges encountered and lessons we have learned in developing and deploying these analyses.

There has been a tremendous amount of work on static analysis, both in industry and academia, and we will not attempt to survey that material here. Rather, we present our rationale for, and results from, using techniques similar to ones that might be encountered at the edge of the research literature, not only simple techniques that are much easier to make scale. Our goal is to complement other reports on industrial static analysis and formal methods,^{1,6,13,17} and we hope that such perspectives can provide input both to future research and to further industrial use of static analysis.

Next, we discuss the three dimensions that drive our work: bugs that matter, people, and actioned/missed bugs. The remainder of the article describes our experience developing and deploying the analyses, their impact, and the techniques that underpin our tools.

Context for Static Analysis at Facebook

Bugs that Matter. We use static analysis to prevent bugs that would affect our products, and we rely on our engineers' judgment as well as data from production to tell us the bugs that matter the most.

» key insights

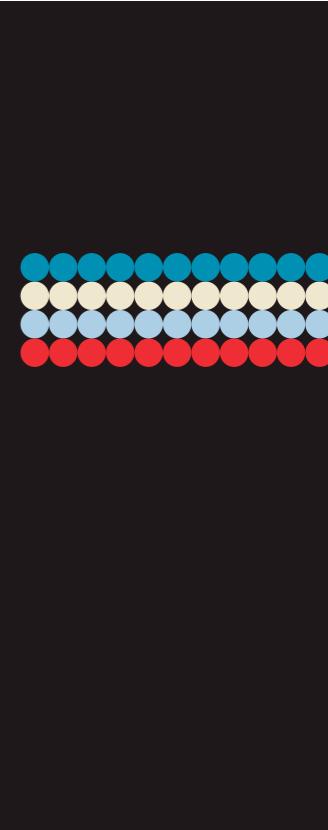
- Advanced static analysis techniques performing deep reasoning about source code can scale to large industrial codebases, for example, with 100-million LOC.
- Static analyses should strike a balance between missed bugs (false negatives) and un-actioned reports (false positives).
- A "diff time" deployment, where issues are given to developers promptly as part of code review, is important to catching bugs early and getting high fix rates.

DOI:10.1145/3188720

For a static analysis project to succeed, developers must feel they benefit from and enjoy using it.

BY CAITLIN SADOWSKI, EDWARD AFTANDILIAN, ALEX EAGLE, LIAM MILLER-CUSHON, AND CIERA JASPAK

Lessons from Building Static Analysis Tools at Google



Not integrated. The tool is not integrated into the developer's workflow or takes too long to run;

Not actionable. The warnings are not actionable;

Not trustworthy. Users do not trust the results due to, say, false positives;

Not manifest in practice. The reported bug is theoretically possible, but the problem does not actually manifest in practice;

» key insights

- Static analysis authors should focus on the developer and listen to their feedback.
- Careful developer workflow integration is key for static analysis tool adoption.
- Static analysis tools can scale by crowdsourcing analysis development.

산업체 적용 사례

WWDC (Apple Worldwide Developers Conference) 2021

The screenshot shows a video player interface for WWDC 2021. The main content area displays a Xcode interface with a static analyzer results window. The results window shows a 'Memory error' issue in 'TransNeptunianObject.m'. The code editor shows a method implementation:

```
- (nonnull SolarSystemPoint *)positionAtDate:(NSDate *)date {
    TransNeptunianObject *object = nil;
    switch (self.shape) {
        case spherical:
            object = self;
            break;
        case irregular:
            object = self.sphericalApproximation;
            break;
        default:
            break;
    }
    SolarSystemPoint *position = [object regularPositionAtDate:date];
    return position;
}
```

The static analyzer highlights several errors: 'nil returned from a method that is expected to return a non-null value' at the assignment to 'object', 'nil returned from a method that is expected to return a non-null value' at the assignment to 'position', and '1. 'object' initialized to nil' at the declaration of 'object'. A blue arrow points from the first error message to the corresponding line of code.

Below the video player, there is a summary section titled 'Detect bugs early with the static analyzer' which provides a brief overview of the tool's capabilities.

Detect bugs early with the static analyzer

Discover how Xcode can automatically track down infinite loops, unused code, and other issues before you even run your app. Learn how, with a single click, Xcode can analyze your project to discover security issues, logical bugs, and other hard-to-spot errors in Objective-C, C, and C++. We'll show you how to use the static analyzer to save you time investigating bug reports and improve your app's overall quality.

<https://developer.apple.com/videos/play/wwdc2021/10202/>

상용 정적 분석 도구들



Infer

Clang Static Analyzer

COVERITY
BY SYNOPSYS

sonarQube

PolySpace
TECHNOLOGIES

Sparrow
The Early Bird

...

The screenshot shows the GitHub repository landing page for Infer. At the top left is the Infer logo. To its right are navigation links: Docs, Support, Blog, Twitter, Facebook, GitHub, a light/dark mode switch, and a search bar. Below this is a large purple header with white text: "A tool to detect bugs in Java and C/C++/Objective-C code before it ships". A descriptive paragraph follows, explaining Infer's purpose: "Infer is a static analysis tool - if you give Infer some Java or C/C++/Objective-C code it produces a list of potential bugs. Anyone can use Infer to intercept critical bugs before they have shipped to users, and help prevent crashes or poor performance." At the bottom of the purple area are two buttons: "Get Started" and "Learn More". Below these buttons is a small stats box showing "Star" and "12,957".

<https://github.com/facebook/infer>

정적 분석기 데모: Infer

- Install (<https://github.com/facebook/infer/blob/main/INSTALL.md>)

```
# Checkout Infer
git clone https://github.com/facebook/infer.git
cd infer
# Compile Infer
./build-infer.sh java
# install Infer system-wide...
sudo make install
# ...or, alternatively, install Infer into your PATH
export PATH=`pwd`/infer/bin:$PATH
```

- Running Infer: e.g.,

- infer capture -- make
- infer analyze

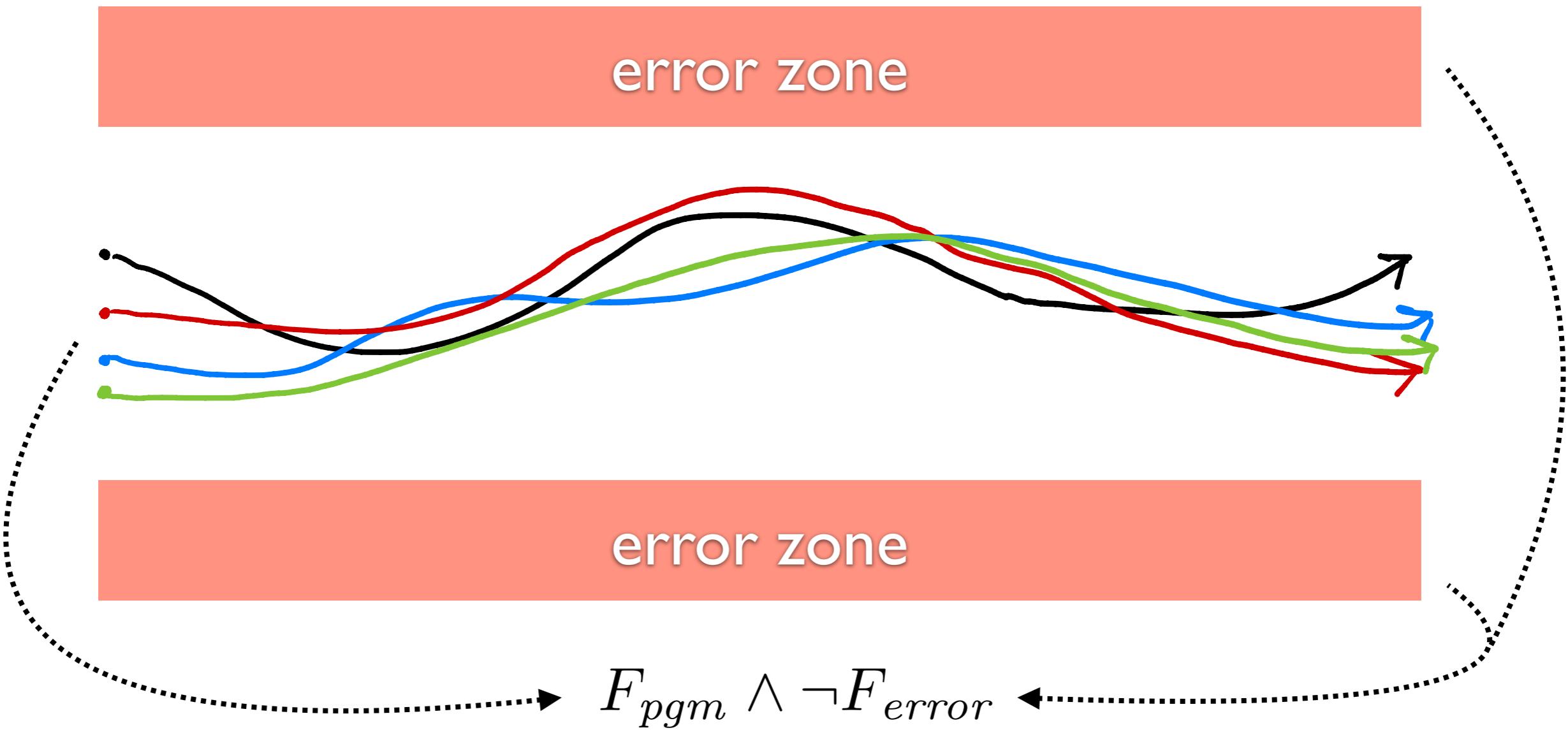
정적 분석기 데모: Infer

```
void deref_null() {
    int *p = nullptr;
    *p = 42;
}
```

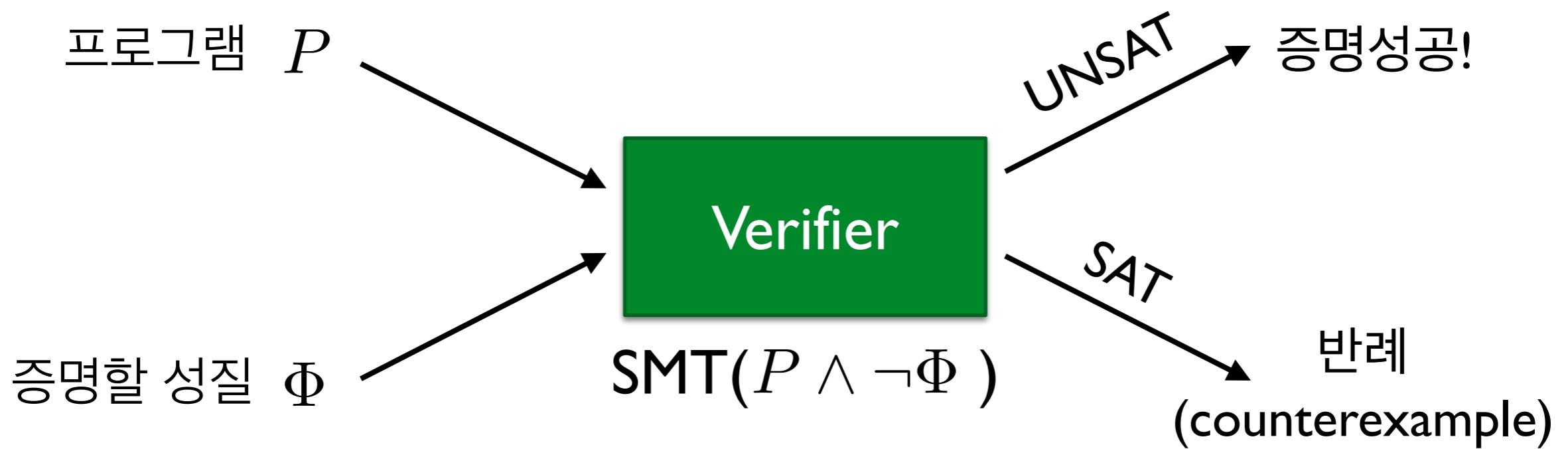
```
$ infer -- clang -c test.c
```

프로그램 검증

- 프로그램 실행 의미와 오류 조건을 논리식으로 변환



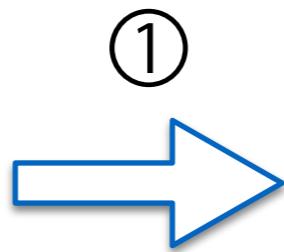
프로그램 검증



- 프로그램과 증명할 성질을 논리식으로 표현
- 논리식의 satisfiability 여부를 판별

예제

```
int f(bool a) {  
    x = 0; y = 0;  
    if (a) {  
        x = 1;  
    }  
    if (a) {  
        y = 1;  
    }  
    assert (x == y)  
}
```

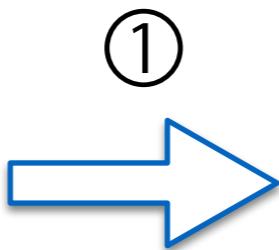


Verification Condition:

$$\begin{aligned} & ((a \wedge x) \vee (\neg a \wedge \neg x)) \wedge \\ & ((a \wedge y) \vee (\neg a \wedge \neg y)) \wedge \\ & \neg(x == y) \end{aligned}$$

예제

```
int f(bool a) {  
    x = 0; y = 0;  
    if (a) {  
        x = 1;  
    }  
    if (a) {  
        y = 1;  
    }  
    assert (x == y)  
}
```



Verification Condition:

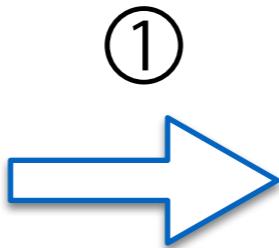
$$\begin{aligned} & ((a \wedge x) \vee (\neg a \wedge \neg x)) \wedge \\ & ((a \wedge y) \vee (\neg a \wedge \neg y)) \wedge \\ & \neg(x == y) \end{aligned}$$

②

SMT solver: unsatisfiable!

예제

```
int f(a, b) {  
    x = 0; y = 0;  
    if (a) {  
        x = 1;  
    }  
    if (b) {  
        y = 1;  
    }  
    assert (x == y)  
}
```

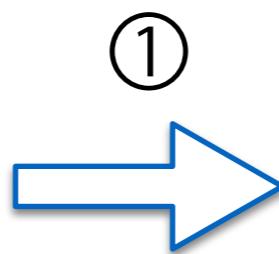


Verification Condition:

$$\begin{aligned} & ((a \wedge x) \vee (\neg a \wedge \neg x)) \wedge \\ & ((b \wedge y) \vee (\neg b \wedge \neg y)) \wedge \\ & \neg(x == y) \end{aligned}$$

예제

```
int f(a, b) {  
    x = 0; y = 0;  
    if (a) {  
        x = 1;  
    }  
    if (b) {  
        y = 1;  
    }  
    assert (x == y)  
}
```



②

Verification Condition:
 $((a \wedge x) \vee (\neg a \wedge \neg x)) \wedge$
 $((b \wedge y) \vee (\neg b \wedge \neg y)) \wedge$
 $\neg(x == y)$

SMT solver:
satisfiable when $a=1$ and $b=0$

반복문 불변 성질 (Invariant)

```
i = 0;  
j = 0;  
while  
(i < 10) {  
    i++;  
    j++;  
}  
assert (i-j==0)
```

반복문 불변 성질 (Invariant)

```
i = 0;  
j = 0;  
while @(i==j)  
(i < 10) {  
    i++;  
    j++;  
}  
assert (i-j==0)
```

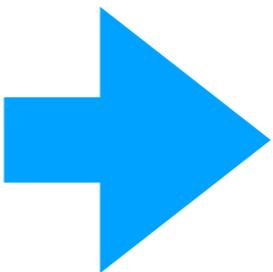
반복문 불변 성질 (Invariant)

```
i = 0;  
j = 0;  
while @(i==j)  
(i < 10) {  
    i++;  
    j++;  
}  
assert (i-j==0)
```

무한히 많은 불변 성질들 ($i \geq 0, j \geq 0, i == j, \text{true}, \dots$) 가운데 증명에 성공하는 것이 필요

프로그램 검증 기술의 장단점

```
bool LinearSearch (int a[], int l, int u, int e) {  
    int i := l;  
    while (i ≤ u) {  
        if (a[i] = e) return true  
        i := i + 1;  
    }  
    return false  
}
```

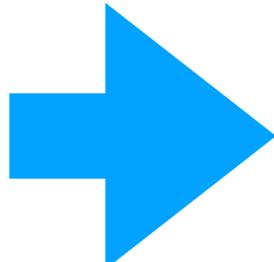


$\text{@pre : } 0 \leq l \wedge u < |a|$
 $\text{@post : } rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e$

```
bool LinearSearch (int a[], int l, int u, int e) {  
    int i := l;  
    while  
        @L : l ≤ i  $\wedge$  ( $\forall j. l \leq j < i \rightarrow a[j] \neq e$ )  
        (i ≤ u) {  
            if (a[i] = e) return true  
            i := i + 1;  
        }  
    return false  
}
```

프로그램 검증 기술의 장단점

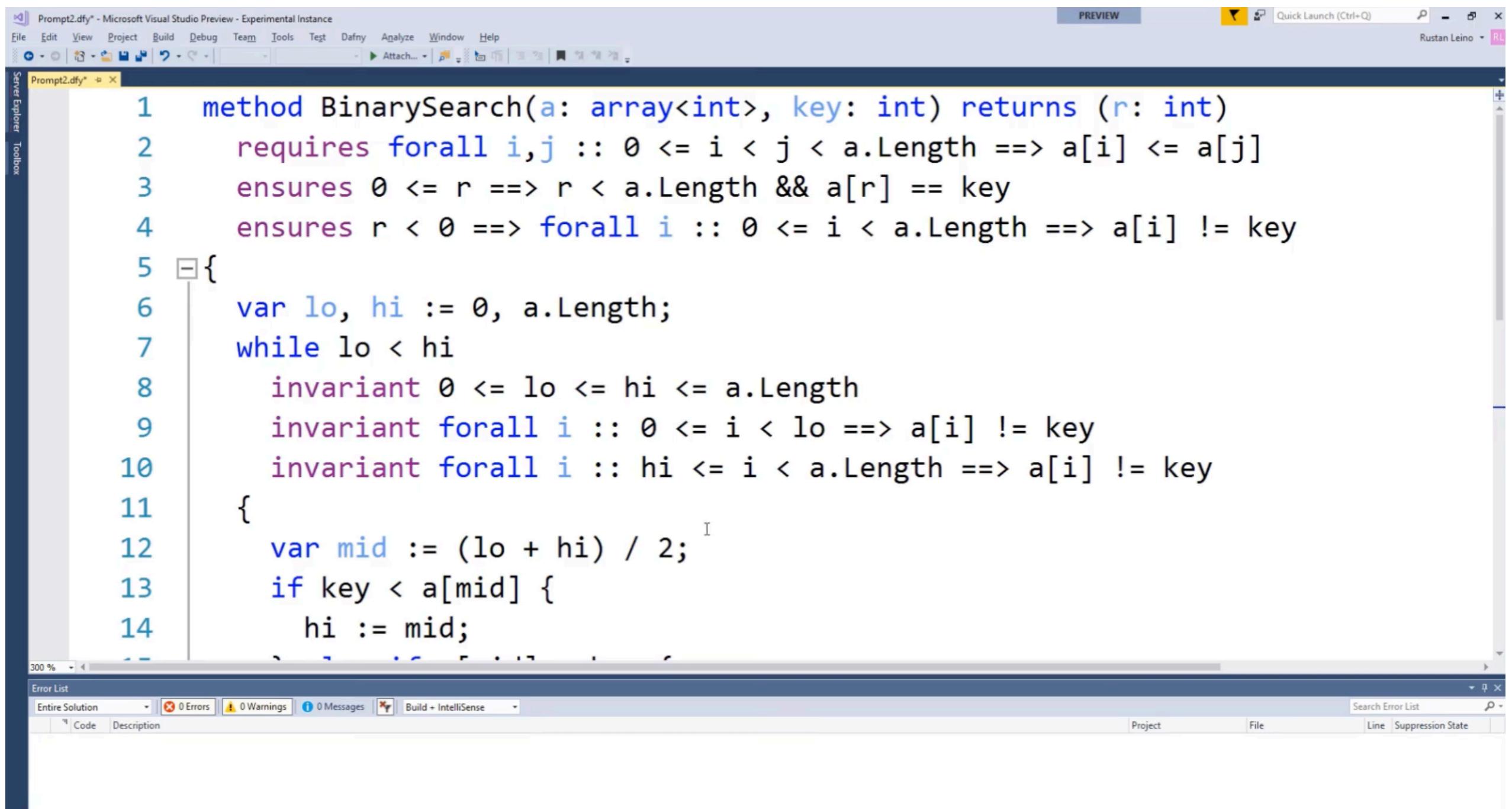
```
bool BubbleSort (int a[]) {  
    int[] a := a0  
    for (int i := |a| - 1; i > 0; i := i - 1) {  
        for (int j := 0; j < i; j := j + 1) {  
            if (a[j] > a[j + 1]) {  
                int t := a[j];  
                int a[j] := a[j + 1];  
                int a[j + 1] := t;  
            }  
        }  
    }  
    return a;  
}
```



```
@pre :  $\top$   
@post : sorted( $rv, 0, |rv| - 1$ )  
bool BubbleSort (int a[]) {  
    int[] a := a0  
    @L1 
$$\left[ \begin{array}{l} -1 \leq i < |a| \\ \wedge \text{partitioned}(a, 0, i, i + 1, |a| - 1) \\ \wedge \text{sorted}(a, i, |a| - 1) \end{array} \right]$$
  
    for (int i := |a| - 1; i > 0; i := i - 1) {  
        @L2 
$$\left[ \begin{array}{l} 1 \leq i < |a| \wedge 0 \leq j \leq i \\ \wedge \text{partitioned}(a, 0, i, i + 1, |a| - 1) \\ \wedge \text{partitioned}(a, 0, j - 1, j, j) \\ \wedge \text{sorted}(a, i, |a| - 1) \end{array} \right]$$
  
        for (int j := 0; j < i; j := j + 1) {  
            if (a[j] > a[j + 1]) {  
                int t := a[j];  
                int a[j] := a[j + 1];  
                int a[j + 1] := t;  
            }  
        }  
    }  
    return a;  
}
```

$$\text{sorted}(a, l, u) \iff \forall i, j. l \leq i \leq j \leq u \rightarrow a[i] \leq a[j]$$

사례: The Dafny Programming Language



The screenshot shows the Microsoft Visual Studio Preview interface with an experimental instance. The main window displays a Dafny program named "Prompt2.dfy". The code implements a binary search algorithm with annotations for preconditions, invariants, and postconditions.

```
1 method BinarySearch(a: array<int>, key: int) returns (r: int)
2     requires forall i,j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
3     ensures 0 <= r ==> r < a.Length && a[r] == key
4     ensures r < 0 ==> forall i :: 0 <= i < a.Length ==> a[i] != key
5 {
6     var lo, hi := 0, a.Length;
7     while lo < hi
8         invariant 0 <= lo <= hi <= a.Length
9         invariant forall i :: 0 <= i < lo ==> a[i] != key
10        invariant forall i :: hi <= i < a.Length ==> a[i] != key
11    {
12        var mid := (lo + hi) / 2;
13        if key < a[mid] {
14            hi := mid;
15        }
16        else {
17            lo := mid + 1;
18        }
19    }
20    ensures 0 <= r < a.Length && a[r] == key
21    ensures r < 0 ==> forall i :: 0 <= i < a.Length ==> a[i] != key
22    returns r
23}
```

The code uses Dafny's type system and annotations to specify the behavior of the function. It includes requirements for the input array and key, ensures clauses for the output result, and invariant statements within the loop to maintain the search space. The final ensures clause provides a guarantee about the function's correctness.

산업체 적용 사례

Code-Level Model Checking in the Software Development Workflow

Nathan Chong
Amazon

Byron Cook
Amazon
UCL

Konstantinos Kallas
University of Pennsylvania

Kareem Khazem
Amazon

Felipe R. Monteiro
Amazon

Daniel Schwartz-Narbonne
Amazon

Serdar Tasiran
Amazon

Michael Tautschnig
Amazon
Queen Mary University of London

Mark R. Tuttle
Amazon

ABSTRACT

This experience report describes a style of applying symbolic model checking developed over the course of four years at Amazon Web Services (AWS). Lessons learned are drawn from proving properties of numerous C-based systems, e.g., custom hypervisors, encryption code, boot loaders, and an IoT operating system. Using our methodology, we find that we can prove the correctness of industrial low-level C-based systems with reasonable effort and predictability. Furthermore, AWS developers are increasingly writing their own formal specifications. All proofs discussed in this paper are publicly available on GitHub.

CCS CONCEPTS

• Software and its engineering → Formal software verification; Model checking; Correctness; • Theory of computation → Program reasoning.

KEYWORDS

Continuous Integration, Model Checking, Memory Safety.

ACM Reference Format:

Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. 2020. Code-Level Model Checking in the Software Development Workflow. In *Software Engineering in Practice (ICSE-SEIP '20)*, May 23–29, 2020, Seoul, Republic of Korea. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3377813.3381347>

1 INTRODUCTION

This is a report on making code-level proof via model checking a routine part of the software development workflow in a large industrial organization. Formal verification of source code can have a significant positive impact on the quality of industrial code. In

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ICSE-SEIP '20, May 23–29, 2020, Seoul, Republic of Korea

© 2020 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-7123-0/20/05.

<https://doi.org/10.1145/3377813.3381347>

particular, formal specification of code provides precise, machine-checked documentation for developers and consumers of a code base. They improve code quality by ensuring that the program's *implementation* reflects the developer's *intent*. Unlike testing, which can only validate code against a set of concrete inputs, formal proof can assure that the code is both secure and correct for all possible inputs.

Unfortunately, rapid proof development is difficult in cases where proofs are written by a separate specialized team and *not* the software developers themselves. The developer writing a piece of code has an internal mental model of their code that explains why, and under what conditions, it is correct. However, this model typically remains known only to the developer. At best, it may be partially captured through informal code comments and design documents. As a result, the proof team must spend significant effort to reconstruct the formal specification of the code they are verifying. This slows the process of developing proofs.

Over the course of four years developing code-level proofs in Amazon Web Services (AWS), we have developed a proof methodology that allows us to produce proofs with reasonable and predictable effort. For example, using these techniques, one full-time verification engineer and two interns were able to specify and verify 171 entry points over 9 key modules in the AWS C Common¹ library over a period of 24 weeks (see Sec. 3.2 for a more detailed description of this library). All specifications, proofs, and related artifacts (such as continuous integration reports), described in this paper have been integrated into the main AWS C Common repository on GitHub, and are publicly available at <https://github.com/awslabs/aws-c-common/>.

1.1 Methodology

Our methodology has four key elements, all of which focus on communicating with the development team using artifacts that fit their existing development practices. We find that of the many different ways we have approached verification engagements, this combination of techniques has most deeply involved software developers in the proof creation and maintenance process. In particular, developers have begun to write formal functional specifications for code as they develop it. Initially, this involved the development team asking the verification team to assist them in writing specifications for new

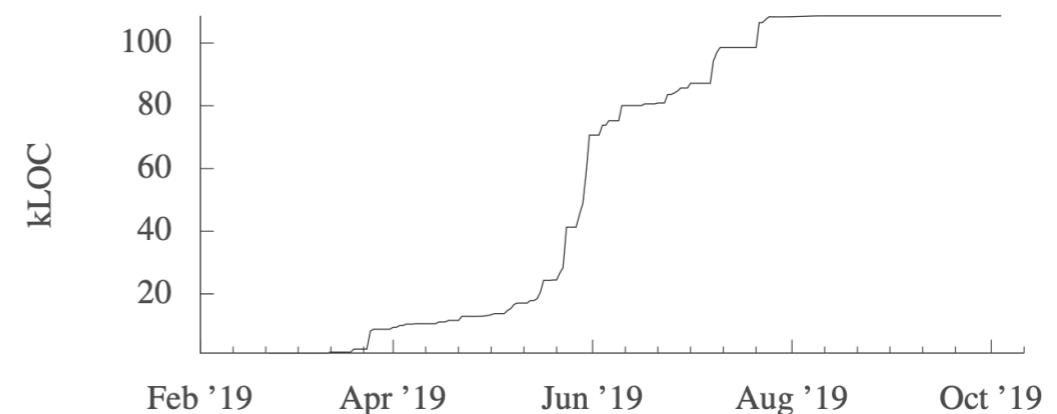


Figure 1: Cumulative number of LOC proven.

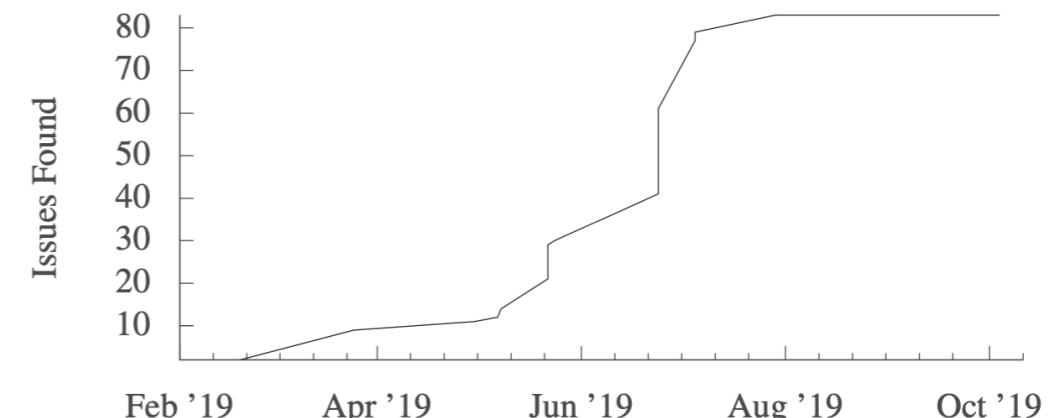


Figure 2: Cumulative number of issues found.

Table 1: Severity and root cause of issues found.

Root cause	# issues	Severity		
		High	Medium	Low
Integer overflow	10 (12%)	2	8	0
Null-pointer deref.	57 (69%)	0	14	43
Functional	11 (13%)	0	4	7
Memory safety	5 (6%)	0	5	0
Total	83	2	31	50
		(3%)	(37%)	(60%)

¹<https://github.com/awslabs/aws-c-common>

프로그램 분석 기술 요약

	Automatic	Sound	Complete	When
Testing/ Fuzzing				
Symbolic Execution				
Sound Static Analysis				
Bug-Finding Static Analysis				
Program Verification				
?				