Introduction to Software Security: Why It Matters

Bernard Nongpoh
Assistant Professor
Department of Computer Science and Engineering
IIT Guwahati

Research and Industrial Conclave 2025
Indian Institute of Technology Guwahati

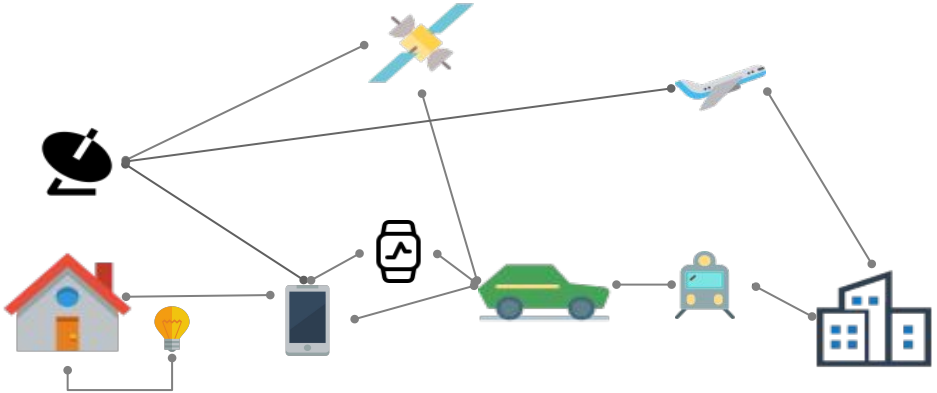
About Me

- Assistant Professor, Department of Computer Science and Engineering, IIT Guwahati
- Previously Staff Engineer at Qualcomm Product Security Initiative (QPSI), Hyderabad
- Research interests: Software and system security, program analysis, fuzzing, AI for Security and secure software design

Outline

- Context
- What is Software Security
- Background: Software Testing
- Automated Software Analysis & Verification
  - Static, Dynamic & Hybrid Analysis
- Dynamic Analysis: Fuzzing
- AI for Security
- Key Takeaways
- Bonus

Why Security Matters



School district breach

Dallas ISD reports compromise of personal data of students, staff

03 September 2021

Software is everywhere in Society

Attack on French government visa website exposes applicants' data

06 September 2021

Data breach at New York university potentially affects 47,000 citizens

16 August 2021

Singapore eye clinic potentially breaches 73,000 patients' data

27 August 2021

IIT Roorkee data leak: personal records of 30,0[illegible] students and alumni exposed for years

11 August 2025

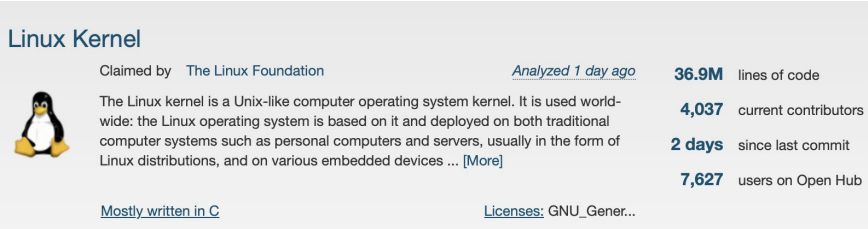
info@dpdpconsultants.com

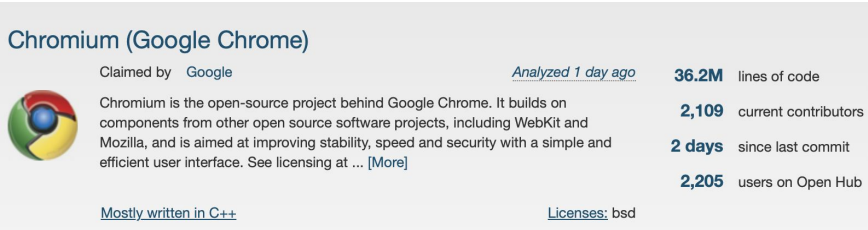
www.dpdpconsultants.com

https://portswigger.net/daily-swig/cyber-attacks
https://freeicons.io/

Modern System Software (1/10)

Extremely Large and Complex



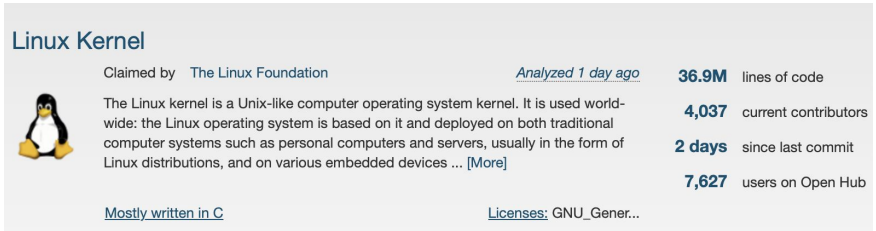


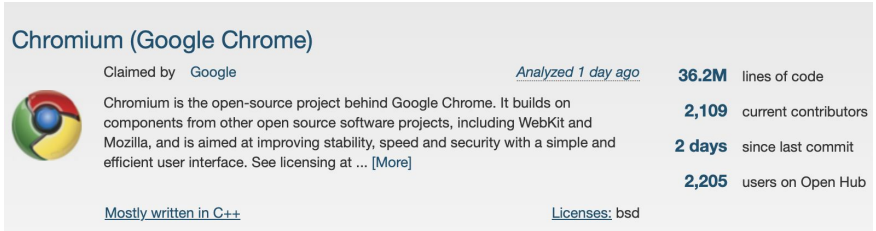
https://openhub.net/

Modern System Software (2/10)

Extremely Large and Complex

More Complex!

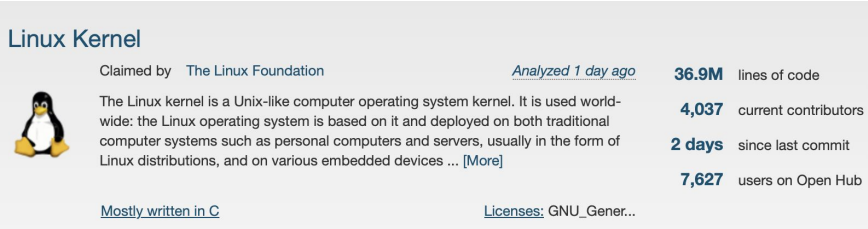




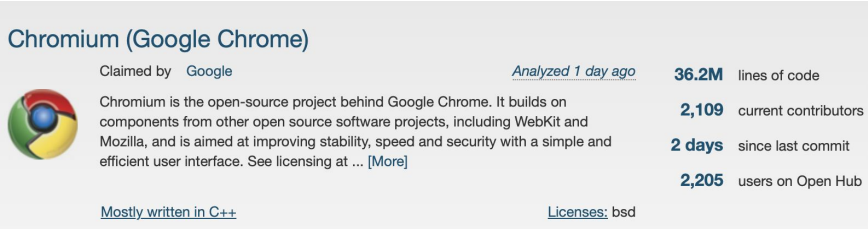
https://openhub.net/

Modern System Software (3/10)

Extremely Large and Complex



More Complex!



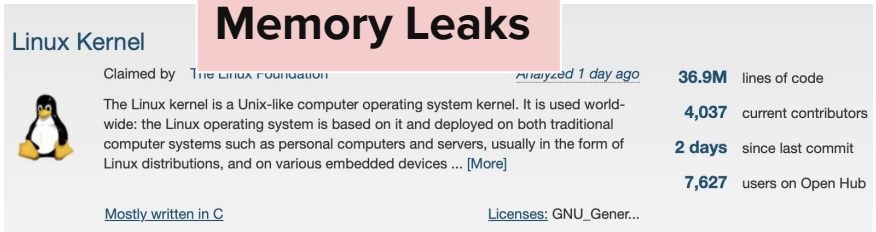
More Buggy!

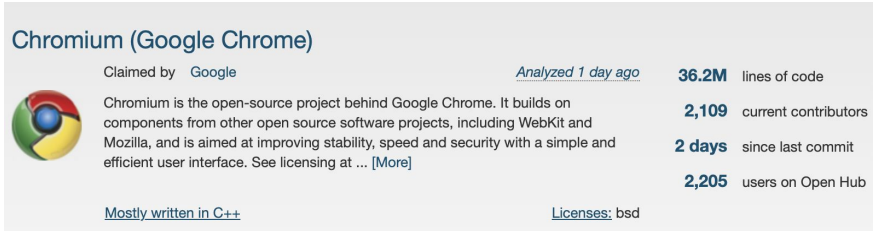
https://openhub.net/

Modern System Software (4/10)

Extremely Large and Complex

Memory Leaks





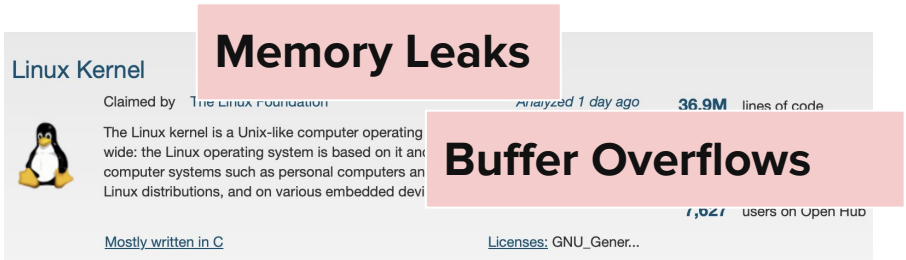
More Complex!

More Buggy!

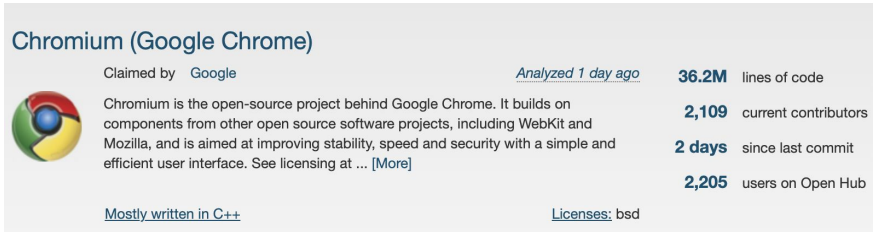
https://openhub.net/

Modern System Software (5/10)

Extremely Large and Complex



More Complex!

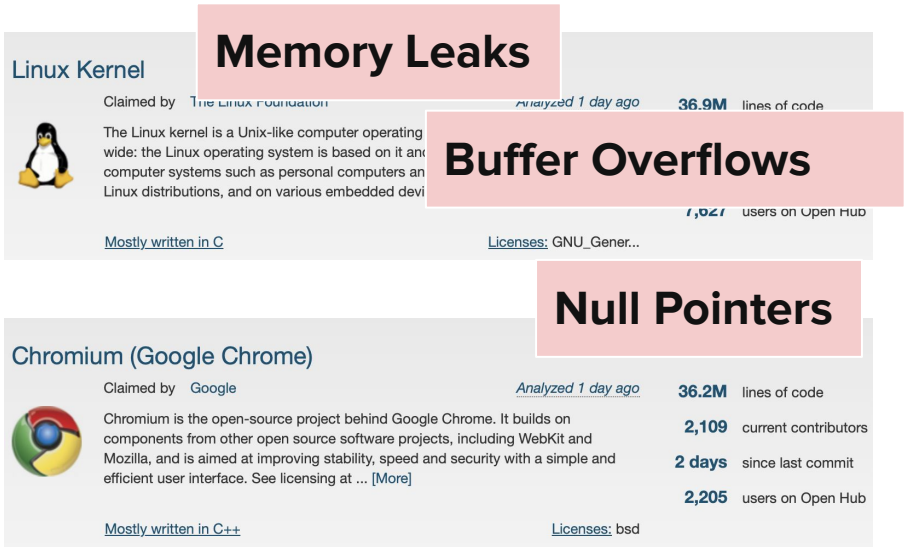


More Buggy!

https://openhub.net/

Modern System Software (6/10)

Extremely Large and Complex



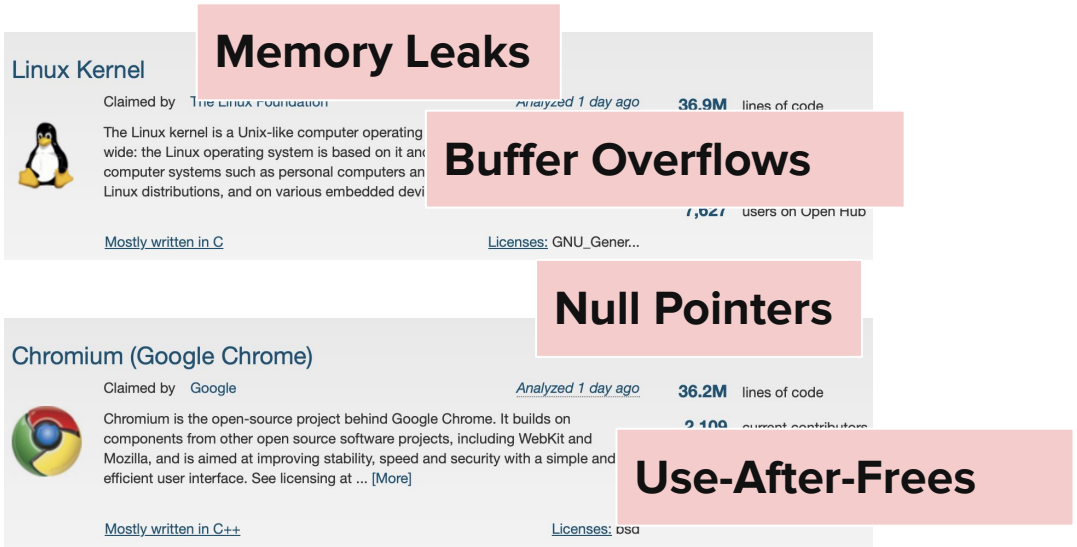
More Complex!

More Buggy!

https://openhub.net/

Modern System Software (7/10)

Extremely Large and Complex



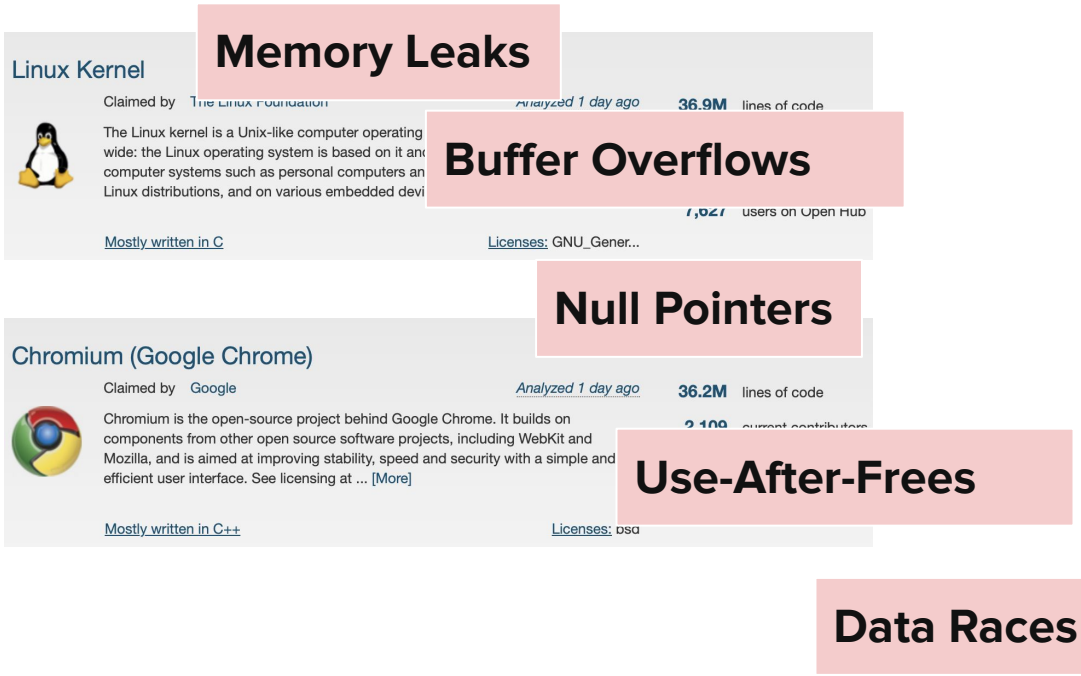
More Complex!

More Buggy!

https://openhub.net/

Modern System Software (8/10)

Extremely Large and Complex



More Complex!

More Buggy!

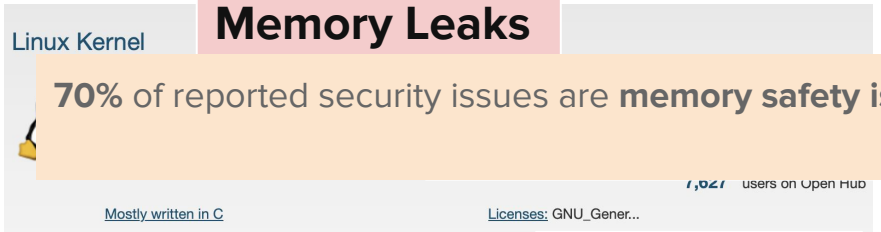
https://openhub.net/

Modern System Software (9/10)

Extremely Large and Complex

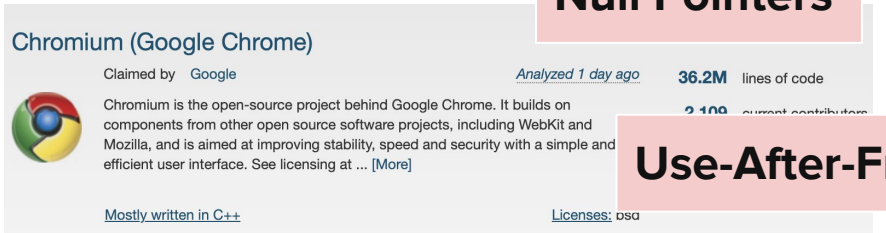
More Complex!

Memory Leaks



70% of reported security issues are **memory safety issues** according to **Microsoft** and **Google**.

Null Pointers



Use-After-Frees

More Buggy!

Data Races

https://www.cisa.gov/sites/default/files/2023-12/CSAC_TAC_Recommendations-Memory-Safety_Final_20231205_508.pdf

Modern System Software (10/10)

Extremely Large and Complex

More Complex!

Memory Leaks

70% of reported security issues are **memory safety issues** according to **Microsoft** and **Google**.

This memory safety issue persists despite widespread use of security techniques such as **fuzzing**, **secure software development lifecycle processes**, **static code analysis**, and **penetration testing**.

More Buggy!

Use-After-Frees

Data Races

https://www.cisa.gov/sites/default/files/2023-12/CSAC_TAC_Recommendations-Memory-Safety_Final_20231205_508.pdf

Software Becomes More Buggy

Extremely Larg

Memo

Linux Kernel

Claimed by The Linux Foundation

The Linux kernel is a Unix-like compute
wide: the Linux operating system is bas
computer systems such as personal co
Linux distributions, and on various emb

Mostly written in C

Chromium (Google Chrome)

Claimed by Google

Chromium is the open-source project b
components from other open source so
Mozilla, and is aimed at improving stab
efficient user interface. See licensing at

Mostly written in C++

More Complex!

More Buggy!

Data Races

https://www.linkedin.com/posts/mohassine_is-your-team-able-to-see-the-hidden-part-activity-7216868580041068546-SKTQ/

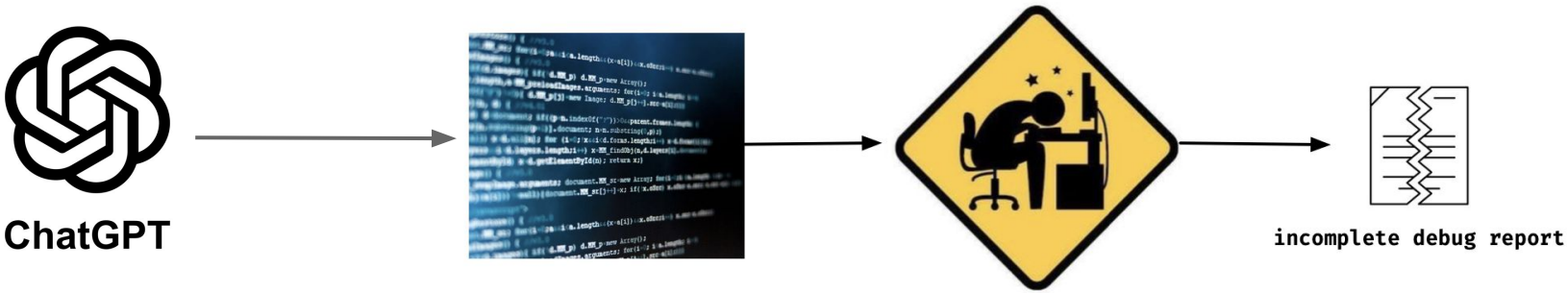
Code Review By Developers

Large Project with millions of lines of code is almost impossible to be manually reviewed by human:

- Intractable due to potentially unbounded number of paths that must be analyze
- Undecidable in the presence of dynamically allocate memory and recursive data structures

https://github.com/SVF-tools/Software-Security-Analysis/blob/slides/1.intro.pdf

Code Review By Developers



Large project with millions of lines of code is almost impossible to be manually reviewed by human:

- Too many execution paths to check exhaustively
- Dynamic memory and recursion create unpredictable behaviors
- Concurrency (multi-threading) introduces race conditions

https://github.com/SVF-tools/Software-Security-Analysis/blob/slides/1.intro.pdf

Code Review By Developers

Practically impossible for humans to reason about all paths manually — especially under time & resource constraints.

Large project with millions of lines of code is almost impossible to be manually reviewed by human:

- Too many execution paths to check exhaustively
- Dynamic memory and recursion create unpredictable behaviors
- Concurrency (multi-threading) introduces race conditions

https://github.com/SVF-tools/Software-Security-Analysis/blob/slides/1.intro.pdf

Incidents

- Heartbleed (2014) ➜ Data Leaks
- Log4Shell (2021) ➜ Remote Code Execution

One line of insecure code can compromise millions of systems

What Is Software Security?

- Ensuring correct and safe behavior under adversarial conditions
- A common entry point for attacks is **software vulnerabilities**
- Finding and fixing vulnerabilities before **attackers** find them

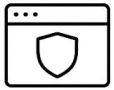
https://freeicons.io/

Threat Model

Attacker capabilities

- Local
- Remote
- Insider

Example:



Web App

Operating System Kernel

IoT Camera

Security depends on assumptions - define your **threat model** before defending

https://freeicons.io/

Buffer Overflow (A Vulnerable Example) (1/3)

```c
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
char buffer[4];
int access;

int main(int argc, char **argv)
{
 char *input = argv[1];
 char *secret = "pass";

 access = 0;
 strcpy(buffer,input);

 if (strcmp(buffer,secret)==0)
 {
 access = 1;
 }
 if(access)
 {
 printf("\nAccess Granted-->Root access...");
 }
 else
 {
 printf("\nAccess Denied!");
 }
return 0;
}
```

```
./buggy pass
Access Granted→Root access...
```

Buffer Overflow (A Vulnerable Example) (2/3)

```c
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
char buffer[4];
int access;

int main(int argc, char **argv)
{
 char *input = argv[1];
 char *secret = "pass";

 access = 0;
 strcpy(buffer,input);

 if (strcmp(buffer,secret)==0)
 {
 access = 1;
 }
 if(access)
 {
 printf("\nAccess Granted-->Root access...");
 }
 else
 {
 printf("\nAccess Denied!");
 }
return 0;
}
```

```
./buggy pass
Access Granted→Root access...
```

```
./buggy abcd
Access Denied!
```

Buffer Overflow (A Vulnerable Example) (3/3)

```c
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
char buffer[4];
int access;

int main(int argc, char **argv)
{
 char *input = argv[1];
 char *secret = "pass";

 access = 0;
 strcpy(buffer,input);

 if (strcmp(buffer,secret)==0)
 {
 access = 1;
 }
 if(access)
 {
 printf("\nAccess Granted-->Root access...");
 }
 else
 {
 printf("\nAccess Denied!");
 }
return 0;
}
```

```
./buggy pass
Access Granted→Root access...
```

```
./buggy abcd
Access Denied!
```

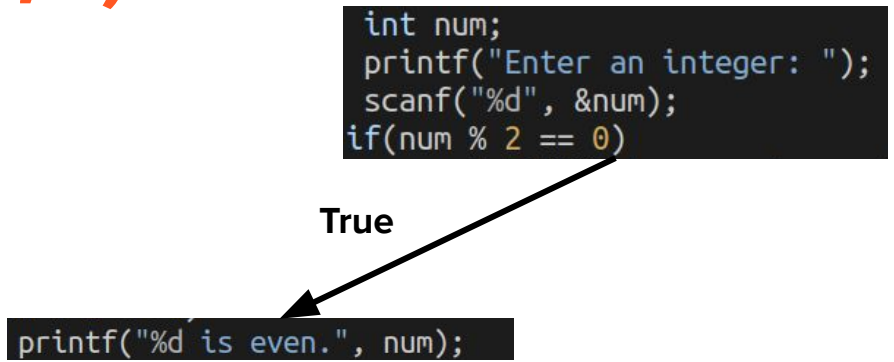
```
./buggy abcdefghijk
Access Granted→Root access...
```

Background: Control Flow Graph (1/6)

```c
#include <stdio.h>
int main() {
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);
    if(num % 2 == 0)
        printf("%d is even.", num);
    else
        printf("%d is odd.", num);

    return 0;
}
```

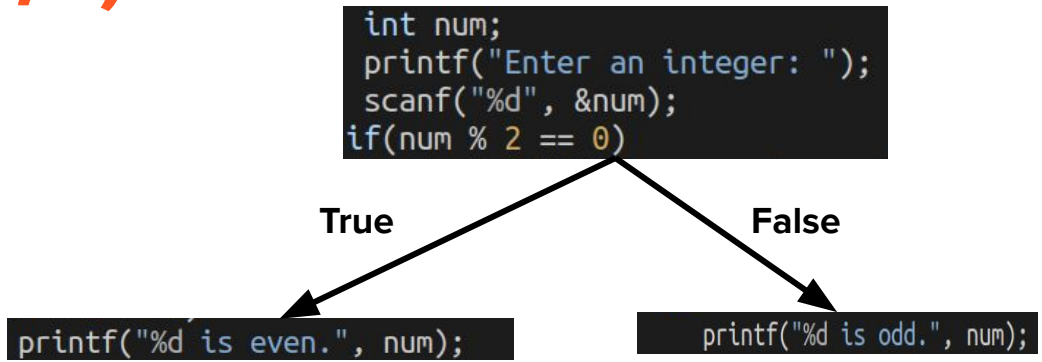
Control Flow Graph (2/6)

```c
#include <stdio.h>
int main() {
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);
    if(num % 2 == 0)
        printf("%d is even.", num);
    else
        printf("%d is odd.", num);

    return 0;
}
```



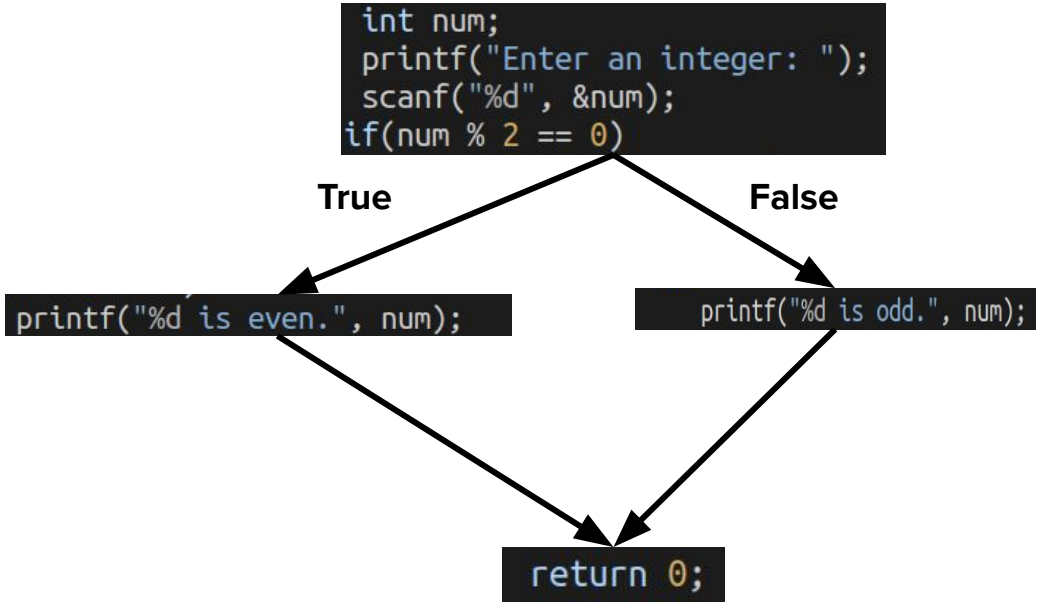
Control Flow Graph (3/6)

```c
#include <stdio.h>
int main() {
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);
    if(num % 2 == 0)
        printf("%d is even.", num);
    else
        printf("%d is odd.", num);

    return 0;
}
```



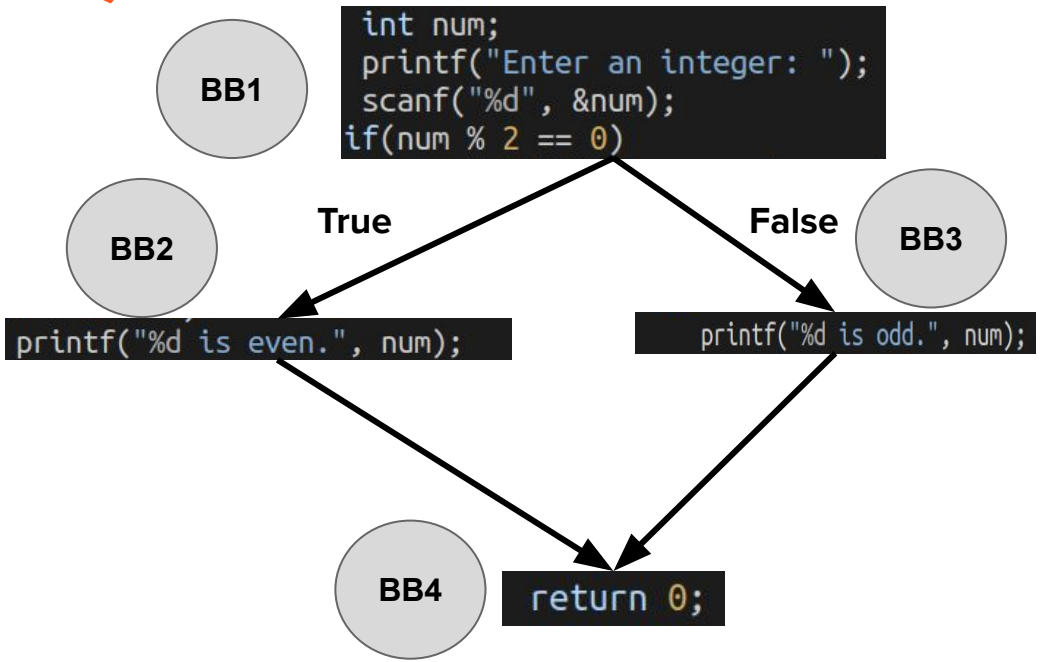
Control Flow Graph (4/6)

```c
#include <stdio.h>
int main() {
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);
    if(num % 2 == 0)
        printf("%d is even.", num);
    else
        printf("%d is odd.", num);

    return 0;
}
```



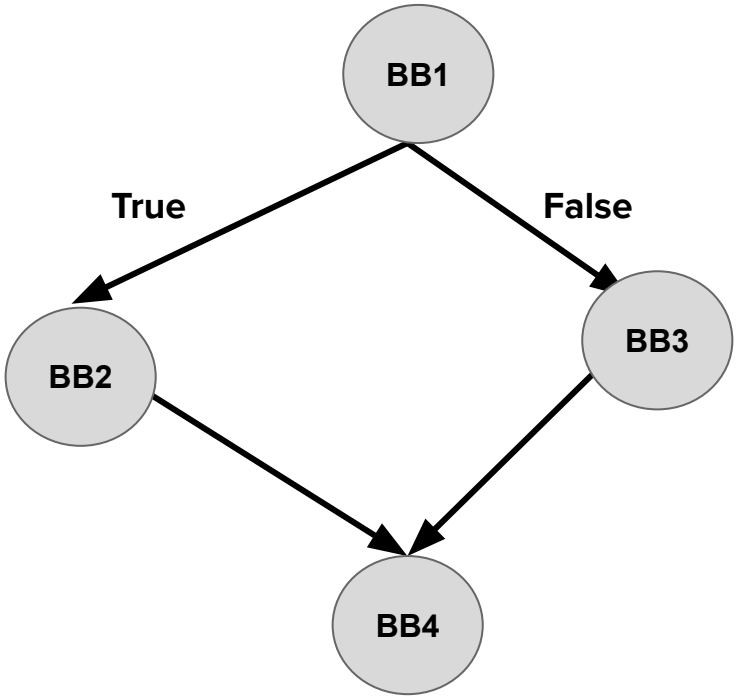
Control Flow Graph (5/6)

```c
#include <stdio.h>
int main() {
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);
    if(num % 2 == 0)
        printf("%d is even.", num);
    else
        printf("%d is odd.", num);

    return 0;
}
```



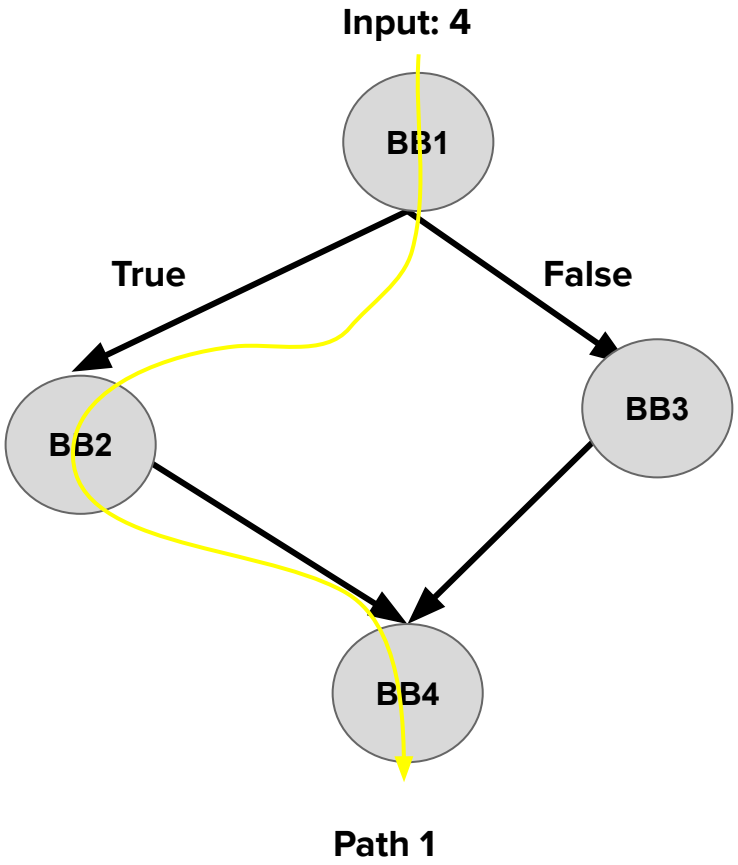
Control Flow Graph (6/6)

```c
#include <stdio.h>
int main() {
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);
    if(num % 2 == 0)
        printf("%d is even.", num);
    else
        printf("%d is odd.", num);

    return 0;
}
```



Execution Path (1/4)

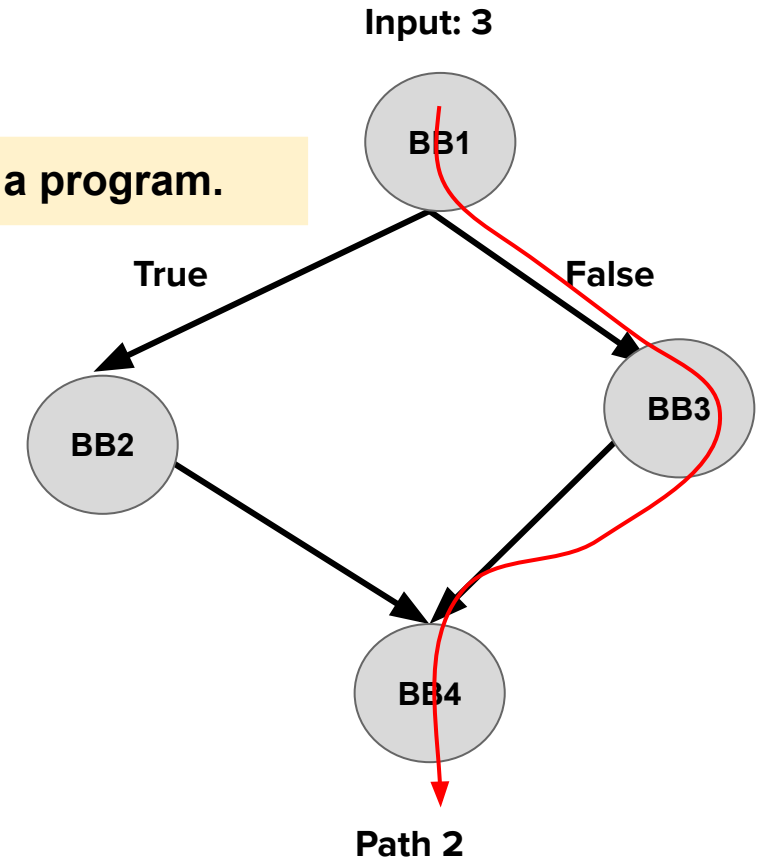
```c
#include <stdio.h>
int main() {
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);
    if(num % 2 == 0)
        printf("%d is even.", num);
    else
        printf("%d is odd.", num);

    return 0;
}
```



Execution Path (2/4)

An execution path is a possible flow of control of a program.

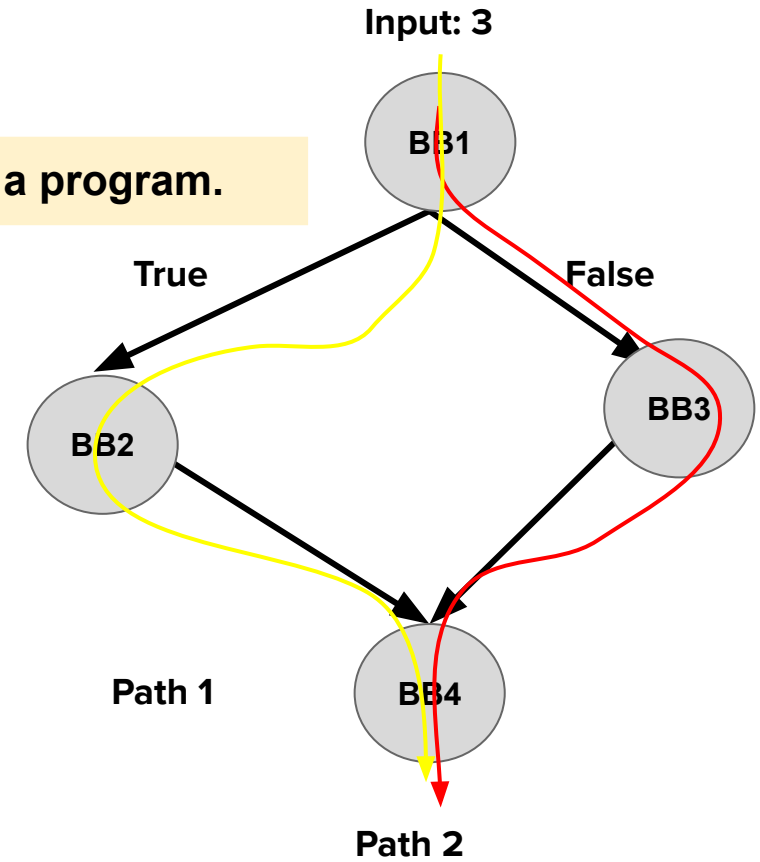
```c
#include <stdio.h>
int main() {
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);
    if(num % 2 == 0)
        printf("%d is even.", num);
    else
        printf("%d is odd.", num);

    return 0;
}
```



Execution Path (3/4)

An execution path is a possible flow of control of a program.

```c
#include <stdio.h>
int main() {
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);
    if(num % 2 == 0)
        printf("%d is even.", num);
    else
        printf("%d is odd.", num);

    return 0;
}
```



4 3

Test cases

100% Code Coverage

Automated Code Analysis & Verification

- **Software Analysis:** Identifies whether bugs exist in some execution paths

- **Software Verification:** Prove that no bugs exist in any execution path.
  - User specification should be satisfied and no bug should be triggered

Why Software Analysis and Verification?

- Better quality in terms of more secure and reliable software
- Reduce time to Market
  - Less time for debugging
  - Less time for later phase testing and bug fixing
- Consistent with user expectations/specifications

Automated Code Analysis: A Necessity

Dynamic Analysis

- Run the code and observes the behavior at run time
- Example
  - Unit testing
  - Fuzz testing

Automated Code Analysis: A Necessity

Dynamic Analysis

- Run the code and observes the behavior at run time
- Example
  - Unit testing
  - Fuzz testing

Static Analysis

- Inspect the code without running it (compile-time)
- Example
  - Symbolic execution
  - Abstract interpretation

Automated Code Analysis: A Necessity

Dynamic Analysis

- Run the code and observes the behavior at run time
- Example
  - Unit testing
  - Fuzz testing

Static Analysis

- Inspect the code without running it (compile-time)
- Example
  - Symbolic execution
  - Abstract interpretation

Hybrid Analysis

Dynamic Analysis (During Execution)

- **Fuzzing:** Feeding random or malformed inputs to the program
- **Unit Testing:** Testing individual components with known inputs and expected outputs
- **Stress Testing:** Pushing the system to its limits (e.g., memory exhaustion)
- **Model-Based Testing:** Tests derived from system models or specifications

Dynamic Analysis (During Execution)

- **Fuzzing:** Feeding random or malformed inputs to the program
- **Unit Testing:** Testing individual components with known inputs and expected outputs
- **Stress Testing:** Pushing the system to its limits (e.g., memory exhaustion)
- **Model-Based Testing:** Tests derived from system models or specifications

Limitations:

- May not explore all possible paths

Dynamic Analysis (During Execution)

```
int buffer[10];
buffer[10] = 5; // out-of-bounds write – may not crash in testing
```

A test case may or may not catch this depending on input

We need **TEST ORACLE**: Address Sanitizer

https://github.com/google/sanitizers/wiki/addresssanitizer

Dynamic Analysis (During Execution)

- **Purity**: Array Bound Checking
- **Valgrind**: Memory Leak Detection
- **Eraser**: Data Race Detection
- **Daikon**: Finding likely invariants

Static Analysis (Without Execution)

- Analyze all possible code paths
- Works on Control Flow Graphs (CFGs) and Data Flow Graphs
- Helps catch bugs even if no test input triggers them

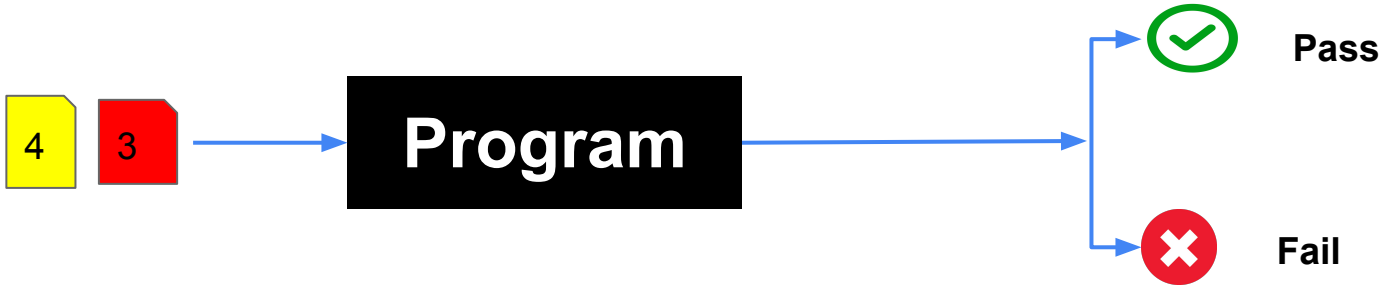
Limitations:

- False positives
- Difficult to reproduce the bug

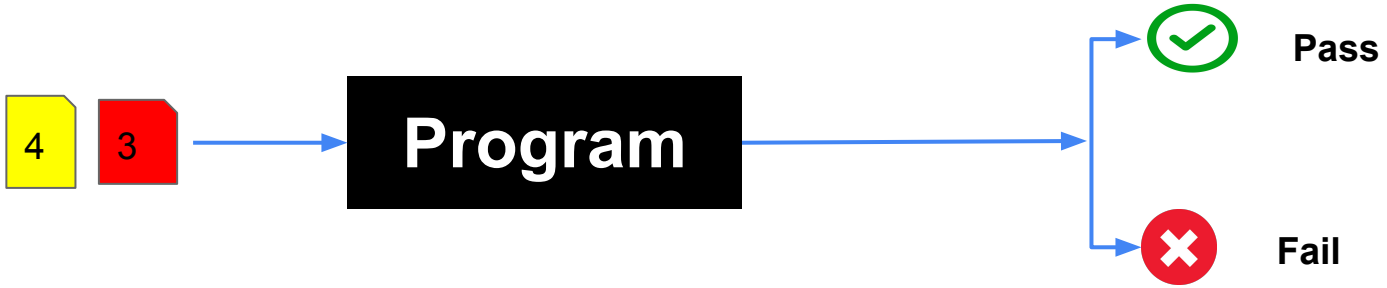
Static Analysis (Without Execution)

- **Lints, FindBugs, Coverity**: Suspicious error patterns
- **Microsoft SLAM**: Checking API usage
- **Facebook Infer**: Memory Leak Detection
- **SVF**: Static Value Flow Analysis, a general framework for static analysis for C/C++

Software Testing (1/3)

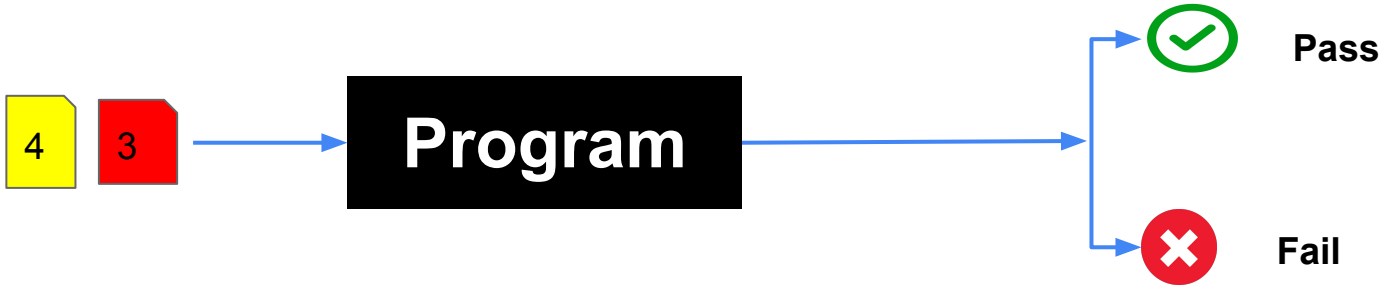


Software Testing (2/3)



Test oracle

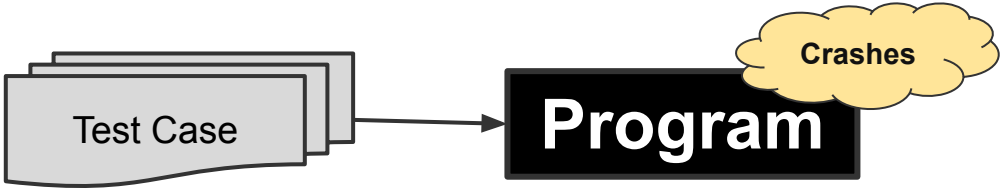
Software Testing (3/3)



Test case = test input + test oracle

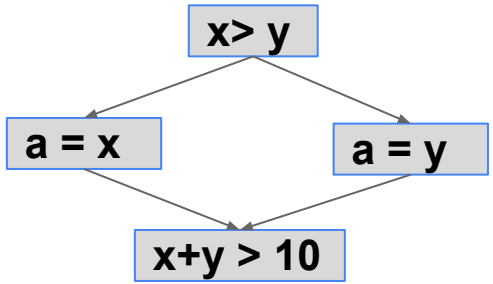
Automated Software Testing

Security Vulnerabilities

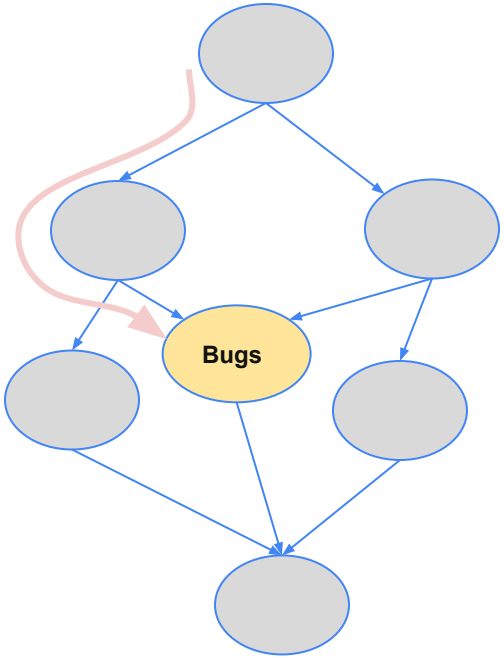


Symbolic Execution

KLEE



Path-1 = (x>y) ∧ (x+y> 10)
Path-2 = ¬ (x>y) ∧ (x+y> 10)

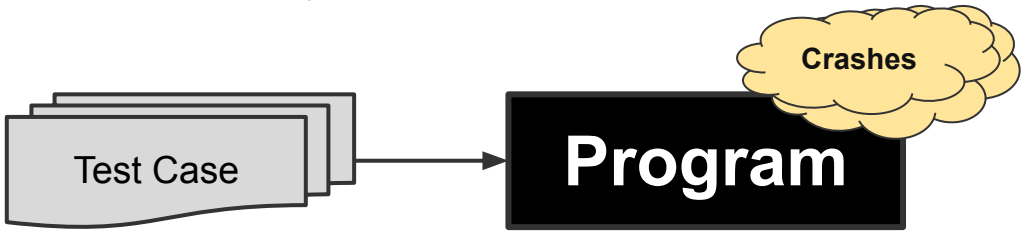


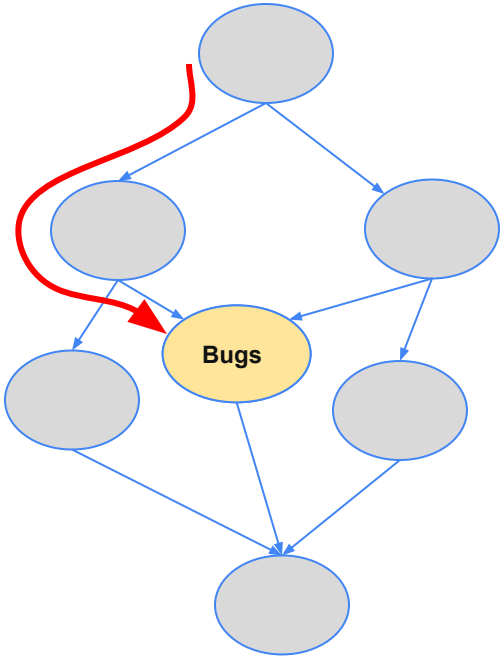
Stack Overflow
Heap Overflow
Use After Free
.....
.....

Automated Software Testing

Software Bugs

Security Vulnerabilities





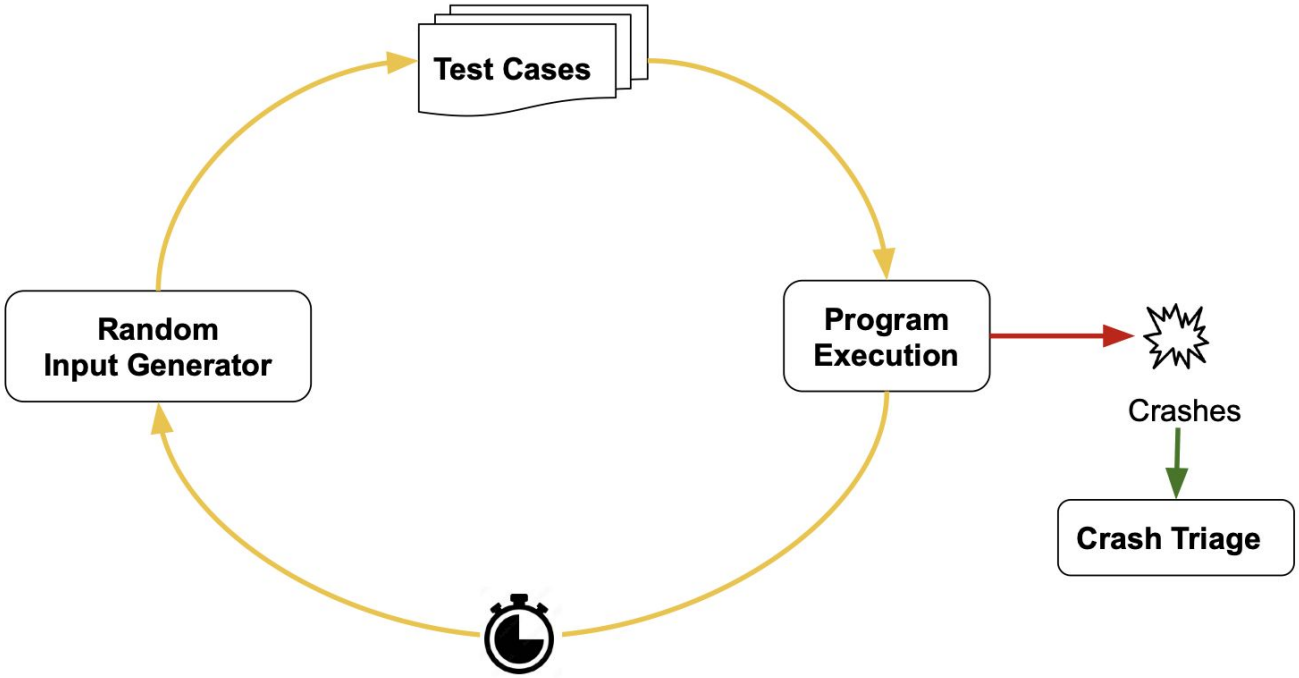
Symbolic Execution

KLEE

Fuzzing

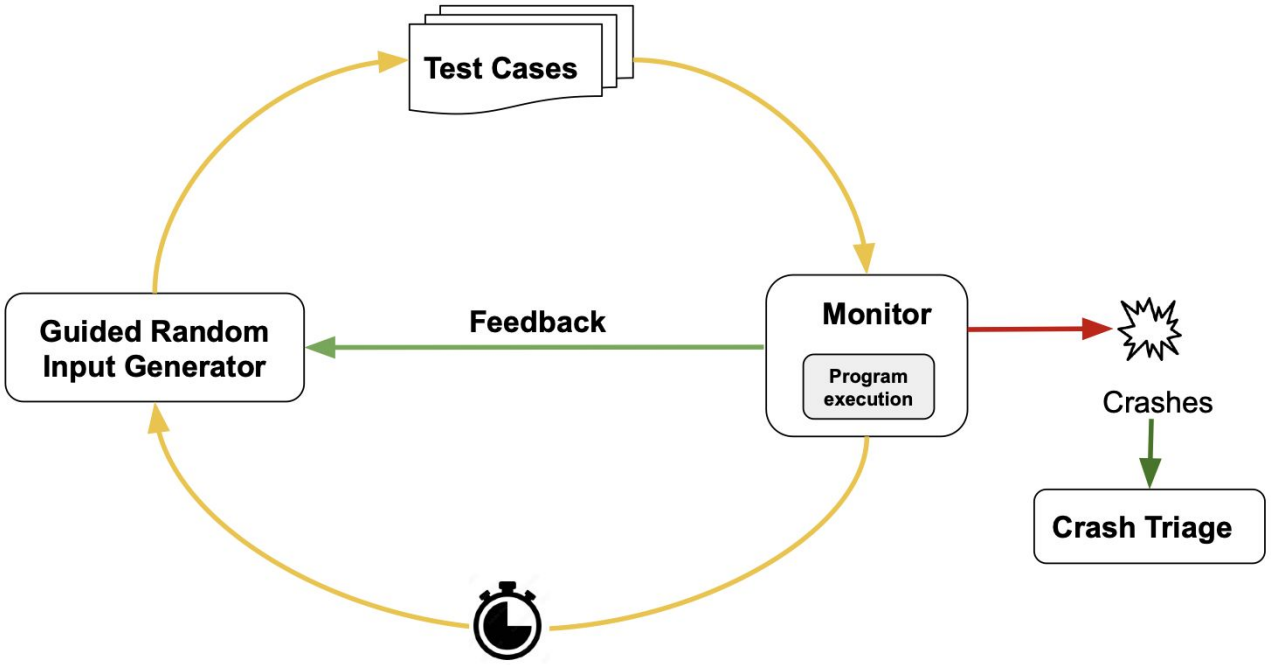
Stack Overflow
Heap Overflow
Use After Free
.....
.....

Fuzzing



Working process of fuzzing test

Fuzzing

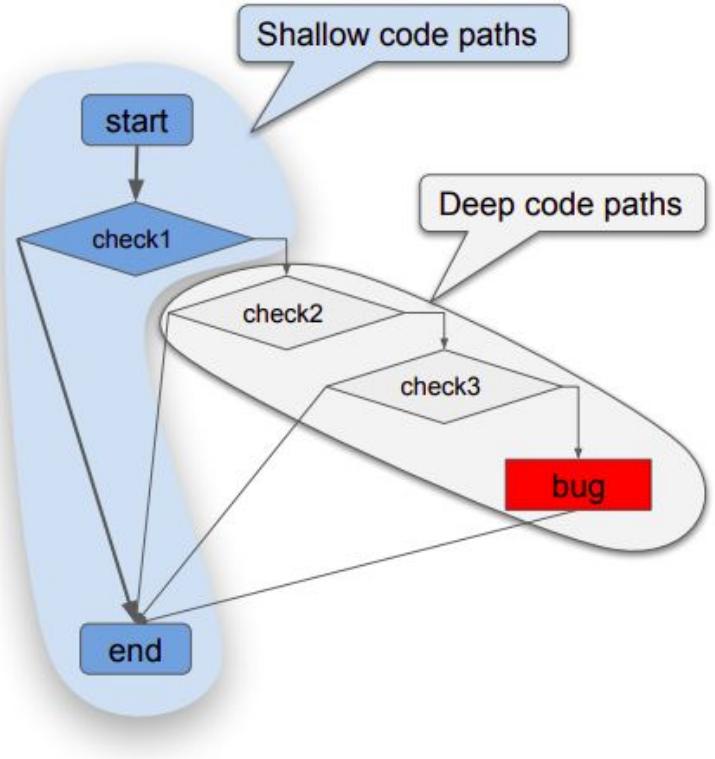


Working process of fuzzing test

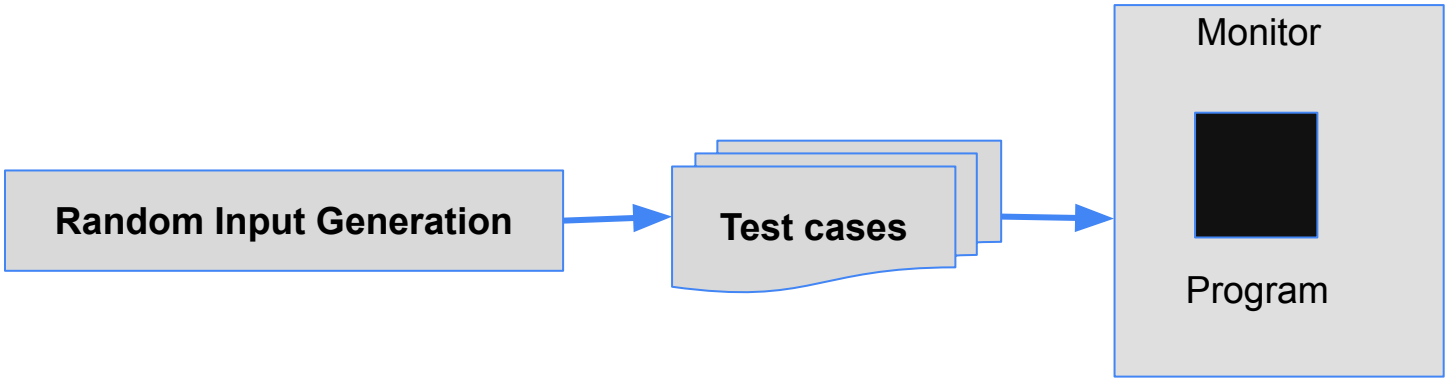
Challenges for Fuzzers

- **Challenges**
  - Shallow coverage
  - Hard to find “deep” bugs

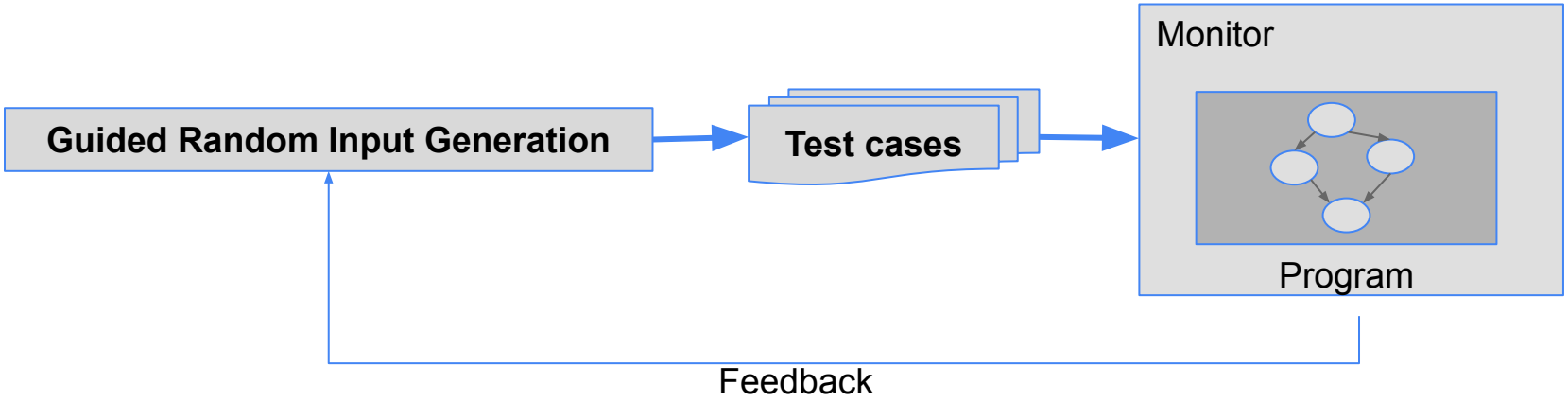
- **Root cause**
  - Fuzzer-generated inputs cannot bypass complex sanity checks in the target program



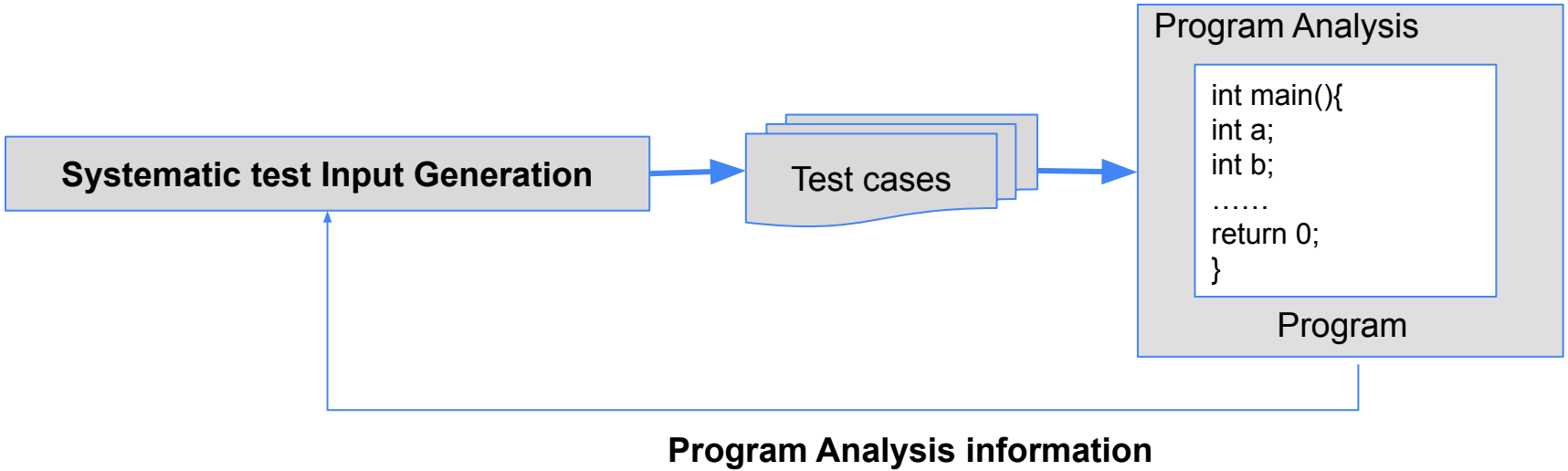
Blackbox Fuzzing



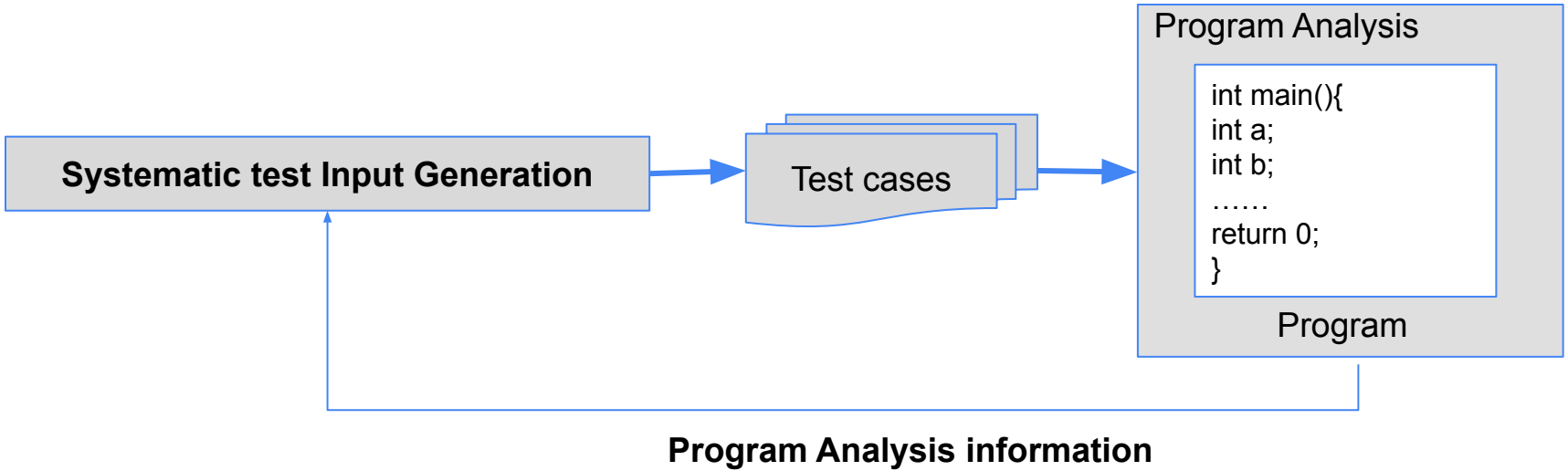
Greybox Fuzzing



Whitebox Fuzzing



Whitebox Fuzzing



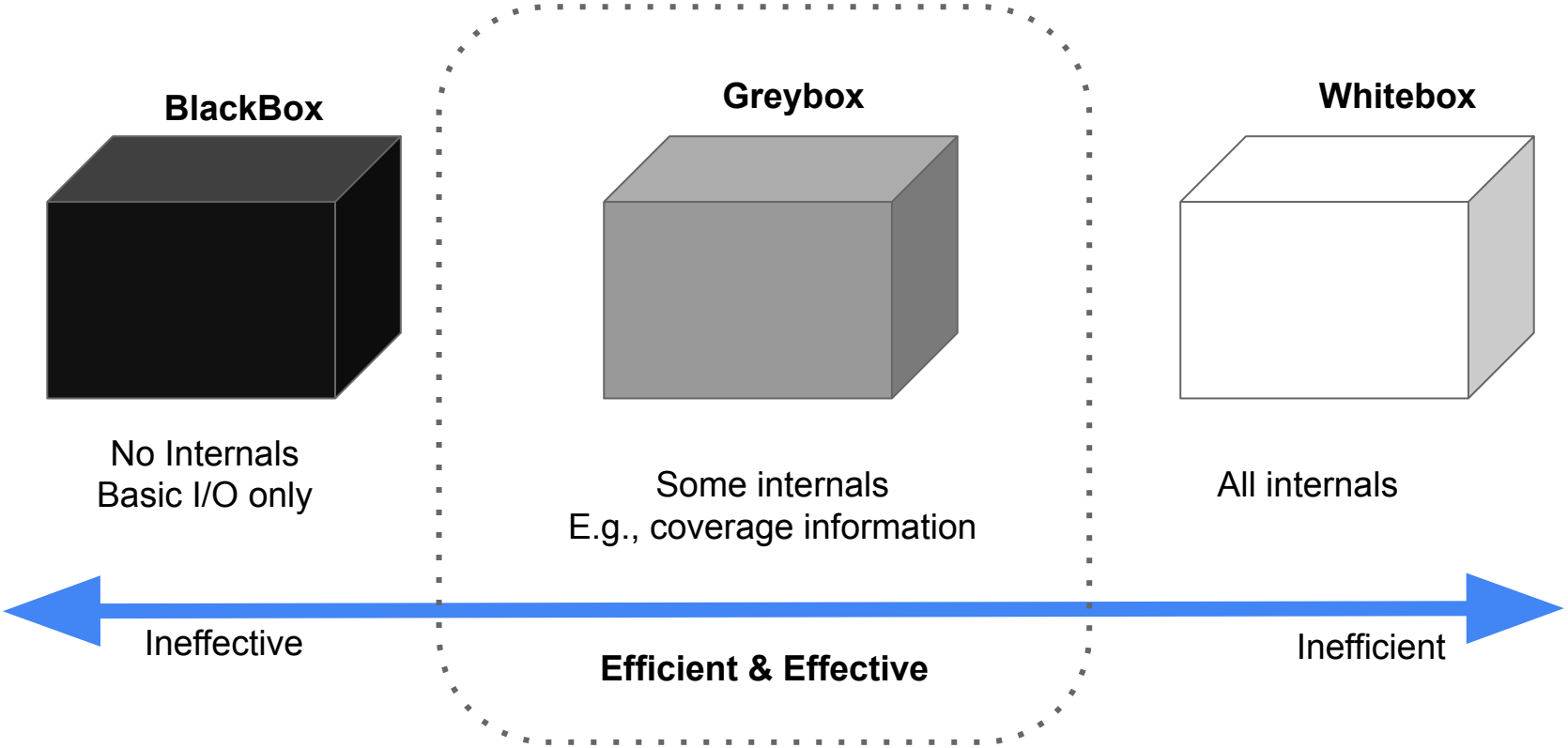
Fine-grained Coverage-based Fuzzing

Authors: Wei-Cheng Wu, Bernard Nongpoh, Marwan Nour, Michaël Marcozzi, Sébastien Bardin, Christophe Hauser | Authors Info & Claims

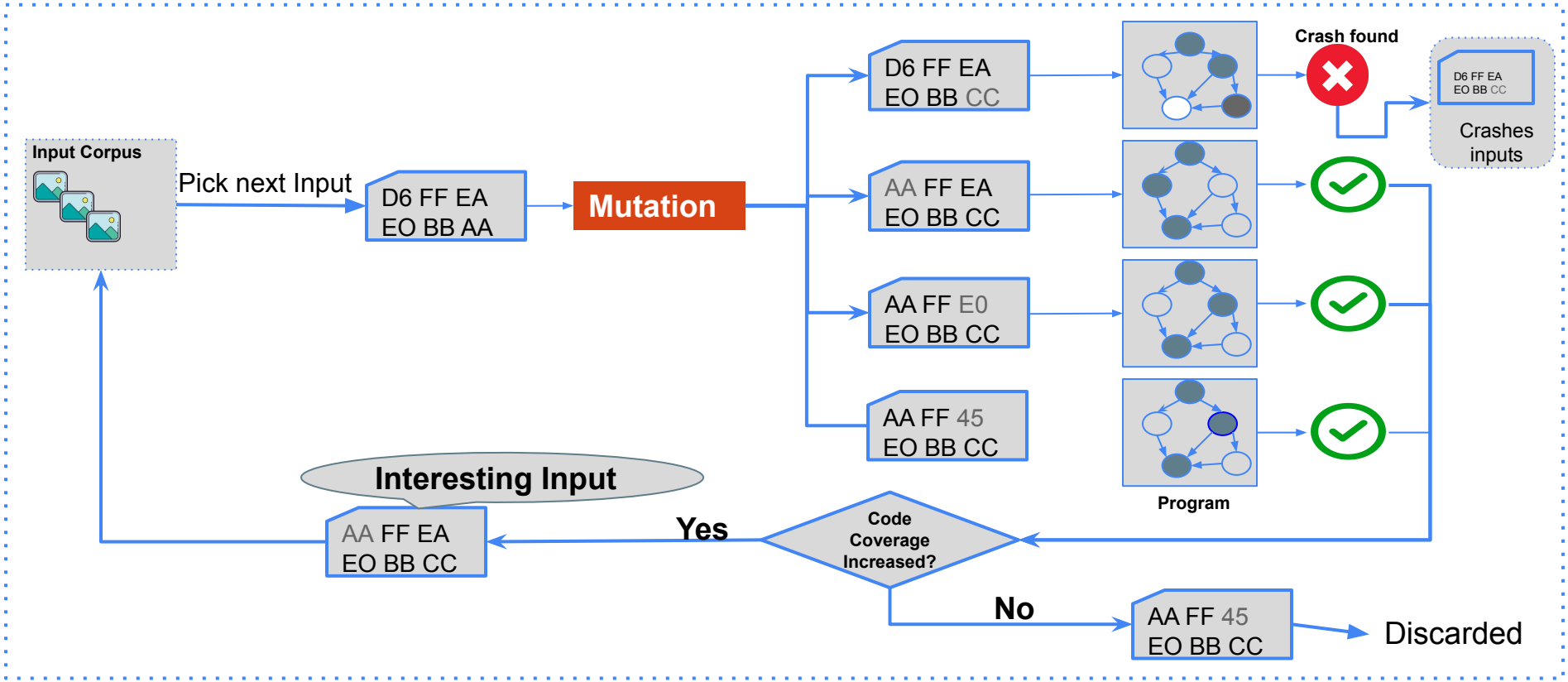
ACM Transactions on Software Engineering and Methodology, Volume 33, Issue 5 • Article No.: 138, Pages 1 - 41
https://doi.org/10.1145/3587158

https://dl.acm.org/doi/10.1145/3587158

Type of Fuzzers



Greybox Fuzzing



State of the art Fuzzers

American Fuzzy Lop plus plus (AFL++)

https://github.com/AFLplusplus/AFLplusplus

OSS-Fuzz - continuous fuzzing for open source software.

https://google.github.io/oss-fuzz

Pivot: AI/LLMs in Security

- AI-guided fuzzing (learning valid inputs)
- LLM-assisted static analysis
- AI-based patch suggestion / code repair
- Vulnerability triage from reports/logs

Pivot: AI/LLMs in Security

- AI-guided fuzzing (learning valid inputs)
- LLM-assisted static analysis
- AI-based patch suggestion / code repair
- Vulnerability triage from reports/logs

Limitations
Hallucination & lack of guarantees

AI for Security: Promise and Pitfalls

- Learn from large codebases, but lack formal soundness
- Improve developer productivity, may introduce insecure code
- Great triage assistant, needs human oversight

Darpa AIxCC: AI Cyber Challenge

- Launched in 2023, the Artificial Intelligence Cyber Challenge (AIxCC) is a two-year competition that brings together the best and brightest in AI and cybersecurity to safeguard the software critical to all Americans
- $4 million price

https://www.darpa.mil/research/programs/ai-cyber

Darpa AIxCC: AI Cyber Challenge

Rank	Team	Total Score	Accuracy
1	Team Atlanta	392.76	91.27%
2	Trail of Bits	219.35	89.33%
3	Theori	210.68	44.44%
4	All You Need IS A Fuzzing Brain	153.7	53.77%
5	Shellphish	135.89	94.83%
6	42-b3yond-6ug	105.03	89.23%
7	Lacrosse	9.59	42.86%

https://www.darpa.mil/research/programs/ai-cyber

Darpa AIxCC: AI Cyber Challenge

Team Atlanta

ATLANTIS: *AI-driven Threat Localization, Analysis, and Triage Intelligence System*

Taesoo Kim

Fuzzing + **Traditional Program Analysis** + **LLMs**

https://taesoo.kim/

https://arxiv.org/pdf/2509.14589

Key Takeaways

- To build resilient systems, we must combine:
  - **Static & symbolic reasoning** → propose candidate flaws and proofs

Key Takeaways

- To build resilient systems, we must combine:
  - **Static & symbolic reasoning** ➜ propose candidate flaws and proofs
  - **Dynamic testing / fuzzing** ➜ validate behavior, catch real bugs

Key Takeaways

- To build resilient systems, we must combine:
  - **Static & symbolic reasoning** → propose candidate flaws and proofs
  - **Dynamic testing / fuzzing** → validate behavior, catch real bugs
  - **AI / LLM-assisted reasoning** → scale insight, guide prioritization and patch generation

Key Takeaways

- To build resilient systems, we must combine:
  - **Static & symbolic reasoning** → propose candidate flaws and proofs
  - **Dynamic testing / fuzzing** → validate behavior, catch real bugs
  - **AI / LLM-assisted reasoning** → scale insight, guide prioritization and patch generation
  - **Confidence & correctness metrics** → patches must be trustworthy, not just plausible

Key Takeaways

- To build resilient systems, we must combine:
  - **Static & symbolic reasoning** → propose candidate flaws and proofs
  - **Dynamic testing / fuzzing** → validate behavior, catch real bugs
  - **AI / LLM-assisted reasoning** → scale insight, guide prioritization and patch generation
  - **Confidence & correctness metrics** → patches must be trustworthy, not just plausible
  - **Scalability & cost awareness** → techniques must handle millions of lines and act quickly

Key Takeaways

- To build resilient systems, we must combine:
  - **Static & symbolic reasoning** → propose candidate flaws and proofs
  - **Dynamic testing / fuzzing** → validate behavior, catch real bugs
  - **AI / LLM-assisted reasoning** → scale insight, guide prioritization and patch generation
  - **Confidence & correctness metrics** → patches must be trustworthy, not just plausible
  - **Scalability & cost awareness** → techniques must handle millions of lines and act quickly
  - **Hybrid Approach** → no single method suffices: combine, fallback, cross-validate

Thank you for your attention!

Getting into Computer Systems Security

./ pwn.college

Learn to hack!

Welcome to pwn.college!

pwn.college is an education platform for students (and other interested parties) to learn about, and practice, core cybersecurity concepts in a hands-on fashion. In martial arts terms, it is designed to take a “white belt” in cybersecurity to becoming a “blue belt”, able to approach (simple) CTFs and wargames. The philosophy of pwn.college is “practice makes perfect”.

pwn.college was created by Zardus (Yan Shoshitaishvili) and kanak (Connor Nelson) at Arizona State University. It powers ASU’s Computer Systems Security course, CSE466, and is now **open, for free, to participation for interested people around the world**!

Yan Shoshitaishvili

>> Module 0: Introduction
>> Module 1: Program Interaction
>> Module 2: Program Misuse
>> Module 3: Assembly Refresher
>> Module 4: Shellcoding
>> Module 5: Sandboxing
>> Module 6: Debugging Refresher
>> Module 7: Binary Reverse Engineering
>> Module 8: Memory Errors
>> Module 9: Exploitation
>> Module A: Return Oriented Programming
>> Module B: Dynamic Allocator Misuse
>> Module C: Race Conditions
>> Module D: Kernel Security
>> Module E: Advanced Exploitation

LiveOverflow

679K subscribers

Bonus

Bug Bounty Program