UNIVERSITÄT DUISBURG ESSEN

Offen im Denken

Vorlesung Application Management, Wintersemester 2025/26

Software Testing

Prof. Dr.-Ing. Lucas Vincenzo Davi
Fakultät für Informatik
Arbeitsgruppe Systemsicherheit

PALUNO
The Ruhr Institute for Software Technology

Motivation

- Auch wenn man die besten und achtsamsten Entwickler hat, Fehler in der SW-Entwicklung sind nicht vermeidbar
- Stichwort: „Edge Cases“
- Unerwarteter Input führt im Worst-Case zu:
  - Datendiebstahl (Confidentiality Violation)
  - Datenmanipulation (Integrity Violation)
  - Einschränkung der Verfügbarkeit des Dienstes (Availability Violation)
- Programmierfehler können häufig Security Probleme verursachen:
  - Buffer Overflow
  - Cross-Site Scripting

Software Testing erhöht die Sicherheit (Security) und Zuverlässigkeit (Reliability)

Konzept von Unit Testing

- Reminder:
  - Software in kleine, abgekapselte Komponenten unterteilen und isoliert testen
  - Keine externen Abhängigkeiten
- Eine „Unit“ wird mit verschiedenen Inputs getestet
- Viele Testing Frameworks nutzen die xUnit (https://en.wikipedia.org/wiki/XUnit) Architektur
- Schnelle und zuverlässige Antwort, ob eine Code Änderung zu unerwarteten Resultaten führt

Unit Test Beispiel mit GoogleTest


```
int Factorial(int n);  // Returns the factorial of n
```

A test suite for this function might look like:

```
// Tests factorial of 0.
TEST(FactorialTest, HandlesZeroInput) {
  EXPECT_EQ(Factorial(0), 1);
}

// Tests factorial of positive numbers.
TEST(FactorialTest, HandlesPositiveInput) {
  EXPECT_EQ(Factorial(1), 1);
  EXPECT_EQ(Factorial(2), 2);
  EXPECT_EQ(Factorial(3), 6);
  EXPECT_EQ(Factorial(8), 40320);
}
```

Quelle: https://google.github.io/googletest/primer.html

Das Entwickeln von Unit Tests

Klassische Strategie

- Häufig werden Unit Tests direkt nach Schreiben des Codes entwickelt
- Unit Tests und neuer Code werden in einem Commit zusammen hochgeladen
- Wenn Code Reviews genutzt werden, sollten der Review Prozess auch die Unit Tests begutachten

Neuerer Ansatz: Test-Driven Development (TDD)

- Schreiben der Tests bevor der neue Code entwickelt ist
- Die Tests schlagen so lange fehl bis ein Feature vollständig implementiert ist

Für komplexe Systeme kommt es häufig vor, dass man mehr Zeit zum Schreiben der Tests als zum Schreiben des Codes benötigt

Auswirkungen des Unit Testing auf den Code

- Damit Code getestet werden kann, bedarf es häufig Refactoring bzw. Schreiben von neuem Code, den man gut testen kann
- Möglichkeit zum Abfangen von Aufrufen zu externen Systemen
- Beispiel: Verhindern von „Flaky“ Tests in einem Issue Tracker
  - Unit Tests sollten kein echtes Ticket erstellen können (Noise)
  - Tests könnten fehlschlagen nur weil das Ticketing System nicht verfügbar ist (externe Abhängigkeit)
  - Besser (Mocks, Stubs, Fakes): Abfangen der Aufrufe zum Ticketing System und ein Interface zu einem IssueTrackerService Objekt implementieren
  - Die Tests würden dann nur prüfen, welche Daten an das Interface gehen, aber keinen Aufruf im Ticketing System ausführen

Konzept von Acceptance Tests

Reminder:

- Acceptance Tests sind nicht auf einzelne Einheiten (Units) und Fake Aufrufen zu externen Datenbanken und Netzwerkdiensten beschränkt
- Diese Tests analysieren komplette Code Pfade
- Häufig auch als Integration Test bezeichnet

Unterschiede zum Unit Testing

- Tests haben mehr „Flakiness“ als Unit Tests
- Sie benötigen deutlich mehr Zeit
- Sie führen zu deutlich mehr Vertrauen in die Software
- Große Firmen haben extra Teams, die sich nur um die Infrastruktur für Integration Testing kümmern
- Wenn ein Acceptance Test fehlschlägt, aber alle Unit Tests erfolgreich waren, bedarf es häufig viel Aufwand, um den Fehler zu finden
- Beachte:
  - Acceptance Tests ersetzen nicht Unit Tests; beide Testarten sind wichtig

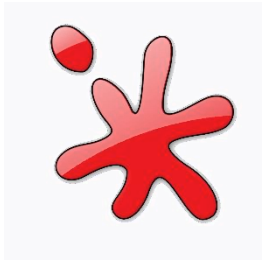
Programmanalyse für Acceptance Tests (*folgende Folien sind in Englisch verfasst*)

Program Analysis


```
004034F0  0C 00 00 FF D0 BE B0 82  40 00 56 E8 52 32 00 00  ...ÿÐ¾°‚@.VèR2..
00403500  56 FF 15 50 81 40 00 8D  74 06 01 80 3E 00 75 EA  Vÿ.P.@..t..€>.uê
00403510  6A 0A E8 AB 32 00 00 6A  08 E8 A4 32 00 00 6A 06  j.è«2..j.è¤2..j.
00403520  A3 44 A2 42 00 E8 98 32  00 00 3B C3 74 0F 6A 1E  £D¢B.è˜2..;Ãt.j.
00403530  FF D0 85 C0 74 07 80 0D  4F A2 42 00 40 55 FF 15  ÿÐ…Àt.€.O¢B.@Uÿ.
00403540  44 80 40 00 53 FF 15 A0  82 40 00 A3 18 A3 42 00  D€@.Sÿ. ‚@.£.£B.
00403550  53 8D 44 24 34 68 B4 02  00 00 50 53 68 E8 16 42  S.D$4h´...PShè.B
```

Dynamic Analysis

Static Analysis



Source Code or Binary Code?

- Both is possible
- Both could collect performance profiling information, code coverage metrics, security and reliability issues
- Dynamic Binary Instrumentation
  - Utilize a kind of process virtual machine to inspect the behavior of a binary file
  - Examples include Valgrind, Intel Pin, QEMU
- Dynamic analysis based on compiler support
  - Often required for detecting memory-related errors (e.g., buffer overflows)
  - The compiler inserts additional instructions and callbacks

Sanitizers

Address Sanitizer (ASan)

Originally developed as part of the LLVM compiler infrastructure:
https://clang.llvm.org/docs/AddressSanitizer.html

Finds memory-related bugs

- Out-of-Bounds memory access (heap & stack)
- Use-after-free bugs

Builds an instrumented binary during compilation

- The instrumentation includes callbacks to the sanitizer runtime
- The runtime maintains metadata about program execution and memory accesses
- This allows detection of invalid memory accesses

ASan Instrumentation and Shadow Memory

Utilizes shadow memory to store metadata about memory

- 8 byte memory require 1 byte shadow memory

Code instrumentation on every memory access (read/write)

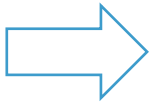
- Retrieve shadow memory and check if memory is “poisoned”
- Abort if accessed memory is poisoned

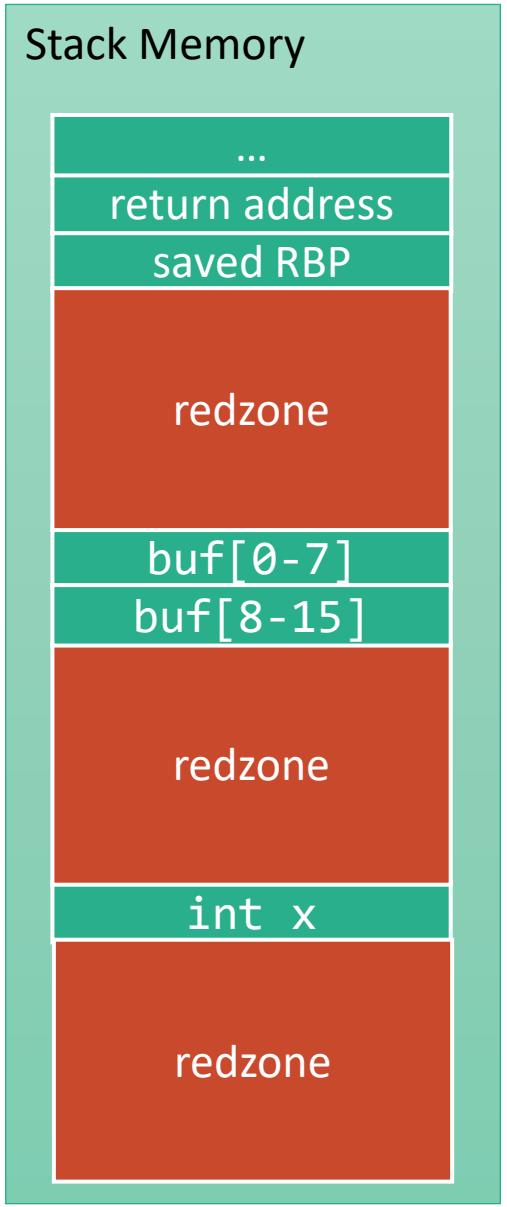
Instrumentation adds 32 byte “redzones” around every allocation

- Store debugging metadata

Example of Memory Layout

```
void foo() {
  char buf[16];
  int x;
  // ...
}
```





$ **cat -n use-after-free.c**

```
 1  #include <stdlib.h>
 2  int main() {
 3    char *x = (char*)calloc(10, sizeof(char));
 4    free(x);
 5    return x[5];
 6  }
```

$ **clang -fsanitize=address -O1 -fno-omit-frame-pointer -g use-after-free.c**

% **./a.out**

```
=================================================================
==142161==ERROR: AddressSanitizer: heap-use-after-free on address 0x602000000015
at pc 0x00000050b550 bp 0x7ffc5a603f70 sp 0x7ffc5a603f68
READ of size 1 at 0x602000000015 thread T0
    #0 0x50b54f in main use-after-free.c:5:10
    #1 0x7f89ddd6452a in __libc_start_main
    #2 0x41c049 in _start

0x602000000015 is located 5 bytes inside of 10-byte region [0x602000000010,0x60200000001a)
freed by thread T0 here:
    #0 0x4d14e8 in free
    #1 0x50b51f in main use-after-free.c:4:3
    #2 0x7f89ddd6452a in __libc_start_main

previously allocated by thread T0 here:
    #0 0x4d18a8 in calloc
    #1 0x50b514 in main use-after-free.c:3:20
    #2 0x7f89ddd6452a in __libc_start_main

SUMMARY: AddressSanitizer: heap-use-after-free use-after-free.c:5:10 in main
[...]
==142161==ABORTING
```

Performance and Memory

- Why not executing the program with ASan protection enabled in production environments?
  - Address sanitizers add a huge performance overhead (2-3x slowdown)
  - Memory consumption is increased
- The performance overhead is due to instrumenting each memory read and write
- Hence, ASan-protected binaries are only executed in CI/CD pipelines
- They are also leveraged to perform fuzz testing

Scaling (Security) Testing

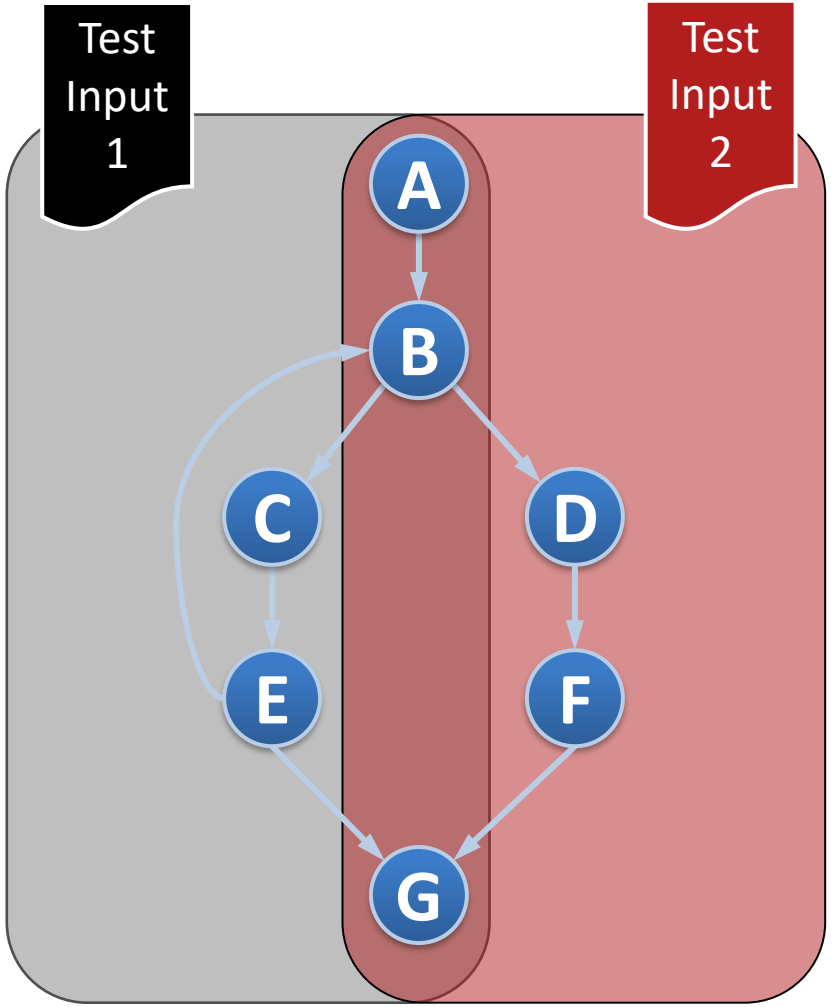
The Quest for Coverage

Goal: Perfect Test Coverage

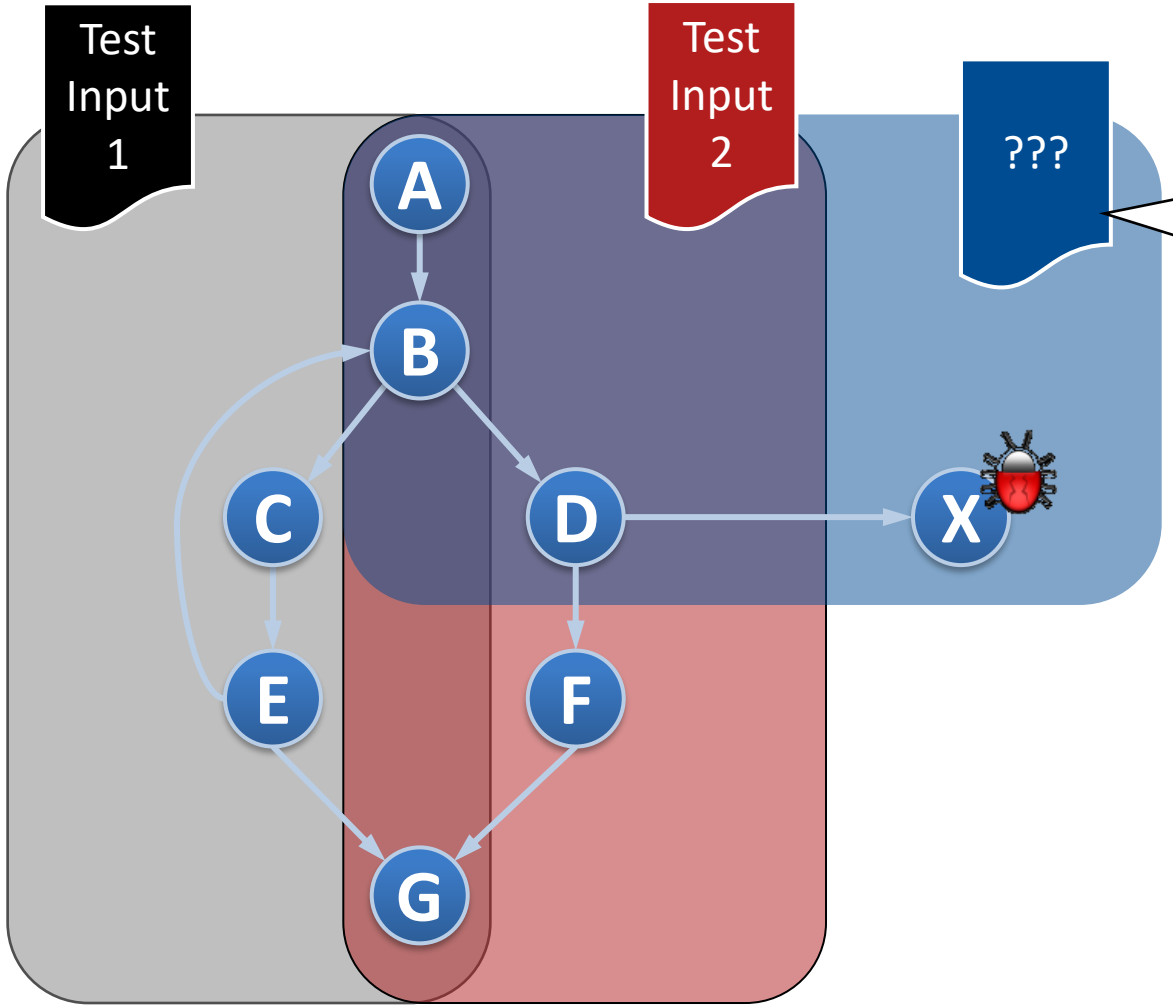
- Explore as many program states as possible
- Reduce testcases
  - All *usefully distinct* program states are explored
  - The testcases have minimum size
  - No redundancy in testcase corpus

Typically: One test case explores one path through the program

- E.g. path inside the Control-Flow-Graph (CFG)



Automatically Find New Paths



Can we generate this testcase automatically?

Symbolic Execution

Main Motivation

Question: “Can we generalize testing?”

Observation:

- One test explores one path through the program
- Can we systematically explore all paths through the program?

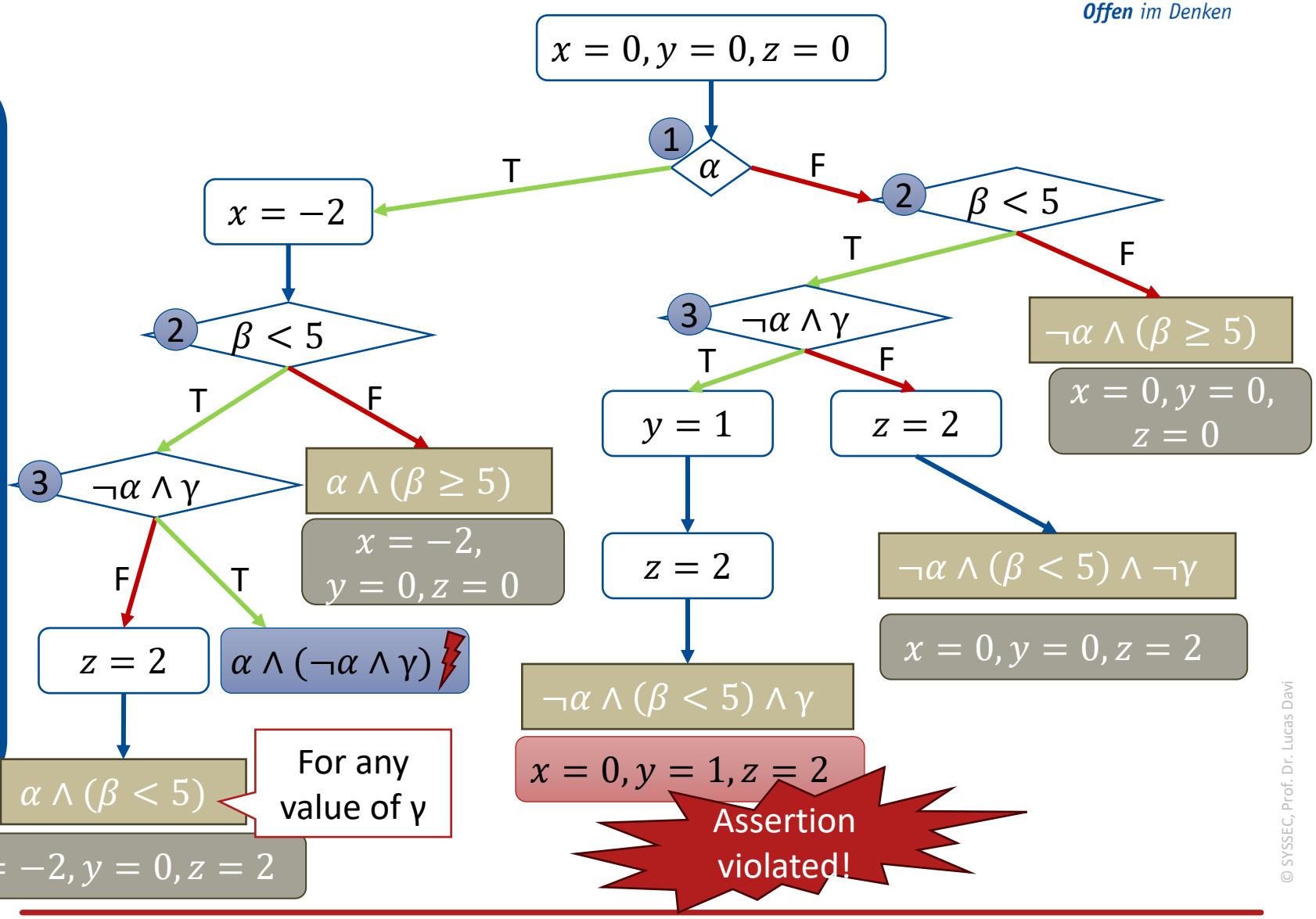
Key idea from the mid-70s:
Treat variables as symbols and collect constraints on the symbols

Motivating Example for Symbolic Execution

Assign symbolic value to input variables

```
func(a=α, b=β, c=γ) {
int x = 0, y = 0, z = 0;
1 if (a) {
    x = -2;
  }
2 if (b < 5) {
3   if (!a && c) {
      y = 1;
    }
    z = 2;
  }
assert(x + y + z != 3);
}
```

Goal: Find a path that violates the assert



Insights from Example

- We symbolically executed the program
- We collected 5 different path constraints
- Each path constraint represents a complete set of possible values
  - It shows that Unit Testing with (for example) b=1 and b=2 result in the same path

- We covered the whole program state space with 5 execution paths
  - We can now generate 5 concrete test-cases to cover all 5 paths

- Complete and sound analysis
  - Execution behavior formally verified

Symbolic Execution in Practice

Problem: computing systems are large and complex

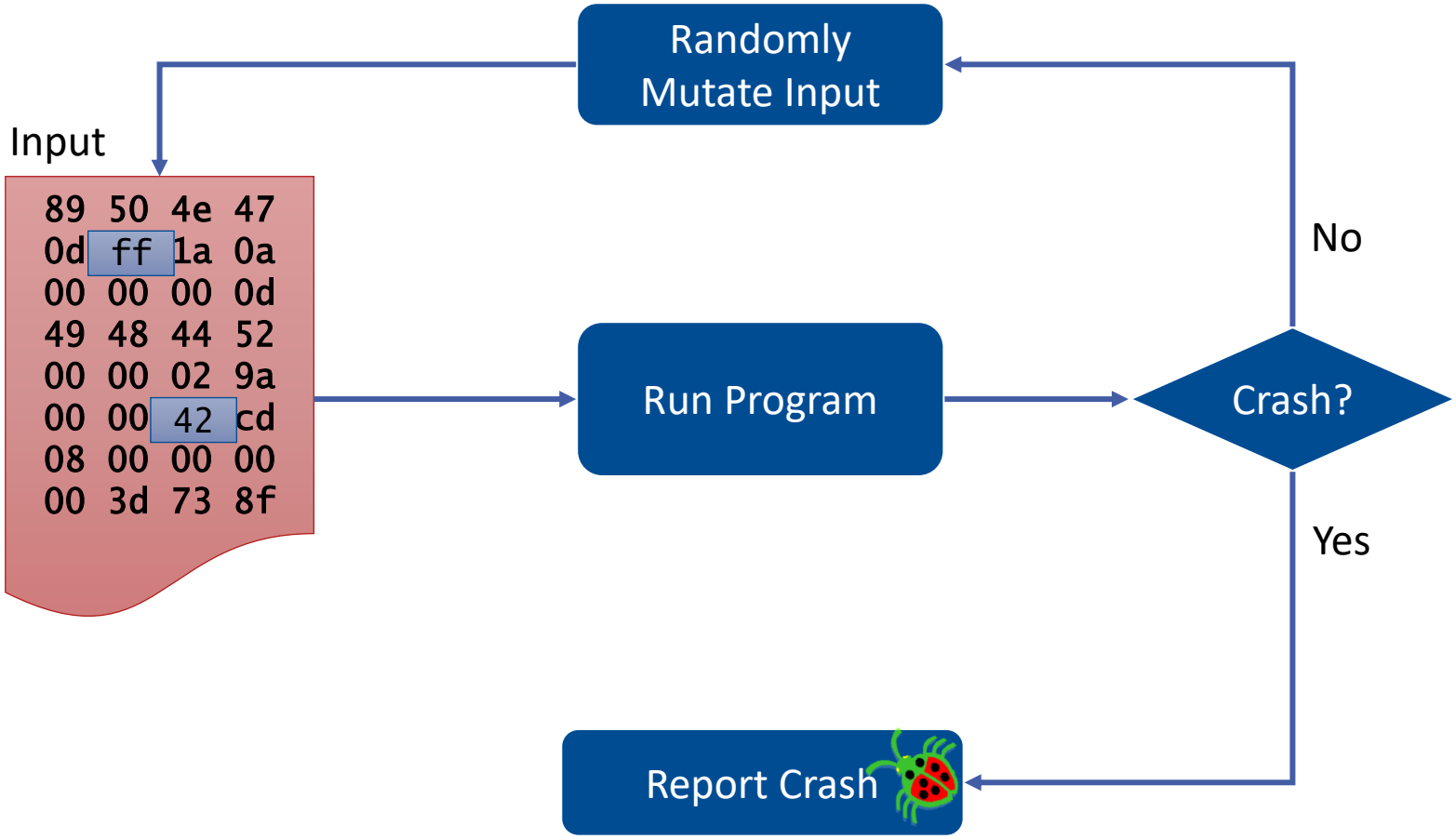
- Libraries, operating system, user input, network, file system, IPC
- Exponentially growing problem space (path explosion)

Strategy: limit scope and focus only on some paths

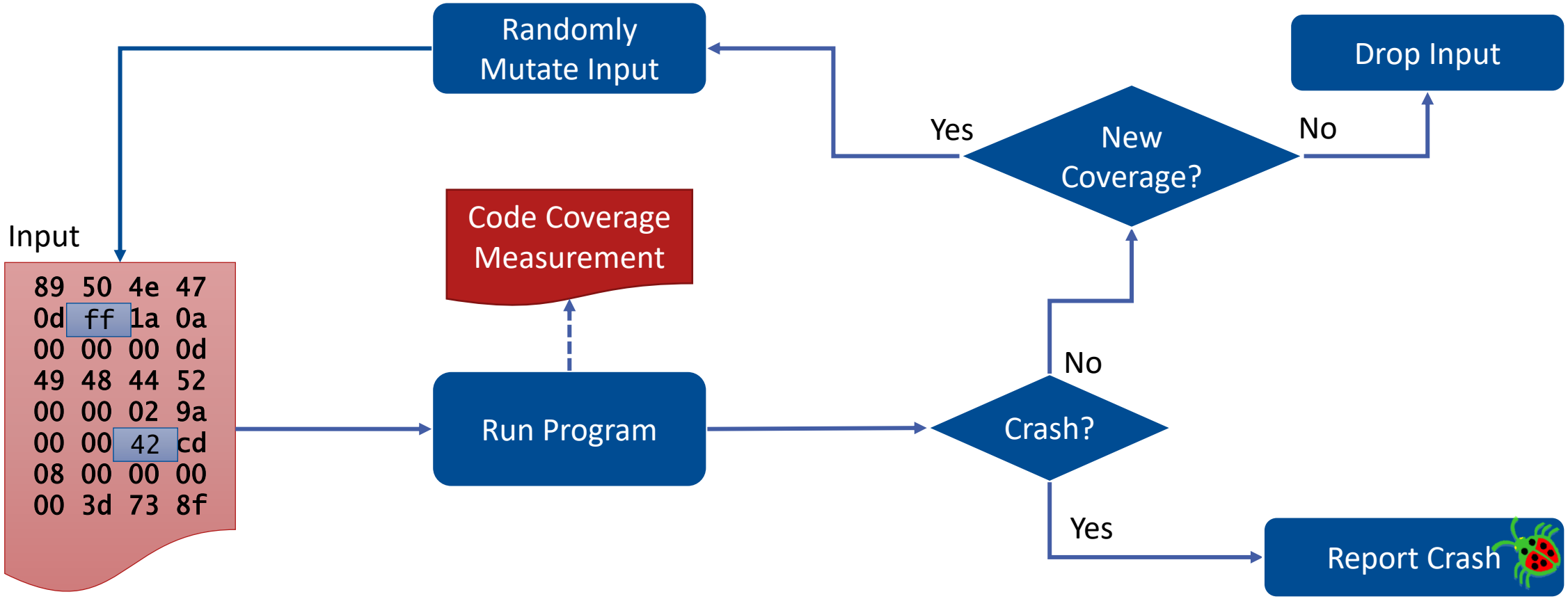
- Drop 100% coverage goal – infeasible anyway
- Concretize values when interacting with OS/IPC
- Use heuristics to find interesting paths

Fuzzing

Basic Fuzzer Workflow



Coverage-Guided Fuzzer Workflow



Internals of the Fuzz Engine

Dump fuzzing

- Just reads bytes from a random number generator

Making fuzzing smart

- Utilizing code coverage metrics
- Modern fuzz engines accept dictionaries and grammars
  - Dictionaries include interesting keywords from the specification
  - Grammars of well-specified protocols (HTTP, SQL, JSON)
- This allows fuzzers to utilize input that is not rejected anyways
- Fuzzers accept seed corpus which includes sample input files, e.g., MP3s for audio libraries, JPEGs for image processing

Some Remarks on Fuzzing

- Very hot topic of research in security and software engineering conferences
  - Fuzz engines exist for many architectures and languages (embedded, smart contracts, kernel, user-space, protocols)
- Fuzz engines leverage address sanitizers to detect invalid program states and provide metadata on the crash
  - It is still possible to fuzz a binary based on a dynamic binary instrumentation frameworks like QEMU
- Popular fuzzers: American Fuzzy Loop (AFL), Honggfuzz, libFuzzer
- Fuzzers find many bugs, but large-scale security bugs are still often found by manual analysis

Security Testing im CI/CD-Pipeline-Kontext

Unit Tests → *bei jedem Commit*

ASan → *nightly / pre-merge*

Fuzzing → *kontinuierlich / dedizierte Jobs*

Symbolic Execution → *gezielt für kritische Komponenten*

Was ist realistisch im Unternehmen?

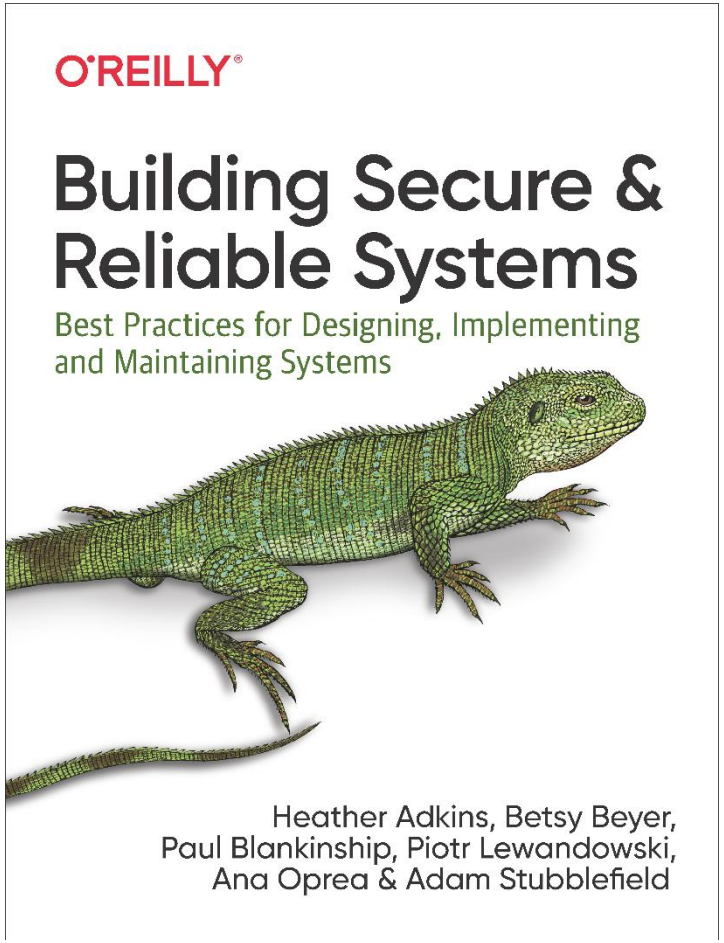
Technik	Typischer Einsatz
Unit Tests	Immer
Acceptance Tests	CI / Pre-Release
ASan	CI
Fuzzing	Security-kritische Projekte
Symbolic Execution	High-Assurance Software

Security Testing ≠ Security

- Testing findet Bugs, garantiert aber keine Sicherheit
- Kombination mit:
  - Secure Coding
  - Code Reviews
  - Threat Modeling

Referenzen

- Die Vorlesungsfolien wurden zum Teil mit Hilfe folgender Quellen erstellt:



UNIVERSITÄT DUISBURG ESSEN

Offen im Denken

Vorlesung Application Management, Wintersemester 2024/25

Vielen Dank für Ihre Aufmerksamkeit

Prof. Dr.-Ing. Lucas Vincenzo Davi
Fakultät für Informatik
Arbeitsgruppe Systemsicherheit

PALUNO
The Ruhr Institute for Software Technology