

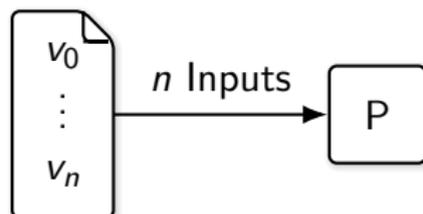
# Automatisches Testen

Testen  
SS 2022

Prof. Dr. Dirk Beyer,  
Thomas Lemberger

- Einführung
- Automatische Testerzeugung
  - Blackbox Testing (Random)
  - Greybox Testing (Distanzmetriken)
  - Whitebox Testing (Symbolic Execution)
- Testziel-Spezifikation mit FQL
- Mutations-Testen
- Übungen: PRTTest, AFL-fuzz, KLEE, CPAchecker, FShell, SRCIROR

# Automatische Testerzeugung



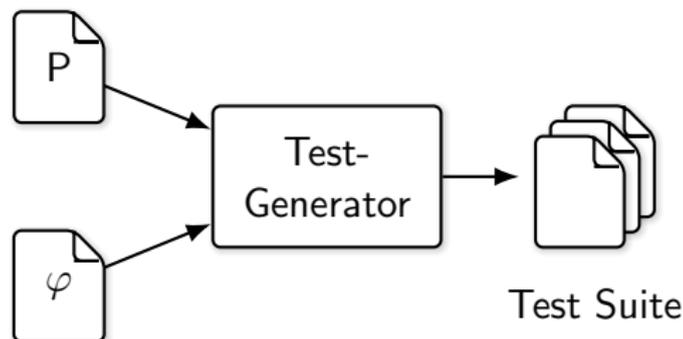
- ▶ Testfall-Beschreibung durch Testvektor  $\langle v_0, \dots, v_n \rangle$  mit  $n$  Programmeingabe-Werten  $v_0$  bis  $v_n$ .
- ▶ Menge von Testfällen: Test Suite.

```
int main() {
    int x = input();
    int y = input();
    while (y > 0 && x < 10) {
        if (x > 0) {
            x--;
        } else {
            x++;
        }
        y--;
    }
}
```

- ▶ Branch-Coverage:  $\{\langle 1, 2 \rangle\}$
- ▶ Condition-Coverage:  $\{\langle 1, 2 \rangle, \langle 10, 2 \rangle\}$

# Test Generator

- ▶ Definition Test-Generator:



- ▶ Input:
  - ▶ Program-under-test P
  - ▶ Test-Goal Spezifikation  $\varphi$
- ▶ Output: Test Suite (Menge von Testfällen)

▶ 3 Klassen von Test-Generatoren:



**Blackbox:** Verwendung von Inputs/Outputs



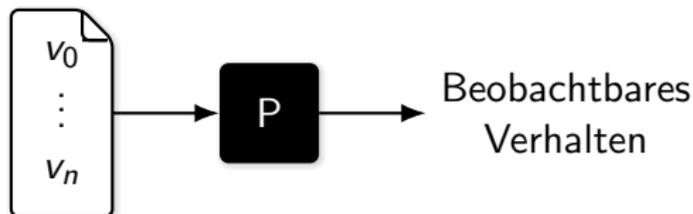
**Graybox:** + Verwendung der Programm-Struktur (Syntax)



**Whitebox:** + Verwendung der Programm-Semantik

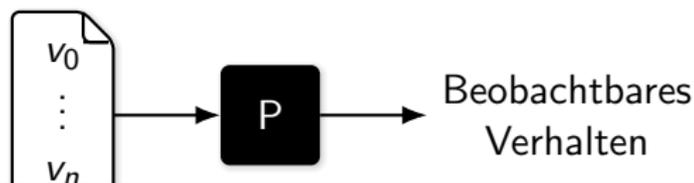
# Blackbox Testing

- ▶ Program-under-Test wird als Blackbox betrachtet



- ▶ Manchmal einzige Testmöglichkeit, z. B. für kompilierte Programme, Libraries, remote Server
- ▶ Extrem schnell (brute force)
- ▶ Extrem geringe Wahrscheinlichkeit, seltene Pfade abzudecken

- ▶ Program-under-Test wird als Blackbox betrachtet



Wahrscheinlichkeit, bei gleichmässiger Verteilung if-Branch zu betreten?

```
int x = input();  
if (x == 0) return 1;
```

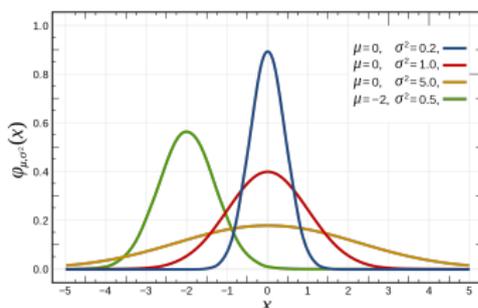
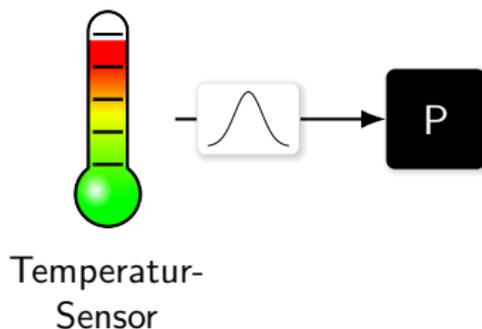
- ▶ Methode zum Blackbox-Testing
- ▶ Zufällige/beliebige Erzeugung von Testwerten
- ▶ Unabhängig von vorherigen Testausführungen
- ▶ Unterschiedliche Strategien:
  - ▶ Nach Zufallsverteilung (z.B. gleichmäßig)
  - ▶ Durch Evolution existierender Inputs
  - ▶ Grammatik-basiert

# Random Testing: Zufallsverteilung

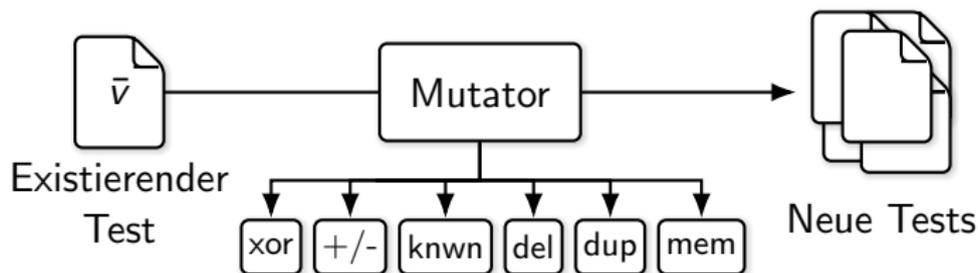
- ▶ Auswahl von Test-Inputs aus möglichen Input-Werten nach Zufallsverteilung
- ▶ Möglichkeiten:
  - ▶ Allgemeine Verteilung (gleichmässig, normal, etc.)
  - ▶ Erwartete Verteilung (kann von typischen Verteilungen stark abweichen)

# Random Testing: Zufallsverteilung

- ▶ Auswahl von Test-Inputs aus möglichen Input-Werten nach Zufallsverteilung
- ▶ Möglichkeiten:
  - ▶ Allgemeine Verteilung (gleichmässig, normal, etc.)
  - ▶ Erwartete Verteilung (kann von typischen Verteilungen stark abweichen)



# Random Testing: Evolutionär



- ▶ Anderer Name: Fuzzing
- ▶ Test-Generator erhält Anfangsmenge von Tests
- ▶ Versucht durch unterschiedliche Mutationen, weitere, vielfältige Tests zu erhalten
- ▶ Reihenfolge der Mutatoren (Mutations-Strategie) oft festgelegt
- ▶ Mutations-Operatoren: Deterministisch oder Zufällig
- ▶ Meist auf Bit- oder Byte-Ebene

# Random Testing: Input-Evolution (Beispiel Splicing)

```
int parseXml(char* input) {  
    if (strncmp(input, "<!ATTLIST", 9)) {  
        // ...  
    } else if (strncmp(input, "<!ELEMENT", 9)) {  
        // ...  
    }  
    if (strstr(input, "html") != NULL) {  
        // handle html  
    }  
}
```

- ▶ Existierende Testfälle:

<	!	A	T	T	L	I	S	T		h	t	m	l
<	!	E	L	E	M	E	N	T		h	e	a	d

- ▶ Neuer Testfall:

<	!	E	L	E	M	E	N	T		h	t	m	l
---	---	---	---	---	---	---	---	---	--	---	---	---	---

## Random Testing: Input-Evolution (Beispiel Bit Mask)

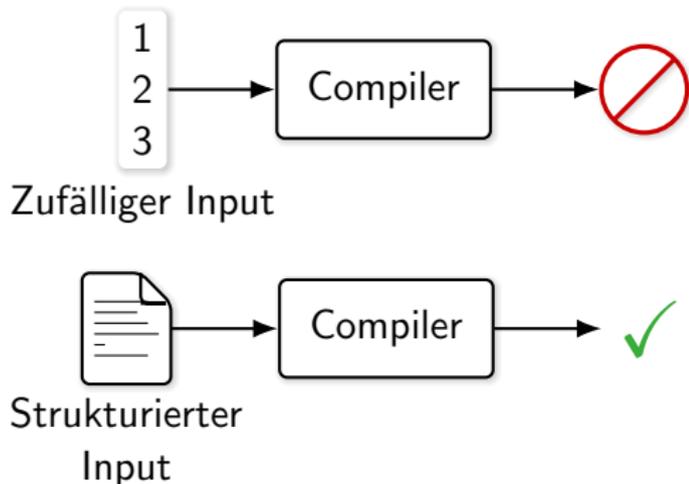
```
int parseXml(char* input) {  
    if (strncmp(input, "<!ATTLIST", 9) ) {  
        // further processing of input  
    }  
}
```

- ▶ **Nur** Input relevant, der mit <!ATTLIST beginnt
- ⇒ Maskiere erste 9 Characters in gültigem Test-Case, damit sie nicht mutiert werden



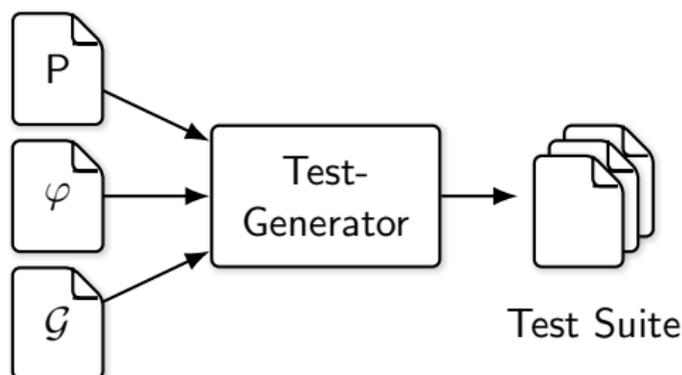
# Random Testing: Grammatik-basiert I

```
void compile(char* input) {  
    if (!isValid(input)) {  
        exit(1);  
    } else {  
        // all the logic  
    }  
}
```



## Random Testing: Grammatik-basiert II

- ▶ Oft erwarten Programme eine bestimmte Struktur ihrer Inputs
- ▶ Bildbearbeitung, Browser, Compiler, Protokolle, etc.
- ▶ Zufällige Inputs erfüllen Struktur mit extrem hoher Wahrscheinlichkeit nicht
  - ⇒ werden direkt abgelehnt
  - ⇒ kein sinnvolles Testen möglich
- ▶ Lösung: Generiere Inputs entlang Grammatik ( $\mathcal{G}$ )



# Random Testing: Grammatik-basiert (Beispiel)

## Beispiel-Grammatik:

```
<expr> ::= <term> "+" <expr>
         | <term>
```

```
<term>  ::= <factor> "*" <term>
         | <factor>
```

```
<factor> ::= "(" <expr> ")"
          | <const>
```

```
<const> ::= 0 | 1 | 2 | ...
```

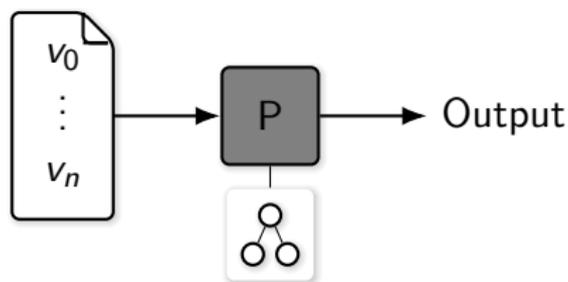
## Zufällige Generierung eines Test:

1. Initialisiere Test mit Start-Symbol  
(*test* = <expr>)
2. Solange Non-Terminale in Test vorhanden: Ersetze jeden Non-Terminal durch zufällig gewählte, rechte Seite.
3. Sobald keine Non-Terminale mehr vorhanden: Gebe Test zurück.

- ▶ Test-Suites wachsen extrem schnell
- ⇒ Extrem viele unnötige/sinnlose Tests (Test-Suite Reduction)
- ▶ Für evolutionäres Testen: Test-Vektoren wachsen (Trimming)

# Graybox Testing

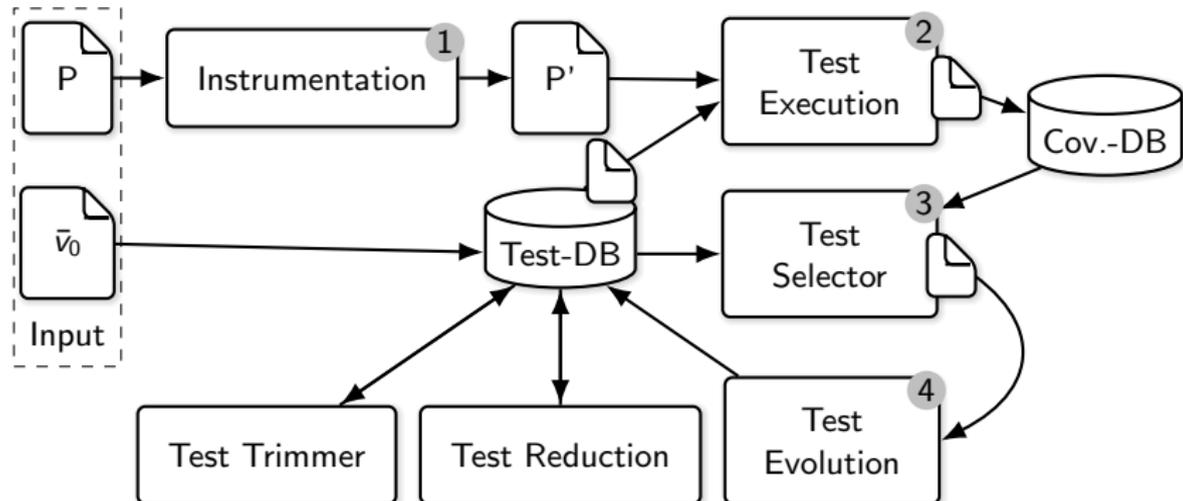
# Graybox Testing: Überblick



- ▶ Nutzt Programm-Syntax und konkrete Testabdeckung
- ▶ **Keine** Semantik!
- ▶ Test-Erstellung: weiterentwickelte Blackbox-Techniken
- ▶ Eingeschränkt auch auf kompilierten Programmen verwendbar
- ▶ Extrem schnell (directed brute force)
  - ▶ Balance zwischen Brute Force und Testoptimierung
- ▶ Potential, auch seltene Pfade abzudecken



# AFL: Workflow



# AFL: Instrumentation

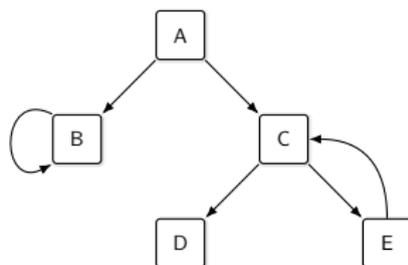
- ▶ Zu Beginn jedes Code-Blocks  $N$  :

```
cur_location = <RANDOM_ID>;  
shared_mem[cur_location ^ prev_location]++;  
prev_location = cur_location >> 1;
```

- ▶ shared\_mem wird als zusätzlicher Eingabe-Parameter zu Programm hinzugefügt

shared\_mem

Transition	Hit Count
A-B	1
B-B	3
A-C	0
C-D	0
C-E	0
E-C	0



- ▶ Random IDs für Blöcke: Kollisionen möglich, aber dafür keine aufwendige Berechnung nötig

*"hmm, this does something new!"*

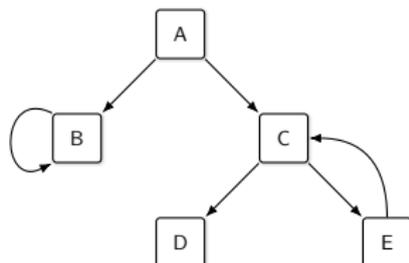
- ▶ Ausführung von noch nicht ausgeführten Tests (pro Evolutions-Level)
- ▶ Resultat pro Ausführung: shared\_mem-Map mit Hit Counts der Branch-Transitions (+ Test-Laufzeit)
- ▶ Falls mind. eine bisher unbesuchte Branch-Transition getroffen wurde: Test-ID und Map wird in Coverage-Datenbank gespeichert
- ▶ Beispiel:

A-C, C-E

A-C, C-D

A-C, C-E, E-C, C-D

A-C, C-E, E-C, C-E



*"hmm, this does something new!"*

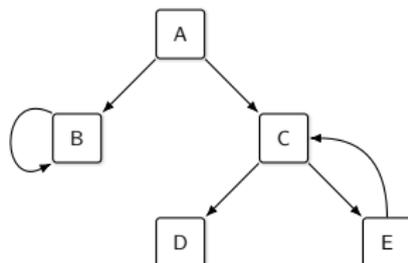
- ▶ Ausführung von noch nicht ausgeführten Tests (pro Evolutions-Level)
- ▶ Resultat pro Ausführung: shared\_mem-Map mit Hit Counts der Branch-Transitions (+ Test-Laufzeit)
- ▶ Falls mind. eine bisher unbesuchte Branch-Transition getroffen wurde: Test-ID und Map wird in Coverage-Datenbank gespeichert
- ▶ Beispiel:

A-C, C-E ✓

A-C, C-D

A-C, C-E, E-C, C-D

A-C, C-E, E-C, C-E



# AFL: Test Execution + Selection I

*"hmm, this does something new!"*

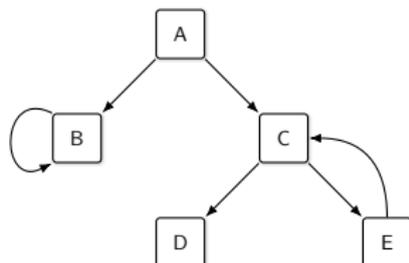
- ▶ Ausführung von noch nicht ausgeführten Tests (pro Evolutions-Level)
- ▶ Resultat pro Ausführung: shared\_mem-Map mit Hit Counts der Branch-Transitions (+ Test-Laufzeit)
- ▶ Falls mind. eine bisher unbesuchte Branch-Transition getroffen wurde: Test-ID und Map wird in Coverage-Datenbank gespeichert
- ▶ Beispiel:

A-C, C-E ✓

A-C, C-D ✓

A-C, C-E, E-C, C-D

A-C, C-E, E-C, C-E



*"hmm, this does something new!"*

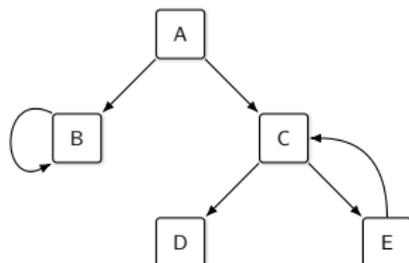
- ▶ Ausführung von noch nicht ausgeführten Tests (pro Evolutions-Level)
- ▶ Resultat pro Ausführung: shared\_mem-Map mit Hit Counts der Branch-Transitions (+ Test-Laufzeit)
- ▶ Falls mind. eine bisher unbesuchte Branch-Transition getroffen wurde: Test-ID und Map wird in Coverage-Datenbank gespeichert
- ▶ Beispiel:

A-C, C-E ✓

A-C, C-D ✓

A-C, C-E, E-C, C-D ✓

A-C, C-E, E-C, C-E



*"hmm, this does something new!"*

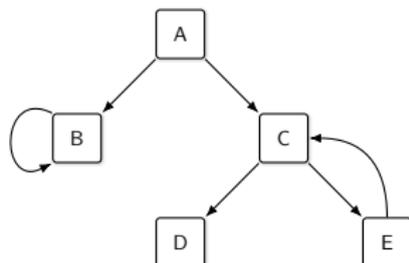
- ▶ Ausführung von noch nicht ausgeführten Tests (pro Evolutions-Level)
- ▶ Resultat pro Ausführung: shared\_mem-Map mit Hit Counts der Branch-Transitions (+ Test-Laufzeit)
- ▶ Falls mind. eine bisher unbesuchte Branch-Transition getroffen wurde: Test-ID und Map wird in Coverage-Datenbank gespeichert
- ▶ Beispiel:

A-C, C-E ✓

A-C, C-D ✓

A-C, C-E, E-C, C-D ✓

A-C, C-E, E-C, C-E X



- ▶ Weitere Metrik um Tests auszuwählen:
- ▶ Branch-Hit-Counts werden in Buckets eingeteilt



- ▶ Tests, die bei einem Hit-Count in anderen Bucket als bisherige Tests fallen, gelten zusätzlich als interessant
- ▶ Alle Tests ohne neue Branch-Transition oder neuen Bucket werden verworfen
- ▶ Verbleibende Tests werden für Evolution verwendet

## AFL: Test Execution + Selection II

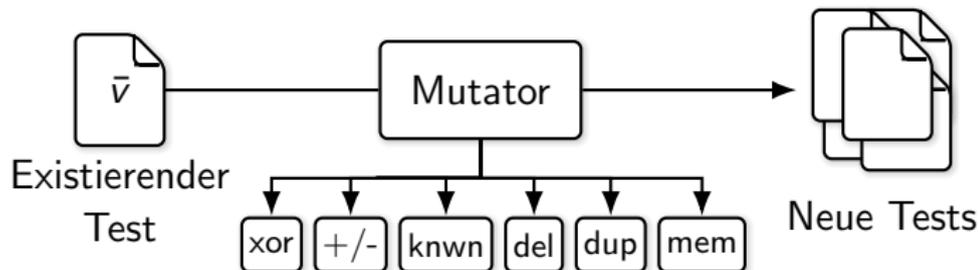
- ▶ Weitere Metrik um Tests auszuwählen:
- ▶ Branch-Hit-Counts werden in Buckets eingeteilt

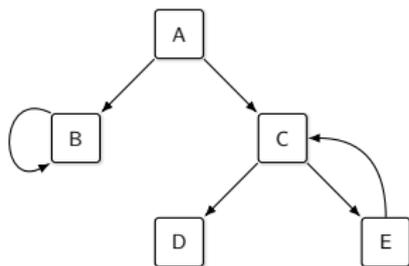


Wieso diese Buckets?

# AFL: Test Evolution (Fuzzing)

- ▶ Zwei Phasen:
  1. Deterministische Mutationen
  2. Zufällige Mutationen
- ▶ Selbe Mutations-Operatoren wie beim Random Testing





► Tests in Test-Set:

1. A-C, C-E
2. A-C, C-D
3. A-C, C-E, E-C, C-D

► Test 3 deckt alle Transitionen ab

⇒ Nach AFL-Logik überflüssig, 1 und 2 zu mutieren

⇒ Periodische Analyse der bestehenden Test-Suite und Auswahl ausreichender und „guter“ Tests

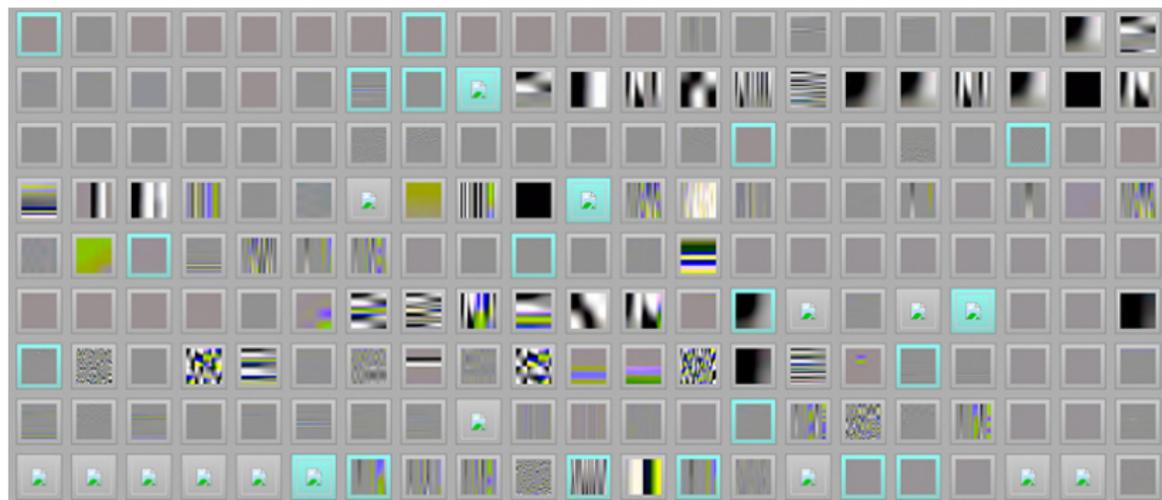
# AFL: Test Reduction Algorithm

1. Für jeden ausgeführten, aber noch nicht mutierten Test:  
Vergebe Score basierend auf Ausführungszeit und  
Test-Vektor-Länge
  2. Sortiere Tests aufsteigend nach Score
  3. Für jede Branch-Transition, die bereits abgedeckt wurde, und  
für die noch kein Test ausgewählt wurde:
    - 3.1 Wähle Test mit niedrigstem Score
    - 3.2 Wähle selben Test für alle anderen Transitions, die er abdeckt
- ▶ Auswahl ist ca. 10-20% aller Kandidaten (in Experimenten)
  - ▶ Nicht ausgewählte Tests werden **nicht** verworfen, sondern nur  
mit hoher Wahrscheinlichkeit ausgelassen

- ▶ Große Test-Inputs verlangsamen Testing extrem (Beispiel: Compiler, Bildbearbeitung-Programm)
  - ▶ Wahrscheinlichkeit, dass große Test-Inputs notwendig sind, sehr gering
- ⇒ Reduziere Test-Inputs auf notwendigen Inhalt
- ▶ AFL minimiert nicht, sondern setzt Fenster für Lösch-Kandidaten abhängig von Ausführungsdauer
  - ▶ Lange Testausführung → großes Fenster, um weniger Versuche zu verbrauchen
  - ▶ Vorgehen:
    1. Lösche Daten in aktuellem Fenster
    2. Führe neuen Test aus
    3. Selbe Coverage ⇒ ersetze alten Test durch aktuellen
    4. Andere Coverage ⇒ mache Löschen rückgängig

# AFL: Beispiel djpeg

- ▶ Beispiel, dafür, wie viel Information der Kontroll-Fluss beinhalten kann
- ▶ Program under test: djpeg (dekomprimiert jpeg-Bilder)
- ▶ Einziger test seed: hello
- ▶ Nach 6 Stunden:



# Whitebox Testing



- ▶ Benutzt volle Programm-Semantik
- ▶ Kann alle Testziele finden
- ▶ Oft Heuristiken  $\Rightarrow$  Bei unerwarteten Aufgaben langsam

- ▶ Beispiel-Ansatz: KLEE  
<https://klee.github.io/>
- ▶ Symbolic-Execution-Engine

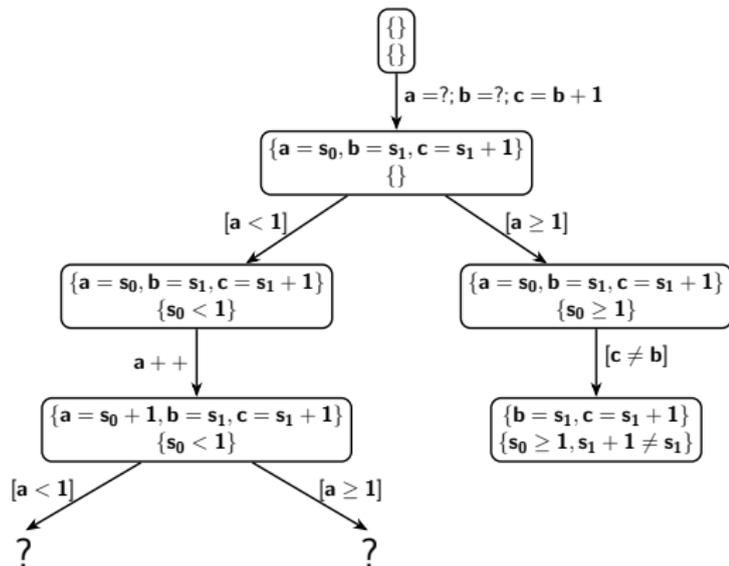
# Symbolic Execution

- ▶ Keine konkreten Werte für Inputs, sondern symbolische Werte
- ▶ **Symbolic Memory:** Map  
Speichert (symbolische) Variablenzuweisungen
- ▶ **Path Constraints:** Menge von (Un)gleichungen  
Beschränken die symbolischen Werte

# Symbolic Execution

- ▶ Keine konkreten Werte für Inputs, sondern symbolische Werte
- ▶ **Symbolic Memory:** Map  
Speichert (symbolische) Variablenzuweisungen
- ▶ **Path Constraints:** Menge von (Un)gleichungen  
Beschränken die symbolischen Werte

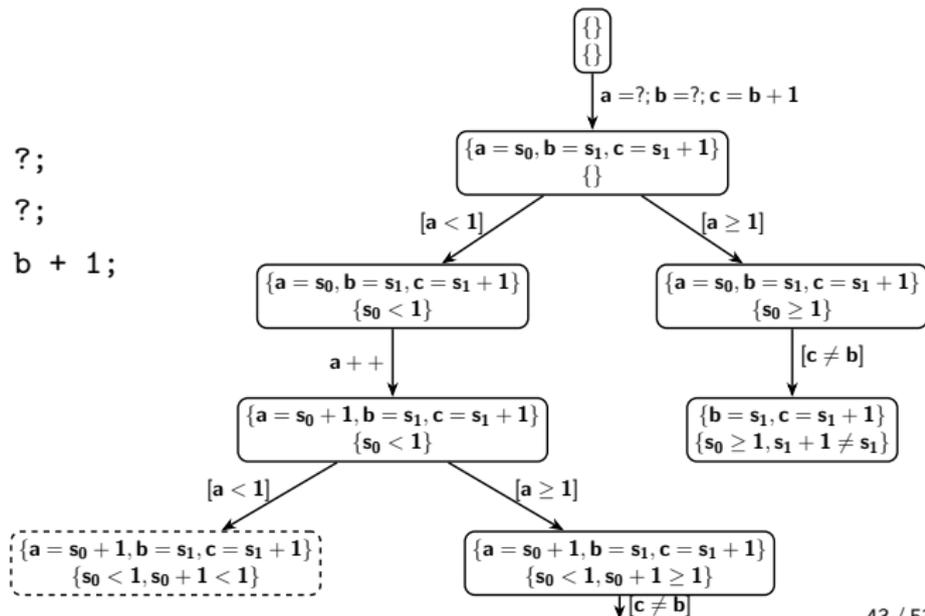
```
unsigned char a = ?;  
unsigned char b = ?;  
unsigned char c = b + 1;  
while (a < 1)  
    a++;  
if (c == b)  
    error();
```



# Symbolic Execution

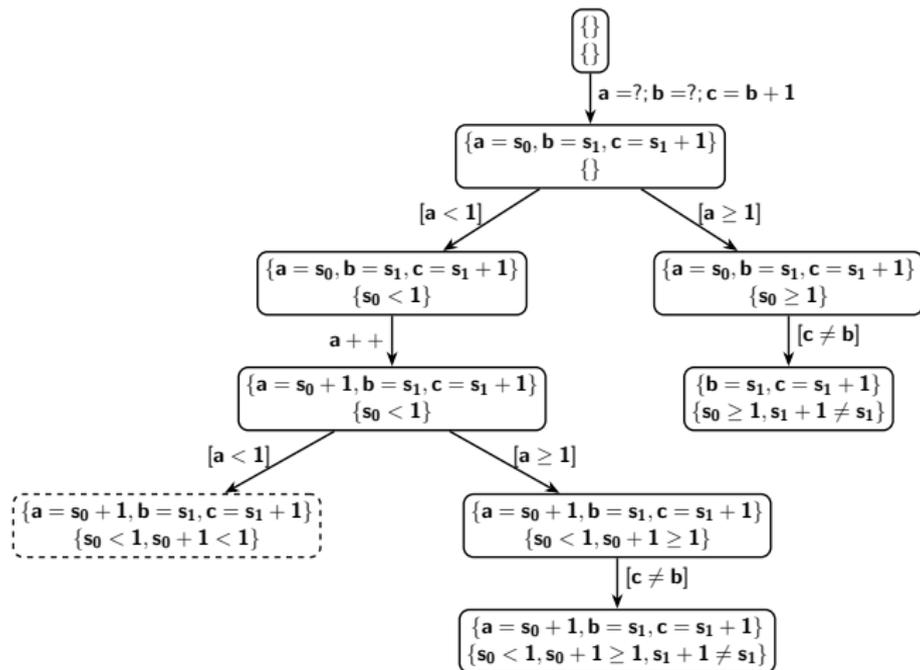
- ▶ Keine konkreten Werte für Inputs, sondern symbolische Werte
- ▶ **Symbolic Memory:** Map  
Speichert (symbolische) Variablenzuweisungen
- ▶ **Path Constraints:** Menge von (Un)gleichungen  
Beschränken die symbolischen Werte

```
unsigned char a = ?;  
unsigned char b = ?;  
unsigned char c = b + 1;  
while (a < 1)  
    a++;  
if (c == b)  
    error();
```



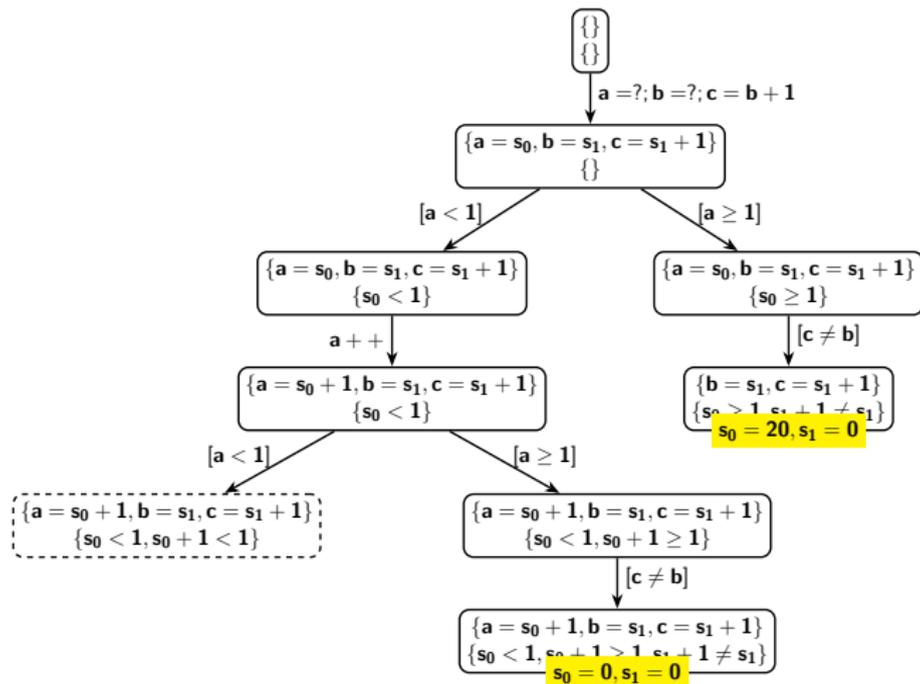
# Symbolic Execution zur Testfallerzeugung

- ▶ Berechneter Baum: *Symbolic Execution Tree*
- ▶ Blätter des Symbolic Execution Tree: Äquivalenzklassen für Testinputs im Bezug auf Programmpfade



# Symbolic Execution zur Testfallerzeugung

- ▶ Berechneter Baum: *Symbolic Execution Tree*
- ▶ Blätter des Symbolic Execution Tree: Äquivalenzklassen für Testinputs im Bezug auf Programmpfade



## Blackbox

- ▶ Extrem schnell
- ▶ Oberflächliche Testabdeckung
- ▶ Viele Redundante Tests

## Graybox

- ▶ Schnell für viele Testziele
- ▶ Tiefgehende Testabdeckung
- ▶ Redundante Tests

## Whitebox

- ▶ Schnell für wenig Testziele
- ▶ Langsam für viele **oder** wenige Testziele
- ▶ Vollständige Testabdeckung
- ▶ Keine redundanten Tests

# Übungsblock 1: Blackbox Testing

https:

//gitlab.lrz.de/lebergerth/blockseminar-testen-ss-22

Mittagspause

# Besprechung Übungsblock 1

## Übungsblock 2: Whitebox Testing und Greybox Testing

https:

//gitlab.lrz.de/lebergerth/blockseminar-testen-ss-22

## Besprechung Übungsblock 2