Vorlesung Fortgeschrittene Softwaretechnik

Wintersemester 2024/25

Prof. Dr. Stephan Diehl

Lucas Kreber, M.Sc.

Informatik

Universität Trier

Software Quality

- **ISO/IEC 9126 Standard**

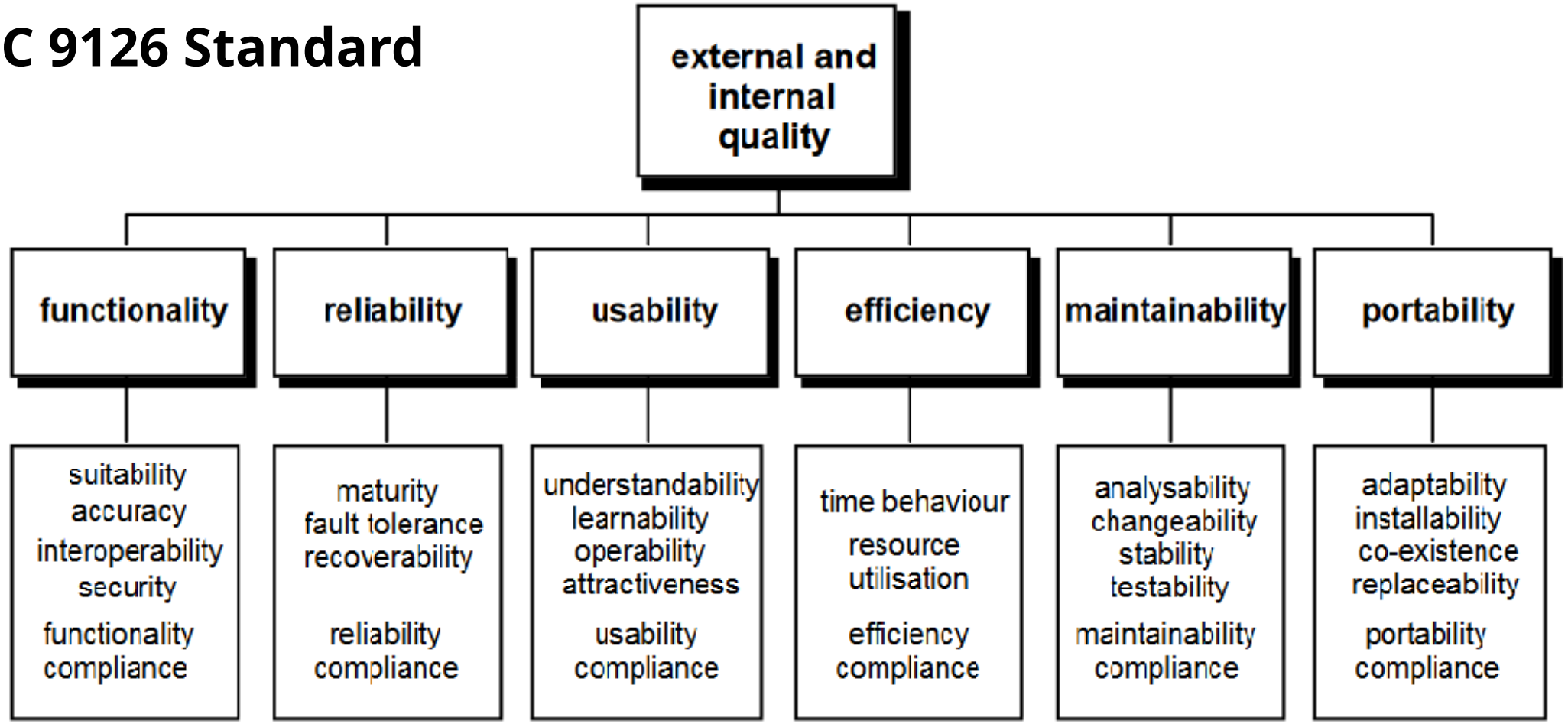


Figure 4 – Quality model for external and internal quality

https://www.cse.unsw.edu.au/~cs3710/PMmaterials/Resources/9126-1%20Standard.pdf

Schlagwörter

- Acceptance testing
- Installation testing
- Alpha and beta testing
- Regression testing
- Performance testing
- Security testing
- Stress testing
- Back-to-back testing
- Recovery testing
- Configuration testing
- Usability testing:
- Test-driven development

Schlagwörter

- Acceptance testing
- Installation testing
- Alpha and beta testing
- Regression testing
- Performance testing
- Security testing
- Stress testing
- Back-to-back testing
- Recovery testing
- Configuration testing
- Usability testing
- Test-driven development

Schlagwörter

Schlagwörter

- **Acceptance testing:** Letzter Test vor dem Einsatz/der Auslieferung prüft, ob die Anforderungen der Kund:innen erfüllt sind.
- **Installation testing:** Überprüfung, ob Software richtig installiert und konfiguriert ist und ggf. wieder deinstalliert werden kann.
- **Alpha testing:** Interne Überprüfung durch Entwickler:innen/Testteam vor Einsatz/Auslieferung
- **Beta testing:** Test der (fast) fertigen Software in der realen Umgebung durch externe Nutzer:innen vor tatsächlichem Einsatz/Auslieferung.
- **Regression testing:** Automatisches oder regelmässiges Testen der Software nach Änderungen.
- **Performance testing:** Test der Ressourcennutzung (Speicher, Zeit, Durchsatz, ..) insbesondere bei hoher Last
- **Security testing:** Test, ob Software vor sicherheitsrelevanten Bedrohungen (unautorisierter Zugriff, Datenverlust, etc.) geschützt ist.
- **Stress testing:** Test, ob Software unter extremen Belastungen, die über den normalen Betrieb hinausgehen, noch zuverlässig oder sicher funktioniert.
- **Back-to-back testing:** Vergleich der Ergebnisse mit einer Vorgängerversion oder Referenzimplementierung.
- **Recovery testing:** Überprüfung, ob und in welchem Umfang die Software sich nach einem Ausfall oder Fehler wiederherstellen und weiterarbeiten kann.
- **Configuration testing:** Überprüfung, ob die Software in verschiedenen Hardware- und Softwarekonfigurationen korrekt funktioniert.
- **Usability testing:** Verwendung der Software durch echte Benutzer, um die Benutzerfreundlichkeit, Effizienz und Zufriedenheit zu messen.
- **Test-driven development:** Entwicklungsmethode, bei der zuerst automatisierte Tests für eine Funktion geschrieben werden, bevor der eigentliche Code für diese Funktion implementiert wird.

Testen von Software

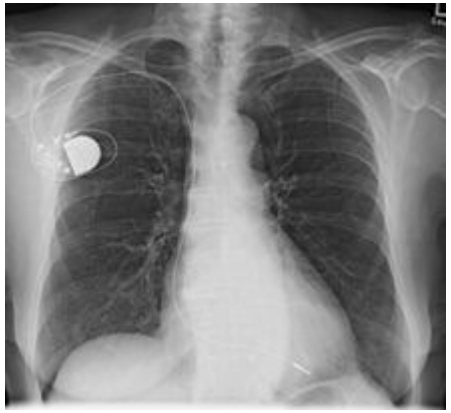
- Die analytische Qualitätssicherung stützt sich im allgemeinen auf den Begriff **Fehler** ab:
  - jede **Abweichung der tatsächlichen Ausprägung** eines Qualitätsmerkmals von der vorgesehenen Soll-Ausprägung
  - jede **Inkonsistenz** zwischen Spezifikation und Implementierung
  - jedes **strukturelle Merkmal** des Programmtextes, das ein fehlerhaftes Verhalten des Programms verursacht
- Ziel des **Testens** ist, durch gezielte Programmausführung Fehler zu erkennen.

Program testing can be used to show the presence of bugs, but never to show their absence. **(Edsger W. Dijkstra)**

Größe kritischer Softwaresysteme

- „So bringt es ein Herzschrittmacher auf 80.000 Codezeilen [JM13], das Elektrofahrzeug Chevrolet Volt auf 10 Millionen Codezeilen [LD10] und zum Vergleich die Flugsoftware der **Boeing 787** auf **14 Millionen** [DL11]).“

80.000 LOC



10.000.000 LOC

14.000.000 LOC

Quelle: Effiziente Risikoanalyse anhand praktischer Erfahrungsbeispiele, Manfred Holzbach, Lecture Notes in Informatics, Band 619, Springer , 2016.

Fehlerdichte

- Industriedurchschnitt:
  - **30-85** Fehler pro 1000 Kodezeilen werden vor der Auslieferung **erkannt und behoben**.
  - **0.5** bis **3** Fehler pro 1000 Kodezeilen werden **nicht** vor Auslieferung der Software **erkannt**.
- Selbst bei extrem kritischem und sorgfältig inspiziertem Programmkode bleiben 1 Fehler auf 10.000 Kodezeilen.

$$\frac{14.000.000}{10.000} = 1.400$$

[Quelle: NASA Study on Flight Software Complexity, Daniel L. Dvorak, 2009, https://www.nasa.gov/pdf/418878main_FSWC_Final_Report.pdf]

Was ist ein Fehler ?

Das Wort **Fehler** wird für verschiedene Konzepte gebraucht:

1. Der **Programmierer** macht einen ***Fehler***
2. und hinterlässt einen ***Fehler*** im **Programmcode**.
3. Wird dieser **ausgeführt**, haben wir einen ***Fehler*** im Programmzustand,
4. der sich als ein ***Fehler*** nach **außen manifestiert**.

Was ist ein Fehler ?

Diese unterschiedlichen ”Fehler“ werden besser so bezeichnet:

1. Der Programmierer begeht einen ***Irrtum*** (mistake)

2. und hinterlässt einen ***Defekt*** (defect) im Programmcode.

3. Wird dieser ausgeführt, haben wir eine ***Infektion*** im Programmzustand,

4. der sich als ein ***Fehlschlagen*** (failure) nach außen manifestiert.

Vom Fehlschlagen zum Defekt

Es gilt:

Fehlschlagen $\Rightarrow$ Infektion $\Rightarrow$ Defekt ($\Rightarrow$ Irrtum)

- aber nicht umgekehrt – **nicht jeder Defekt führt zu einem Fehlschlagen!**
- Dies ist das ***Kernproblem des Testens!***

Was ist ein Fehler ?

Spezifikation:

for input i,
give output $2*i^3$
(i=6 liefert 432)

Programmcode:

```
i=input(STDIN);
i=double(i);
i=power(i,3);
output(STDOUT,i);
```

Ausgabe:

input: 6
doubling input..
computing power..
output: 1728

Was ist ein Fehler ?

Spezifikation:

for input i,
give output $2*i^3$
(i=6 liefert 432)

Defekt

Programmcode:

```
i=input(STDIN);
i=double(i);
i=power(i,3);
output(STDOUT,i);
```

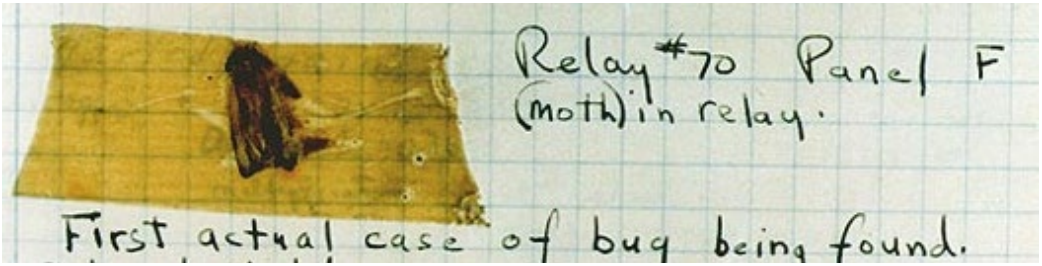
Ausgabe:

Fehlschlag

```
input: 6
doubling input..
computing power..
output: 1728
```

Beispiel – Berechnung des Maximums dreier Zahlen:

```
int max3(int x, int y, int z) {
  int ret;
  if (x > y)
    ret = x;
  else
    ret = max(y, z);
  return ret;
}
```



Erster „Computer-Bug“, 1947
(U.S. Naval Historical Center Online Library Photograph NH 96566-KN)

- Fehlschlagen, `max3(5, 2, 9)` liefert **5**
- Infektion **ret** hat den Wert **5**
- Defekt statt **ret = x** muss es **ret = max(x, z)** heißen
- **Kern des Debugging** = Schließen vom Fehlschlag auf Defekt (engl. bug = Fehler, Käfer)

Auswahl von Testfällen

- Programm hat in der Regel unendlich viele mögliche Eingaben. **Problem**: Welche Eingaben wählt man als Testfälle.

- **Funktionale Verfahren** (Black-box)
  - Auswahl nach *Eigenschaften der Eingabe* oder der Spezifikation

- **Strukturtests** (White-box)
  - Auswahl nach *Aufbau des Programms*
  - Ziel im Strukturtest: hohen **Überdeckungsgrad** erreichen, Testfälle sollen möglichst viele Aspekte der Programmstruktur abdecken (Kontrollfluss, Datenfluss)

Äquivalenzklassen:

- Beispiel: Das System erwartet eine Eingabe zwischen 100 und 999 inklusive.
- Daraus ergeben sich 3 Äquivalenzklassen für die Eingabe
  - kleiner als 100
  - 100 bis 999
  - größer als 999
- Wir testen das System mit charakteristischen Werten aus jeder Äquivalenzklasse:
  - z.B.: 50 (ungültig), 500 (gültig), 1500 (ungültig).

Grenzfälle:

- An den Grenzen der zulässigen Eingabewerte gibt es häufig Programmfehler.
- Beispiel: Das System erwartet eine Eingabe zwischen 100 und 999 inklusive.
  - Die Grenzen sind 100 und 999.
- Daher testen wir mit den Werten:

99 100 101 — Untere Grenze

998 999 1000 — Obere Grenze

Whitebox-Testen

- Verwende Information über die **interne Struktur** des Programms für die Entwicklung der Tests
- Idealvorstellung: Untersuche jeden möglichen Lauf des Programms
- In der Praxis nicht machbar!
- Mögliche Alternative: Versuche **jede Anweisung des Programms mindestens einmal** zu testen
- Beispiel:

```
if (x > 5)
  { out.printf(‘‘hello’’); }
else
  { out.printf(‘‘bye’’); }
```

- Es gibt zwei mögliche Pfade in diesem Programmkode:
  - je einen für $x > 5$ bzw. $x \leq 5$.

Pfadtesten

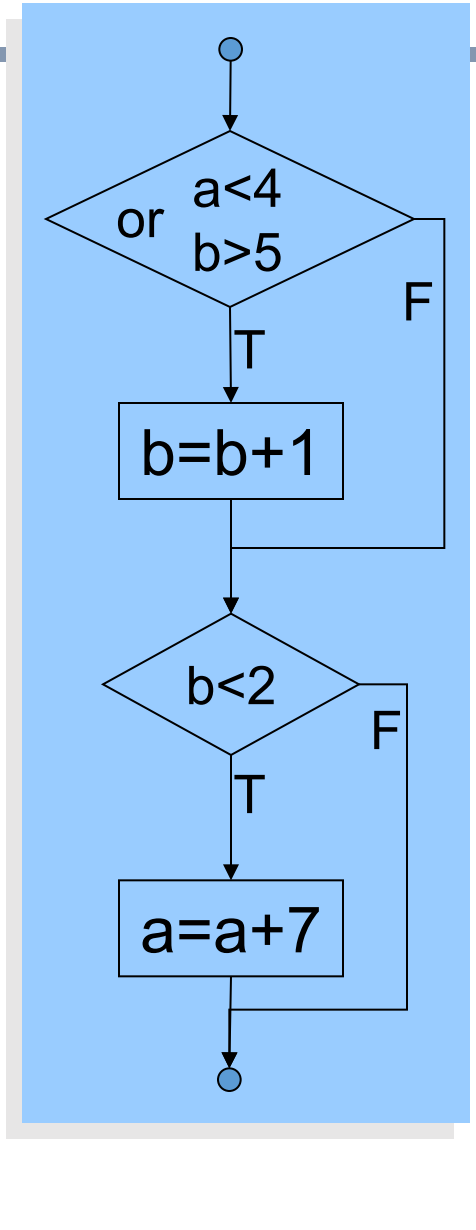
- Anweisungsüberdeckung (Statement coverage) ist erfüllt, wenn jede Anweisung (außer Kontrollflußanweisungen) wenigstens einmal für eine Testmenge ausgeführt wurde.
- Zweigüberdeckung (Branch coverage) ist erfüllt, wenn jede Kante im Kontrollflußgraphen eines Programms wenigstens einmal für eine Testmenge durchlaufen wurde.
- Pfadüberdeckung (Path coverage) ist erfüllt, wenn die Testmenge für jeden möglichen Kontrollpfad im Flußgraphen einen Testfall enthält, der diesen durchläuft.
  - Problem: Anzahl der Pfade wächst exponentiell mit der Anzahl an Verzweigungen.

Pfadtesten

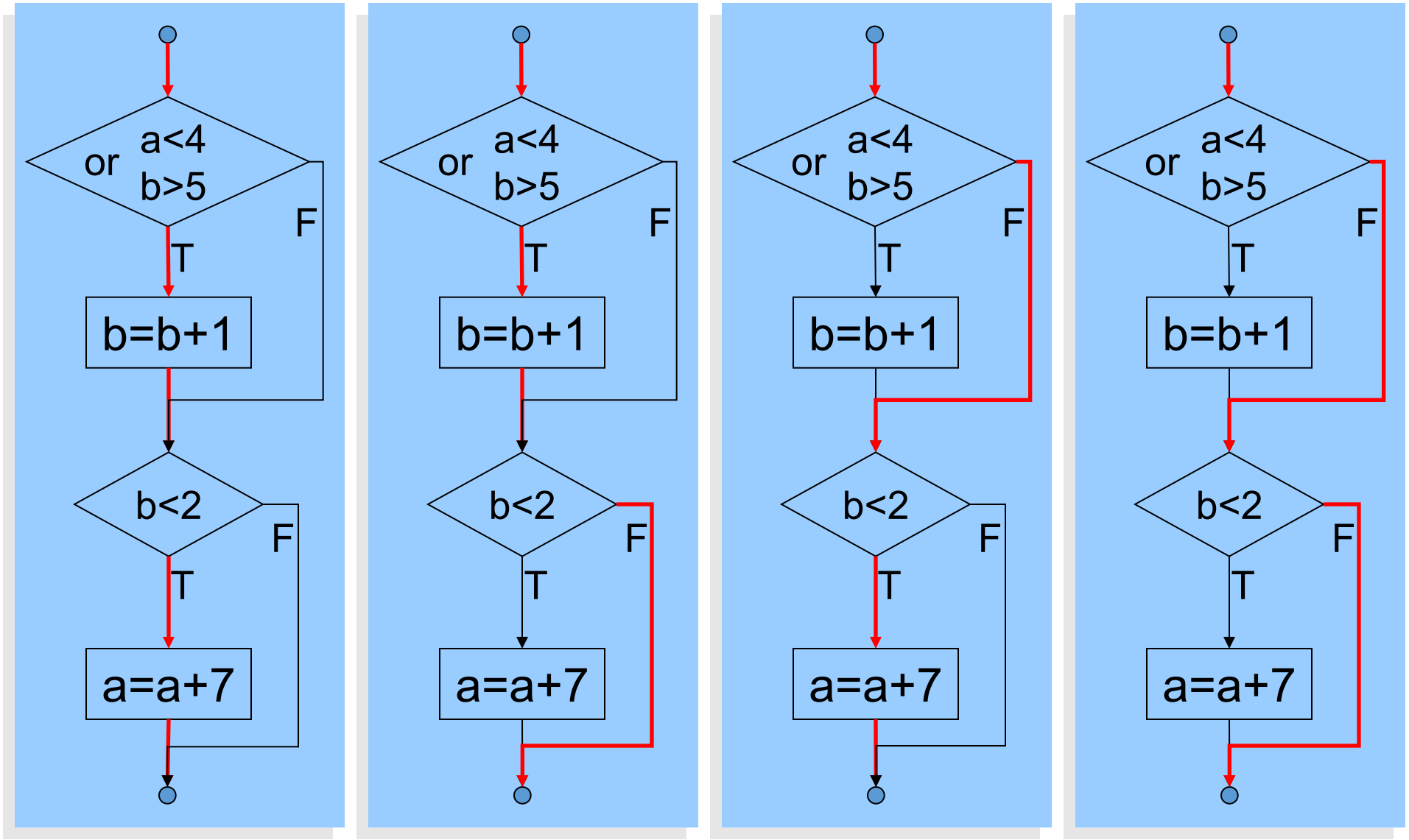
- Aufgabe: Finde minimale Testmenge, die eine der drei Überdeckungsbedingungen erfüllt.

- Beispiel:

```
if ((a<4) || (b>5))
  then b=b+1;
if b<2
  then a=a+7;
```



Pfadüberdeckung

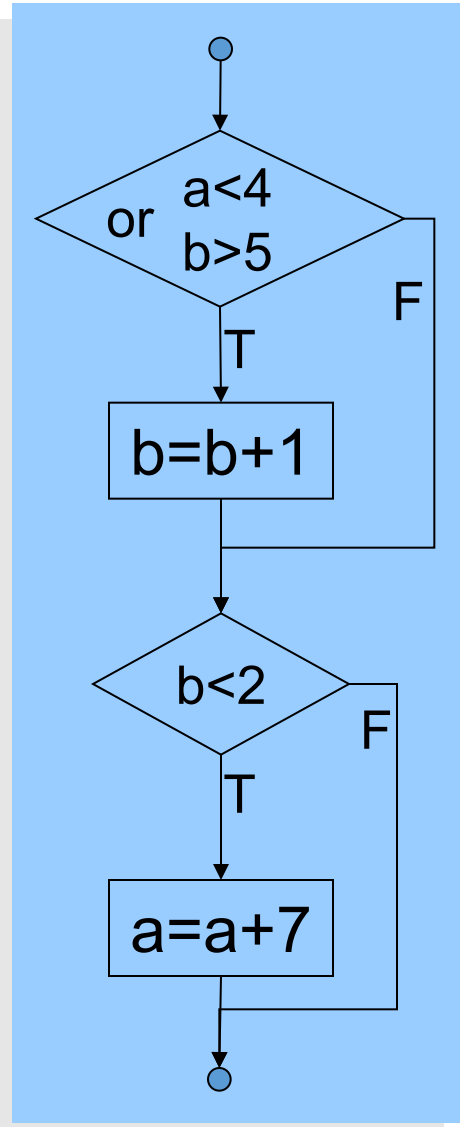


Pfadtesten

- Aufgabe: Finde minimale Testmenge, die eine der drei Überdeckungs-bedingungen erfüllt.
- Beispiel:

```
if ((a<4) || (b>5)) then b=b+1;
if b<2 then a=a+7;
```

- Folgenden Testmengen mit Paaren (a,b) erfüllen z.B. die Bedingungen:
  - Statement coverage: { (0,0) }
  - Branch coverage: { (0,0) , (4,2) }
  - Path coverage: { (0,0) , (0,1) , (4,1) , (4,2) }

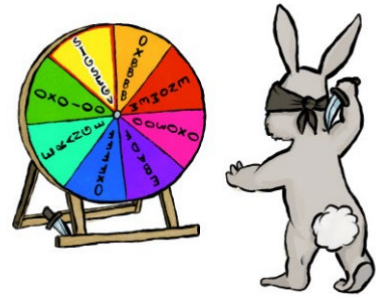


Regressionstesten

- Wenn man häufig an einem Programm Änderungen durchführt, braucht man **automatische Tests**, um Probleme sofort zu erkennen.

- Idealerweise
  - **Regressionstesten** = nach jeder Änderungen werden alle Tests durchlaufen
  - Entwickler **testen** ihren eigenen Code während der Implementierung **ständig**.
  - **White-Box Testing** (Klassen, Methoden, Code sind bekannt)

Fuzzing



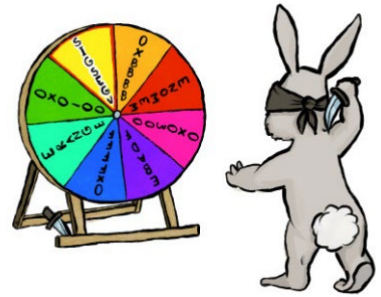
Fuzzing (Fuzzy Testing, Robustness Testing)

Ziel: Finde **Eingaben**, die auf einen Programmfehler (?) hinweisen

Methode: Erzeuge **zufällige** Eingaben, bis das Programm abstürzt.

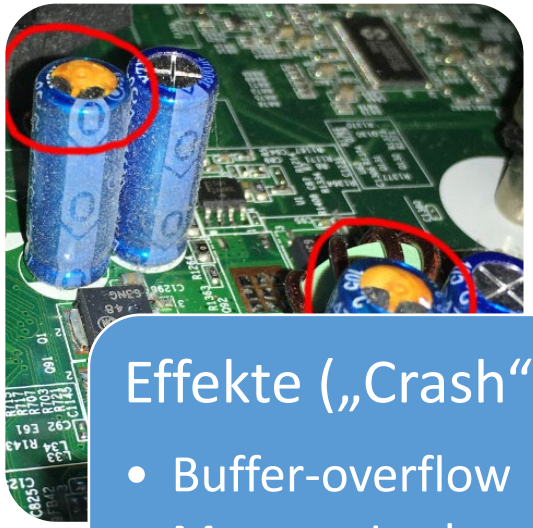
Begriff geht zurück auf Barton Miller, 1989
(Artikel in CACM 1990)
Fuzz = zufällige, unstrukturierte Daten

Fuzzing



Ursachen

- Fehlende/falsche Argumentprüfung
- Fehlerhaftes Type Casting
- Ausführung von „untrusted code“



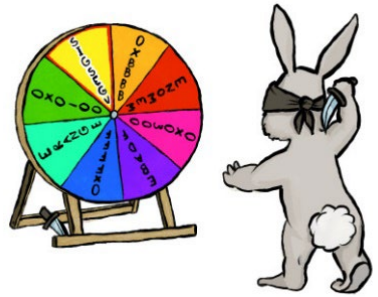
Effekte („Crash“)

- Buffer-overflow
- Memory Leak
- Division durch 0
- Verwendung von freigegebenem Speicher

Negative Auswirkungen auf

- Sicherheit
- Zuverlässigkeit
- Performanz
- Korrektheit

Fuzzing

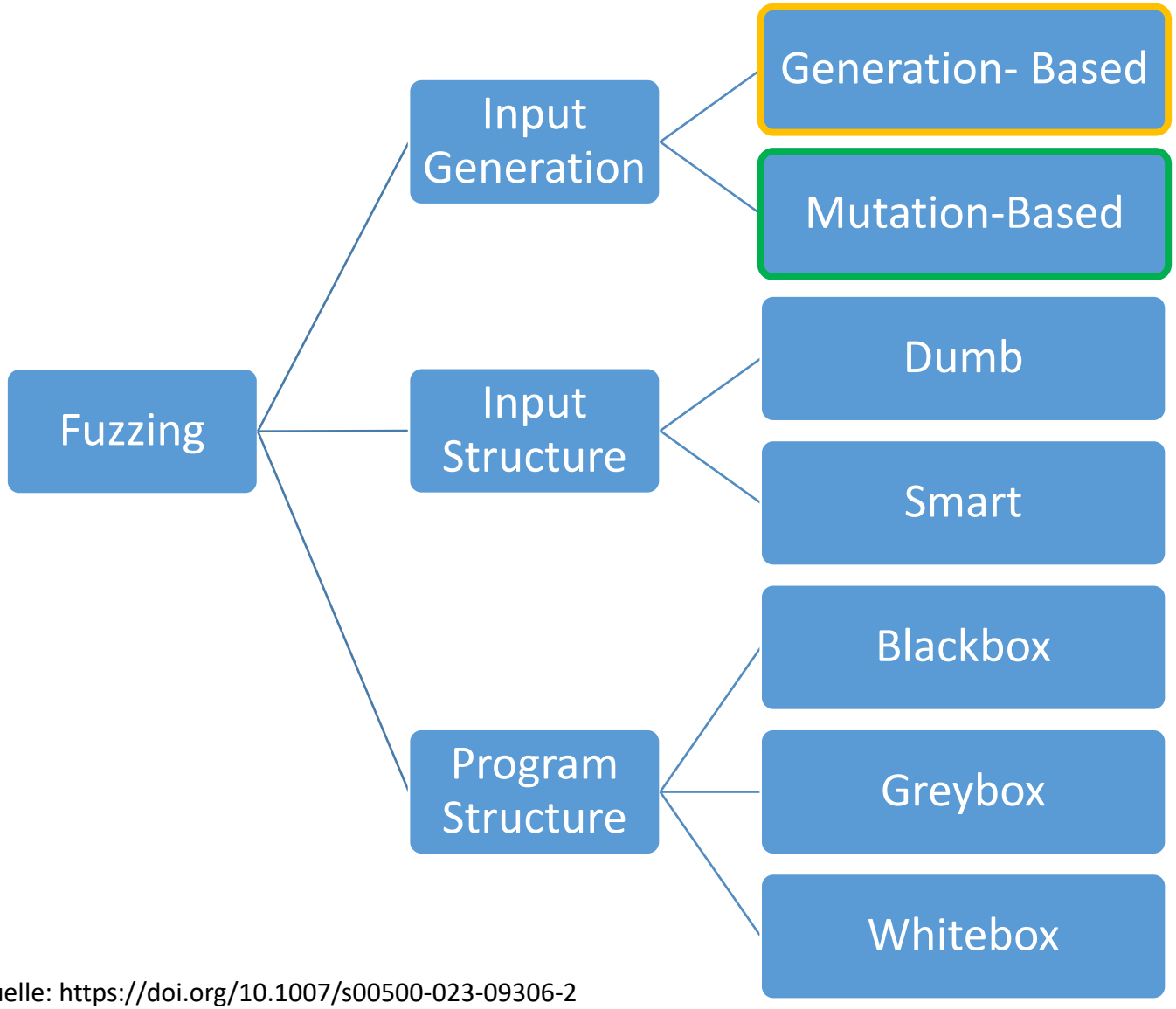


- **Eingabeschnittstelle** wird mit **zufällig** erzeugten Daten ausgeführt, z.B. Funktion mit Zufallsparametern aufgerufen, Zufallsnachrichten geschickt, Tastendrücke simuliert, etc.
- Konkrete Fuzzing-Methoden unterscheiden sich u.a. darin, welches Vorwissen über die zu testende Software benötigt wird (Blackbox bis Whitebox-Testen):
  - **Mutation-based Fuzzing**: Zufällige Änderungen werden in bekannte, gültige Eingaben eingebaut.
  - **Generation-based Fuzzing**: Eingaben werden aufgrund einer Spezifikation des Eingabeformats erzeugt. Zufällige, nicht der Spezifikation entsprechende Änderungen werden eingebaut.

Änderungen sind auf verschiedenen Ebenen möglich:

- Bits (z.B. Bit-Flipping)
- Operatoren (z.B. * statt +)
- Blöcke (z.B. Vertauschen von Blöcken)

Klassifikation



Eingabemodell, z.B. eine Grammatik

Starteingabe (seed) und Änderungsoperationen

Unstrukturierte Daten

Strukturierte Daten

Keine Information über Programmstruktur, weder statisch noch dynamisch

Partielle Information über Programmstruktur, etwa durch Instrumentierung

Kode des Programms (Source, Byte, oder Machine Code)

Quelle: https://doi.org/10.1007/s00500-023-09306-2

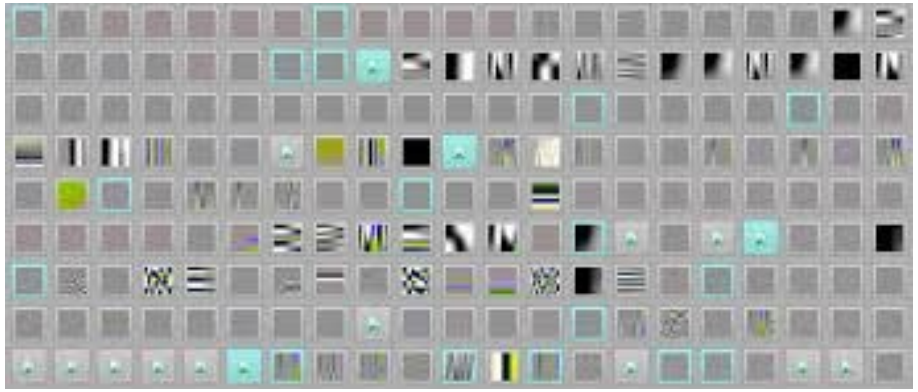
Eingabeformate „entdecken“

- **Coverage-guided Mutation-based Fuzzing**: Sammele Coverage-Informationen während der Ausführung des Programms. Wähle solche Eingaben für die nächste Mutation, die neue Programmzweige ausgeführt haben.
- Beispiel: **afl-fuzz**
  - angewandt auf djpeg mit „hello“ als seed erzeugt JPEGs als Eingabe:

    $ echo 'hello' >in_dir/hello

    $ ./afl-fuzz -i in_dir -o out_dir ./jpeg-9a/djpeg

lcamtuf's old blog: Pulling JPEGs out of thin air



“The first image, hit after about six hours on an 8-core system”

Variante: **PerfFuzz** [Lemieux et al. 2018] entdeckt Eingaben für **Worst-Case Performanz**, indem es Eingaben wiederverwendet, die die Anzahl der Ausführungen eines Programmzweiges maximieren.

Einige interessante Probleme und Lösungen

- Was tun, wenn Eingabedaten Checksummen beinhalten?
  - Eingabe erzeugen (ggf. durch Mutation) und dann Checksumme hinzufügen (oder ändern), oder
  - Programm patchen, so dass die Checksumme nicht geprüft wird.

- Was tun, wenn kein Modell der Eingabedaten vorhanden ist?
  - Verwende Machine Learning, z.B. Neuronale Netze, um ein Model aus einer vorgegebenen Menge von Testdaten zu lernen.

- Was tun, wenn das Initialisieren der Software sehr lange dauert oder eine späte Phase in der Software getestet werden soll?
  - Erstelle einen Snapshot der Software nach der Initialisierung, führe dann das Programm ab diesem Zeitpunkt mit den Testeingaben (durch Ändern im Speicher) weiter aus. Nach jedem Testlauf setze das Programm auf den Snapshot zurück. (→ in-memory fuzzing)

Stammbaum der Fuzzing-Werkzeuge

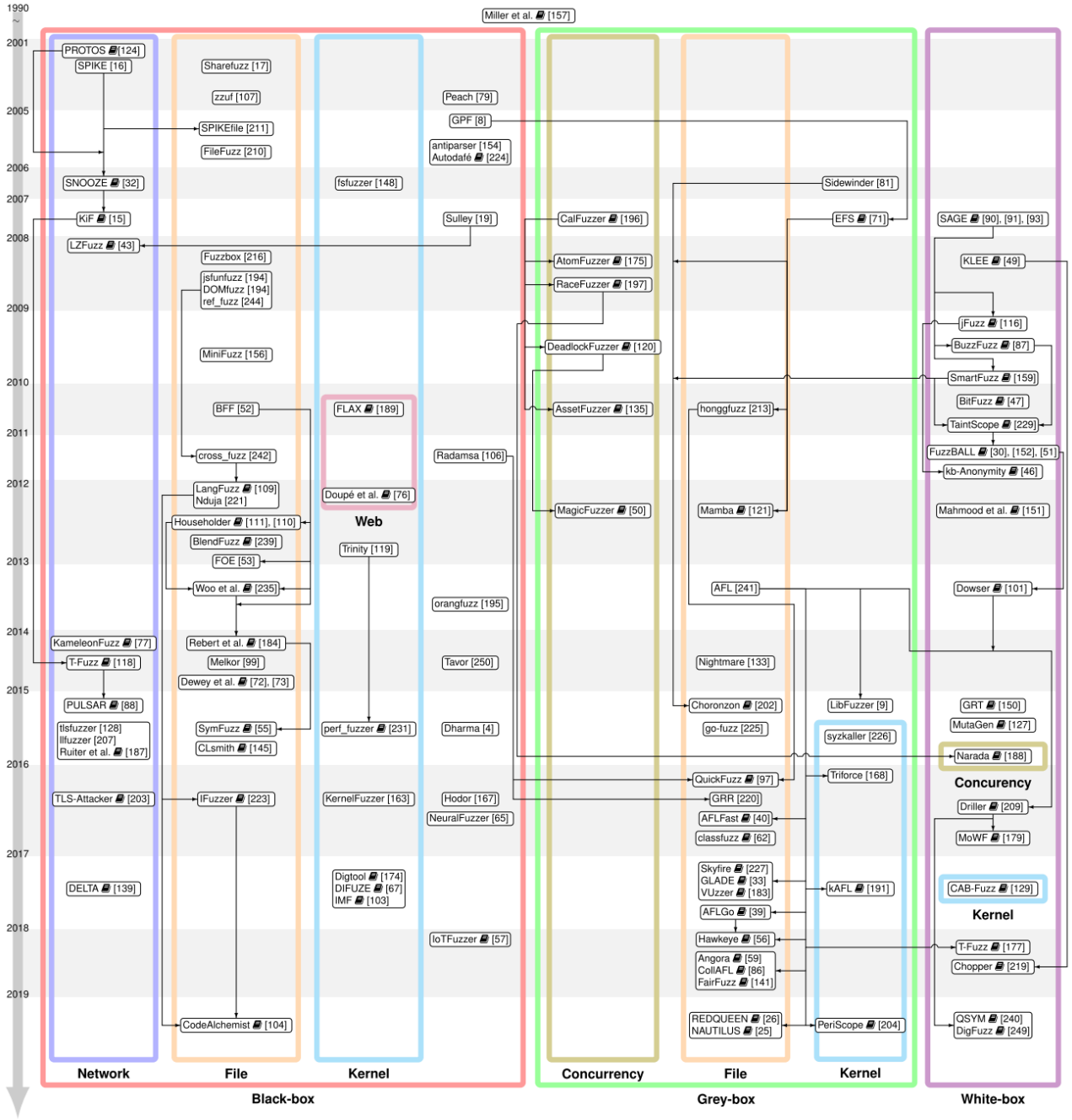


Fig. 1. Genealogy tracing significant fuzzers' lineage back to Miller et al.'s seminal work. Each node in the same row represents a set of fuzz appeared in the same year. A solid arrow from X to Y indicates that Y cites, references, or otherwise uses techniques from X. 🖹 denotes tha paper describing the work was published.

Quelle:

Manes, V. J. M., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., and Woo, M., **The art, science, and engineering of fuzzing: A survey**, IEEE Transactions on Software Engineering, 2021, vol. 47, no. 11, pp. 2312–2331. http://arxiv.org/abs/1812.00140. https://doi.org/10.1109/TSE.2019.2946563

Was tun, wenn man ganz bestimmte Fehlerarten erkennen will?

Code Sanitizer verwenden Instrumentierung, um zur Laufzeit typische Programmierfehler zu erkennen.

Beispiele:

- **Address Sanitizer** (ASan) erkennen Speicherfehler wie etwa Buffer-Overflows, Use-after-free, und Heap- oder Stack-Overflows.
- **Memory Sanitizer** (MSan) erkennen lesende Zugriffe auf nicht initialisierten Speicher.
- **Undefined Behavior Sanitizer** (UBSan) erkennen undefiniertes Verhalten wie etwa Null-Pointer-Dereferenzierung oder Integer-Overflows.
- **Thread Sanitizer** (TSan) erkennen Race Conditions und andere Probleme in Programmen mit mehreren Threads (Multi-Threading).
- **Data Flow Sanitizer** (DFSan) erkennen Datenflussprobleme wie etwa unsichere, noch nicht geprüfte, externe Daten (tainted data) oder Datenlecks (data leak).

Unterschied Regressionstesten und Fuzzing

Regressionstesten führt Programm (in der Regel) mit gültigen Eingaben aus.

- Ziel: Vermeidung von Fehlschlägen bei richtiger Verwendung der Software

Fuzzing führt Programm mit ungültigen (abnormalen) Eingaben aus.

- Ziel:
  - ursprünglich: Vermeidung von Abstürzen bei ungültigen Eingaben (z.B. bei Störungen der Netzwerkverbindung)
  - heutzutage: Vermeidung von durch Angreifer ausnutzbaren Schwachstellen (exploitable errors) in der Software