

# Verlässliche Echtzeitsysteme

## Fehlerinjektion

**Peter Ulbrich**

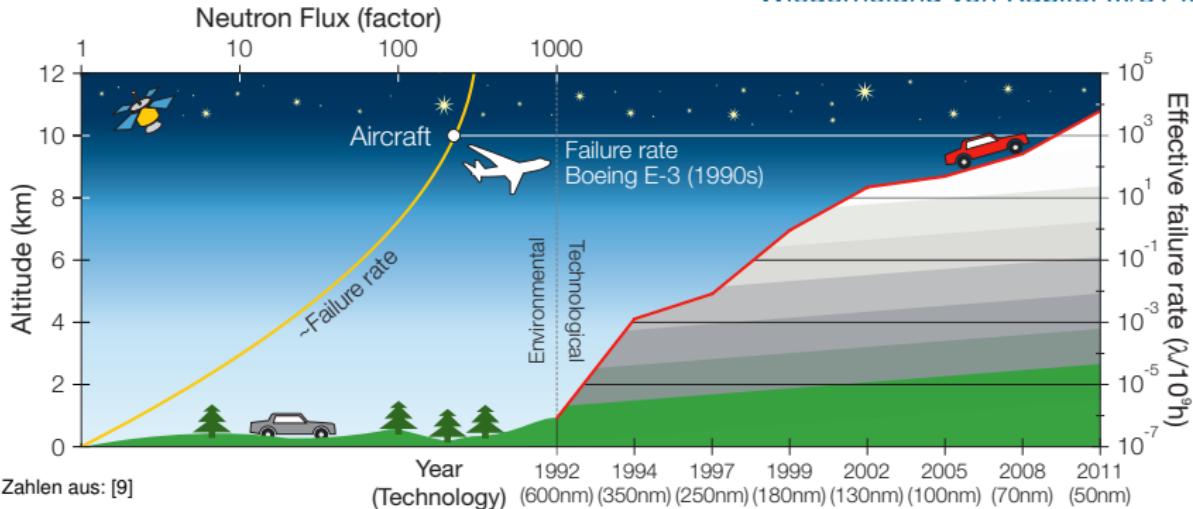
Lehrstuhl für Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität Erlangen-Nürnberg

<https://www4.cs.fau.de>

KW21 2020





- Bitkipper durch **Umladungen in Speicherzellen und Schaltkreisen**
  - Verursacht durch **ionisierende Strahlung**
  - Erzeugung von Elektron-Loch-Paaren im Halbleitermaterial
- Rauschen durch **elektromagnetische Interferenz**
  - Verfälschung von **Kommunikation auf Bussen**
  - z. B. in Automobilen gibt es verschiedene Quellen für Wechselfelder



Extern verursachte Fehler sind die (**absolute**) **Ausnahme!**

- Ausfallrate  $\ll$  Überlebensfunktion (vgl. auch III/12)
- Nachweis der **Wirksamkeit von Fehlertoleranzmechanismen?**



Dedizierte **Testmethoden** sind vonnöten

- Fehlertoleranzmechanismen „verarbeiten Fehler“
  - Test dieser Mechanismen erfordert entsprechende Fehler
- Konkrete Umsetzung der Testverfahren ist aufwendig ...
  - Fehler (auch „häufige“ transiente Fehler) lassen sich **nicht einfach abwarten**
  - Fehler verursachen mitunter **sehr hohe Kosten**



Artifizielle **Fehlerinjektion** als Mittel der Wahl

- Gezielte und reproduzierbare Erzeugung von Fehlern
- Validierung von Fehlertoleranzmechanismen
- Bewertung von Fehlertoleranz
  - Inhärente Robustheit, Fehlerausbreitung, Fehlererkennungslatenz und -rate



# Beispiel: Problemstellung

Fehlerinjektion gibt es nicht nur bei Computern

- Moderne Automobile umfassen eine Vielzahl von Schutzsystemen
  - Air-Bag (Fahrer, Beifahrer, ...), Seitenauflaufschutz, Gurtstraffer, ...  
→ Frage: Wie wirksam sind diese Systeme?
- ⚠ Daten aus dem täglichen Betrieb von Autos mit realen Unfällen ...
  - ... sind **nicht ausreichend vorhanden** (eher seltene Unfälle)
  - ... sind **viel zu teuer** (Verlust von Menschenleben inakzeptabel)
- 👉 Fehlerinjektion durch Crashtests



## 1 Grundlagen

- Aufbau
- Fehlermodell & Fehlerraum
- Aktivierungsmuster

## 2 Fehlerinjektionstechniken

- Hardware-basierte Techniken
- Software-basierte Techniken
- Simulations-basierte Techniken
- Evaluierung CoRed: FAIL\*

## 3 Auswertung und Interpretation

## 4 Zusammenfassung



- FARM-Modell [2] definiert notwendige Voraussetzungen
  - Anmerkungen beziehen sich auf Crashtests (s. Folie 4)

**Fault**  $\leadsto$  Fehlerraum

- Frontal- oder Seitenaufprall, Geschwindigkeit, ...
- Bezieht sich auf eine **realistische Fehlerhypothese**

**Activation**  $\leadsto$  Aktivierungsmuster

- Das beschleunigte Auto fährt auf den Prellbock zu
- Das **Auftreten des Fehler** wird herbeigeführt

**Readout**  $\leadsto$  Messergebnisse

- Deformierung der Fahrgastzelle, ...
- Erhebung der **beobachtbaren Folgen** des Fehlers

**Measure**  $\leadsto$  Bewertung der Messergebnisse

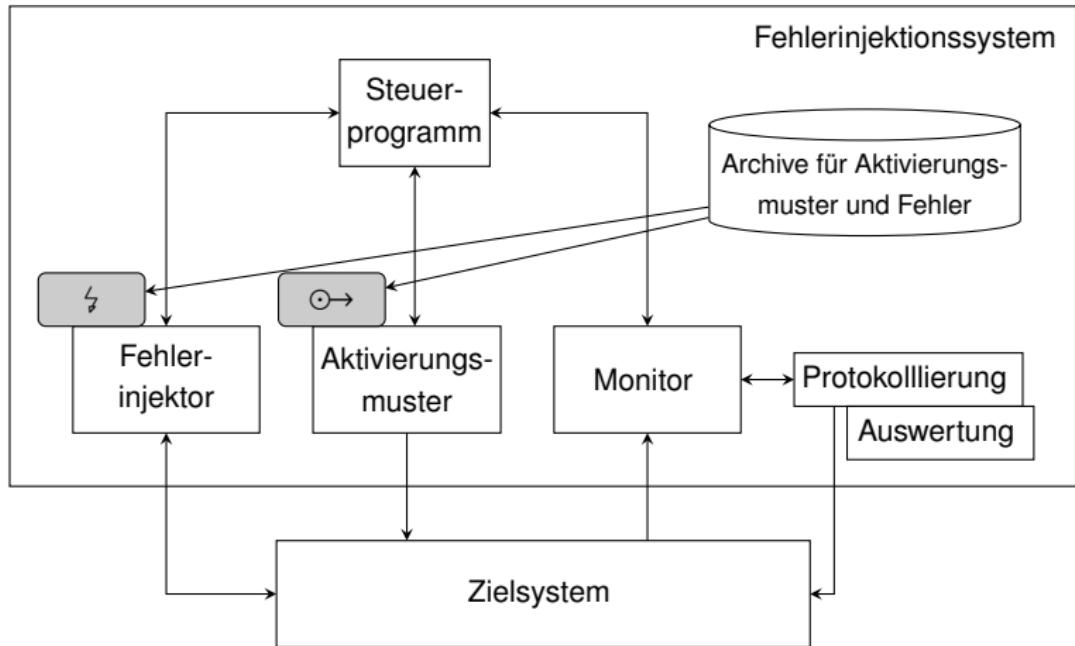
- Insassen würde schwere innere Verletzungen erleiden
- Wie **zuverlässig** ist mein System?



- 1 **Auswahl** des zu injizierenden Fehlers
    - Unterschiedliche Prüfstände für Frontal- bzw. Seitenauftprall
  - 2 **Ausführung** des Aktivierungsmusters
    - Beschleunigung des Fahrzeugs auf die gewünschte Geschwindigkeit
  - 3 **Beobachtung** der Folgen der Fehlersituation
    - Sensoren erfassen Beschleunigungen, Verwindungen, Verformungen, ...
  - 4 **Auswertung** der Messergebnisse
    - Abgleich mit  $\text{\'a-priori}$  Wissen  $\leadsto$  Schluss auf Verletzungen
-  Ein **Werkzeug** übernimmt i. d. R. die Fehlerinjektion
  - Strahlungsquellen, Testschaltungen, Steuerrechner, Debugger, ...

# Genereller Aufbau von Fehlerinjektionswerkzeugen

Quelle Grafik: [4]



- Fehler können auf verschiedenen Ebenen injiziert werden [1]

## Axiomatische Modelle

- Analytische Modelle bilden das Verhalten des Systems ab
- Markov-Ketten, Petri-Netze, Zuverlässigkeitssblockdiagramme

## Empirische Modelle

- Detailliertere Modelle für Systemverhalten und -struktur
- Erfordern i. d. R. simulationsbasierte Ansätze

## Physikalische Modelle

- Reale Implementierung des Systems in Hard- und/oder Software



Wahl der Ebene hat signifikanten Einfluss auf das **Fehlermodell**

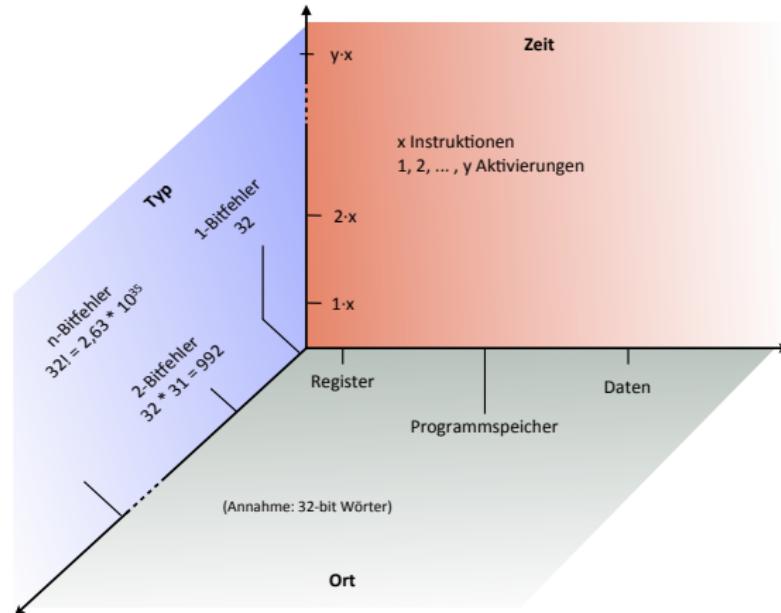
- Insbesondere die Mengen  $F$  und  $A$  hängen von ihnen ab
- Fehlerhypothese: durch Software beobachtbare transiente Fehler
- Konzentration auf empirische/physikalische Modelle



- **Transiente Fehler** haben ihren Ursprung im physikalischen Modell
  - Umwelteinflüsse bewirken **Zustands-/Ladungsveränderungen**
  - Als **Bitkipper** im empirischen Modell beobachtbar
  - Annahme: Bitkipper werden in der Software **sichtbar**
- ☞ Klassisches Fehlermodell → **Einzelbit-Einzelfehler-Annahme**
  - Fehler sind **gleichverteilt** und **unabhängig**
  - Bitfehler treten **im Speicher** auf
  - Zielsystem ist einfache **RISC-Architektur** (**load/store**)
- ⚠ Dies ist eine starke Vereinfachung
  - Fehlermuster können deutlich **komplexer** sein (z.B. durch Pipelines) [3]
  - Fehler treten als **Bündelstörungen** (engl. *error bursts*) auf
  - Anfälligkeit der Hardware für **Gleichtaktfehler** (z.B. CPU-Takt-Kontrolle)
  - In der Praxis wurden 95 % Einzelbit-Fehler beobachtet [5, 10]



## Ein Musterbeispiel für eine kombinatorische Explosion



- Selbst 1-Bitfehler spannen einen **dramatisch großen Fehlerraum** auf!
- In welchem Register möchte man ein Bit kippen lassen?
  - Nach welcher Instruktion soll das Bit gekippt werden?



- Umfassen die Durchführung der zu schützenden Berechnung
  - Einfachster Fall: Vektoren von Eingabeparametern
    - i. d. R. nur für einfache Soft- oder Hardwareimplementierungen anwendbar
    - Präparation **anwendungsspezifischer, passender Eingabedaten**
    - Sensordaten, Netzwerkpakete, ...
  - Kombinationen aus Soft- und Hardware → Kontrollfluss
    - Unterbrechungen werden hier zum Problem
  - Echtzeitsysteme erfordern eine **Umgebungssimulation** (s. VII/34 ff.)
    - Durchführung am „realen Objekt“ häufig nicht möglich/zu gefährlich
    - Eingaben müssen das Verhalten des physikalischen Objekts widerspiegeln

☞ **Referenzlauf** (engl. *golden run*) liefert das gewünschte Verhalten

- Bestimmung des Ergebnisses ohne Fehlerinjektion
  - Aufzeichnung des Ein-/Ausgabeverhaltens des SUT
- Dient dem späteren Abgleich und der Erkennung von SDCs

☞ Anschließend folgt die **eigentliche Fehlerinjektion**

- Einbringen des gewünschten Fehlers



Eine **Kampagne** (engl. *campaign*) beschreibt einen Testfall

- Ein bestimmter Ausführungspfad des Systems
- Hieraus ergeben sich konkrete Möglichkeiten der **Fehleraktivierung**
- Der entstehende Fehlerraum  $\leadsto$  Vielzahl von Einzelexperimenten

■ Fehlerinjektion führt die einzelnen **Experimenten** aus:

- 1 Der Steuerrechner wählt
  - Einen Fehler  $f$  aus dem Fehlerraum  $F$  und
  - Ein Aktivierungsmuster  $a$  aus der Menge der Aktivierungsmuster  $A$
- 2 Anschließend wird die Fehlerinjektion durchgeführt
  - Starten des Aktivierungsmusters  $a$
  - Injizieren des Fehlers  $f$
- 3 Abschließend werden die **Messergebnisse  $r$**  erfasst
  - Jedes Experiment wird durch einen **Tupel  $(f, a, r)$**  beschrieben



**Gesamtheit der Messergebnisse  $R$**   $\leadsto$  Zuverlässigkeitmaße

- Fehlererkennungslatenz- und rate, Erholungszeit, ...



## 1 Grundlagen

- Aufbau
- Fehlermodell & Fehlerraum
- Aktivierungsmuster

## 2 Fehlerinjektionstechniken

- Hardware-basierte Techniken
- Software-basierte Techniken
- Simulations-basierte Techniken
- Evaluierung CoRed: FAIL\*

## 3 Auswertung und Interpretation

## 4 Zusammenfassung



- Injektion von Fehler auf allen Ebenen eines Rechensystems möglich
- ☞ Es existiert eine Vielzahl verschiedener Techniken [11]
  - **Hardware-basierte** Techniken
    - Integriert spezialisierte Hardware in das zu testende System
  - **Software-basierte** Techniken
    - Modifiziert die zu testende Software, um fehlerhaftes Verhalten zu erzeugen
  - **Simulations-basierte** Techniken
    - Simulation des zu testenden Systems, basierend z. B. auf Emulator
  - **Hybride Ansätze**
    - Vereinigt zwei oder mehr der oben genannten Ansätze

- Hardware-basierte Implementierung
  - Enthaltene Testschaltungen injizieren direkt transiente Fehler
  - Basiert auf dem komplett gefertigten Schaltkreis
    - Gefertigter und getester Schaltkreis verhalten sich identisch
    - Das Verfahren ist **nicht-intrusiv** (engl. *non-intrusive*)

Hierbei stehen folgende Möglichkeiten offen:

- **Mit Kontakt** ~ direkte Manipulation elektrischer Signale
  - Anbringungen aktiver Messfühler an einzelnen Prozessорpins
    - Hängen gebliebene Signale (engl. *stuck-at, stuck-open*)
    - Überbrückung mehrerer Signale (engl. *bridging*)
  - Verwendung von **Zwischensockeln** (engl. *socket insertion*)
    - Implementierung beliebiger Funktionen auf den eingehenden Signalen
- **Ohne Kontakt** ~ indirekte Manipulation elektrischer Signale
  - Der Schaltkreis wird physikalischen Phänomenen ausgesetzt
    - Radioaktive Strahlung, elektromagnetische Interferenz, Hitze, ...
    - Rufen (relativ unkontrolliert) transiente/permanente Fehler hervor



## Vorteile

- + Hohe zeitliche Auflösung der Injektion und Beobachtung
  - Ermöglicht akkurate Aussagen zu Fehlererkennungsrate und -latenz
- + Unterstützt nicht-intrusive Fehlerinjektion
  - Betrachtet das komplette System, sowohl Soft- als auch Hardware
- + Durchführung der Experimente ist sehr schnell

## Nachteile

- Eine Beschädigung des getesteten Systems ist möglich
- Hohe Integrationsdichten erschweren die Fehlerinjektion
- Erfordert spezielle Hardware → geringe Portierbarkeit
- Eingeschränkte Kontrollier- und Beobachtbarkeit
  - Nur bestimmte Fehlerarten sind injizierbar
  - Nicht alle Stellen des Schaltkreises sind direkt zugänglich
- Fehlerabdeckung unbekannt (kontaktlose Verfahren)



- Spezielle Softwarekomponenten übernehmen die Fehlerinjektion
- Zur Übersetzungszeit (engl. *compile-time*)
  - Wird das Programmabbild verändert, bevor es geladen wird
    - Für die Fehlerinjektion werden gezielt Software-Defekte eingebracht
    - Die eigentliche Fehlerinjektion ist also die Erzeugung des Abbilds
  - Ausführung des Abbilds aktiviert die eingefügten Defekte
    - Diese simulieren transiente/permanente Hard-/Softwarefehler
- Zur Laufzeit (engl. *run-time*)
  - Erfordert die Aktivierung des Fehlerinjektionsmechanismus
    - z. B. durch Auszeiten, Traps oder Instrumentierung
    - Die Behandlung der Ereignisse führt die Fehlerinjektion durch
  - Instrumentierung bereitet die Fehlerinjektion vor
    - Bringt gezielt Instruktionen in das Programmabbild ein
    - Diese aktivieren dann die Fehlerinjektion



## Vorteile

- + Sehr flexible Injektion von Fehlern möglich
  - Fehler in Registern, Speicher, bei der Kommunikation, im Zeitbereich
  - Injektion ist in Simulationen und realen Systemen möglich
- + Durchführung der Experimente ist sehr schnell
- + Keine Spezialhardware erforderlich

## Nachteile

- Eingeschränkte Auswahl von Injektionsstellen
  - i. d. R. auf der Ebene von Assemblerinstruktionen
- Eingeschränkte Kontrollier- und Beobachtbarkeit
- Erfordert eine Modifikation der getesteten Software
  - Letztendlich wird ein anderes Programmabbild verwendet
  - Injektionsverfahren ist intrusiv  $\sim$  es beeinflusst das Verhalten



- Ein Modell des zu testenden Systems wird im Simulator ausgeführt
  - Das Modell umfasst z. B. Prozessor, Peripherie, Kommunikation, ...
-  Fehlerinjektion basiert auf:
- **Modifikation des Systemmodells** → vgl. software-basierte Lösung
  - **Saboteure**: „boshaft“ in das Modell eingebrachte Komponenten
    - Aktivierung → Ausführung der Fehlerinjektion (z. B. Signalstörung)
    - Ansonsten verhalten sie sich unauffällig
  - **Mutanten**: „boshaft“ veränderte Komponenten des Modells
    - Veränderte, existierende Komponenten injizieren Fehler
- **Modifikation der Simulation** → vgl. hardware-basierte Lösung
  - Erfordert keine Veränderung des Modells sondern des Simulators
  - Injektion von Fehlern an beliebigen Stellen/Zeitpunkten
    - Modifikation von Zuständen oder Signalen





Vereinfachung: In Software sichtbare Fehler  $\leadsto$  Bitkipper

- Ihr Zustandekommen ist uninteressant
- Verzicht auf eine physikalische Fehlerinjektion
- $\rightarrow$  Konsequenzen hinsichtlich der Validität des Fehlermodells (vgl. Folie 10)



Fehlerinjektion bringen Bitkipper direkt in den Ausführungsstrom ein

- Keine direkte, physikalische Manipulation notwendig
- $\rightarrow$  **Simulation von Fehlern auf Registertransferebene**
  - Nicht zu verwechseln mit **Fehlersimulation** (engl. *fault simulation*)
  - $\rightarrow$  Hier wird ein Schaltkreis in Anwesenheit von Fehlern simuliert

### ■ Software Implemented Fault Injection (SWIFI)

- Falls eine Softwareimplementierung die Verfälschung durchführt
- Alternativ: Verwendung spezialisierter Debug-Schnittstellen (z. B. JTAG)
- $\rightarrow$  **Scan-Chain Implemented Fault Injection (SCIFI)**



## Vorteile

- + Größtmögliche Flexibilität: Abstraktionsebene/Fehlerhypothese
  - Auf elektrischer, logischer, funktionaler und architektureller Ebene
  - Injektion zeitlicher, transienter und permanenter Fehler
- + Nicht-intrusive Injektion möglich (unverändertes Programmabbild)
- + Erfordert keine Spezialhardware
- + Maximaler Grad an Kontrollier- und Beobachtbarkeit

## Nachteile

- Hoher Zeitaufwand
  - Erfordert die Entwicklung von (detaillierten) Systemmodellen
  - Die Simulationsgeschwindigkeit ist häufig niedrig
- Hängt von der Akkurateit des Systemmodells ab
  - Kein 100%-iges Abbild der Realität ( $\sim$  keine „Echtzeitsimulation“)



- Validierung von CoRed (s. V/36 ff.) durch Fehlerinjektion
  - Erster Ansatz: Debug-Schnittstelle des TriCore (OCDS)
  - Steuerrechner: Debugger Trace 32 von Lauterbach
    - Skriptgesteuerte Ausführung von Ausführung, Injektion und Protokollierung
- ☞ Durchführung von Fehlerinjektion ist eine **große Herausforderung**
  - Man kämpft mit einem **riesigen Fehlerraum**
  - Experimentbeschreibung ist **nicht standardisiert/wiederverwendbar**
  - **Hoher zeitlicher Aufwand:** 1s/Experiment \* 400.000 Experimente  $\sim$  110 h
- ⚠ Prinzipiell existiert eine Vielzahl von Werkzeugen, **aber:**
  - Diese sind **hochgradig proprietär**
    - Eigene Fehlermodelle, Experimentbeschreibung, Ergebnisauswertung
  - An **bestimmte Zielplattformen** gebunden
    - Erweitern häufig (veraltete Versionen) existierender Emulatoren
  - Eine einfache Verwendung „out-of-the-box“ unmöglich

## ■ FAIL\* $\leadsto$ Fault Injection Leveraged

- Vorrangiges Entwurfsziel: Flexibilität bei der Fehlerinjektion  
→ Effizient durch intelligente Reduktion des Fehlerraums

## ☞ Verwendung existierender virtueller Plattformen

- Aktuelle, gewartete Softwarebasis
- Schneller Wortsrechner  $\leadsto$  schnelle Durchführung von Experimenten
- Voller Zugriff auf und volle Kontrolle über die Plattform
- Verschiedene Zielplattformen (Bochs, Gem5, OpenOCD, ...)

## ☞ Schaffung einer abstrakten Schnittstelle zu diesen Plattformen

- Wiederverwendbare Beschreibung von Experimenten

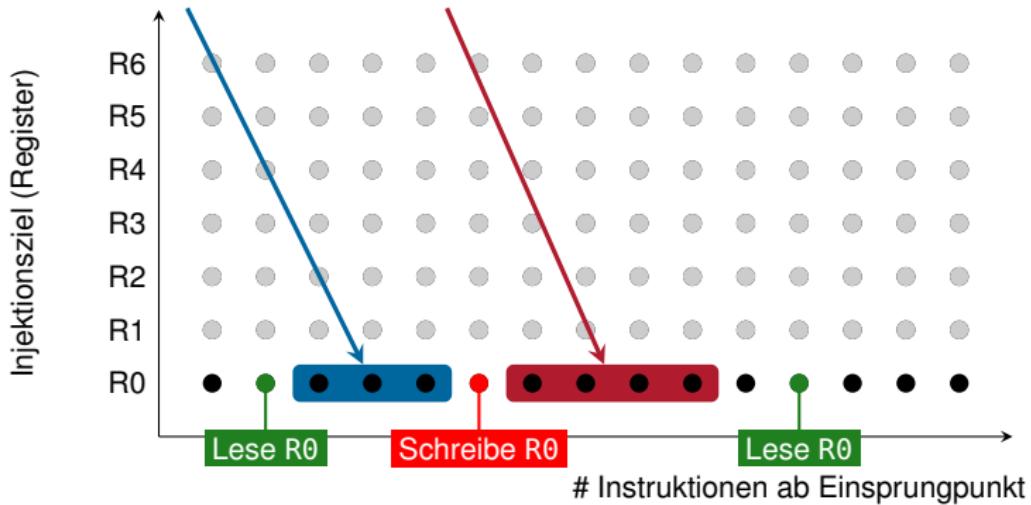


Mehr zur Anwendung von FAIL\* in den Übungen!



# Reduktion der Kampagnendauer

- Reduktion des Fehlerraums durch „fault-space pruning“
  - Register R1 … R6 sind uninteressant
  - Eliminiere **unwirksame** und **idempotente** Injektionen



- Einzelne Experimente sind **unabhängig** voneinander  
→ Sie lassen sich **hervorragend** parallelisieren
  - Auf mehreren Kernen, Prozessoren, Rechnern, ... in der Cloud



## 1 Grundlagen

- Aufbau
- Fehlermodell & Fehlerraum
- Aktivierungsmuster

## 2 Fehlerinjektionstechniken

- Hardware-basierte Techniken
- Software-basierte Techniken
- Simulations-basierte Techniken
- Evaluierung CoRed: FAIL\*

## 3 Auswertung und Interpretation

## 4 Zusammenfassung



- Mächtigkeit des Zielsystems bestimmt erfassbare Messergebnisse

 Hilfreich sind folgende Informationen

- Fehlerparameter
  - Wann und wo wurde der Fehler injiziert? Welcher Typ wurde injiziert?
- Systemkontext
  - Werte der Register, Auszug eines Speicherbereichs
  - Was hat der Fehler verändert? Wie hat er sich fortgepflanzt?
- Rückgabewerte, Rechenergebnisse
  - Hat die Fehlerinjektion die Berechnung beeinflusst?
- Ausführungszeit
  - Wie lange dauert es bis der Fehler aktiviert, entdeckt oder maskiert wurde?
- Fehlererkennungsmechanismen
  - Welcher Fehlerdetektor schlug an?

 Hieraus werden Maße zur Beurteilung der Fehlertoleranz bestimmt

- Rate der Fehlererkennung und Maskierung, Latenz, Erholungszeit



### ■ Wie viel sicherer ist mein System durch Fehlertoleranz geworden?

- Typische Antwort: x % besser
- Basierend auf den beobachteten Fehlerwahrscheinlichkeit:

$$r = \frac{P(sdc)_{\text{Ungeschützt}}}{P(sdc)_{\text{Fehlertolerant}}}$$



### Gültigkeit nur bei Äquivalenz der Fehlerräume

- Beziehungsweise einer statistisch signifikanten Überapproximation
- Klassisches Vorgehen bei hardware-basierter Fehlerinjektion
  - Strahlungsquelle bleibt beispielsweise immer gleich
  - Hinreichend große Montecarlo-Experimente



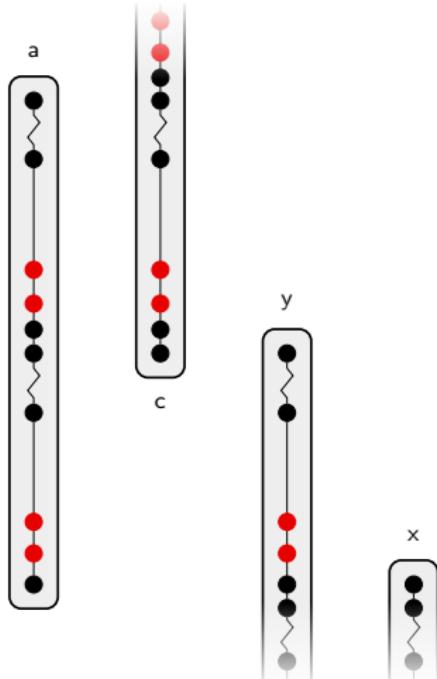
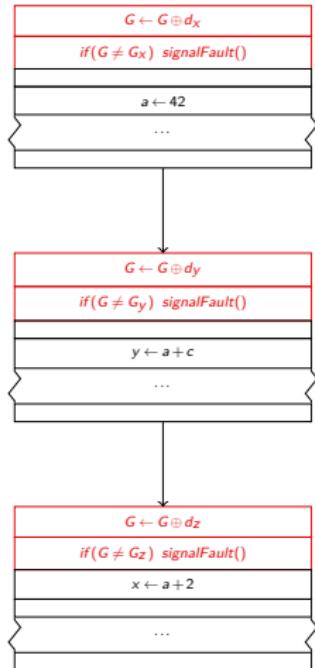
### Softwarebasierte Fehlerinjektion erfordert ein Umdenken [6]

- Fehlerräume unterscheiden sich konzeptbedingt
- Systematische Fehlerinjektion (wie in FAIL\*) bietet keine Überapproximation
- Reduktion des Fehlerraums muss berücksichtigt werden
- **Absolute Fehlerzahlen** anstatt Fehlerraten!



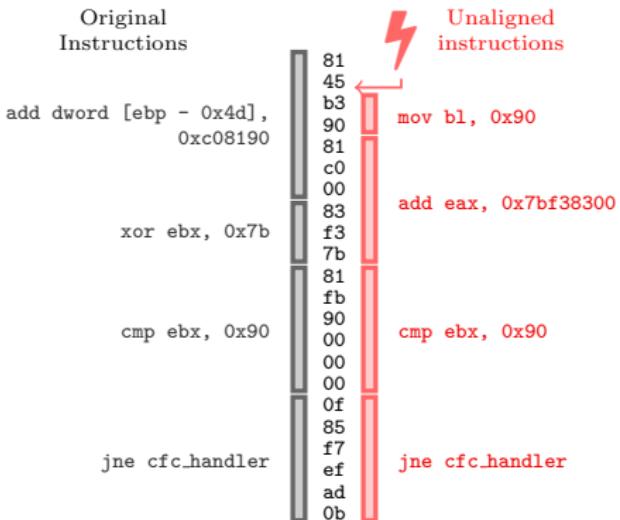
# Beispiel: Lebensdauer von Daten

Äquivalenzklassen in FAIL\*, Grafik: [8]



# Beispiel: Sprung in Instruktionsstrom

## Probleme bei der Interpretation der Fehlerursache



- Architekturen mit variabler Instruktionsbreite
- ☞ Schwierig illegale Instruktionen zu detektieren

## 1 Grundlagen

- Aufbau
- Fehlermodell & Fehlerraum
- Aktivierungsmuster

## 2 Fehlerinjektionstechniken

- Hardware-basierte Techniken
- Software-basierte Techniken
- Simulations-basierte Techniken
- Evaluierung CoRed: FAIL\*

## 3 Auswertung und Interpretation

## 4 Zusammenfassung



## FARM-Modell Für Fehlerinjektion

- Fault, Activation, Readout, Measure
- Auswahl, Ausführung, Beobachtung, Auswertung
- Abstraktionsebenen – axiomatisch, empirisch, physikalisch
- genereller Aufbau und Ablauf von Fehlerinjektionswerkzeugen

## Fehlerinjektionstechniken → grundlegende Kategorisierung

- {hardware, software, simulations}-basiert

## FAIL\* → Grundlage für generische Fehlerinjektion?

- Basierend auf virtuellen Zielsystemen
  - flexible Plattform für Fehlerinjektion
  - schnelle Experimentdurchführung durch Parallelisierung
-  Absolute Fehlerauswertung



- [1] Arlat, J. ; Crouzet, Y. ; Laprie, J.-C. :  
Fault injection for dependability validation of fault-tolerant computing systems.  
In: *Proceedings of the 19th International Symposium on Fault-Tolerant Computing (FTCS-19)*, 1989, S. 348–355
- [2] Arlat, J. ; Aguera, M. ; Amat, L. ; Crouzet, Y. ; Fabre, J.-C. ; Laprie, J.-C. ; Martins, E. ; Powell, D. :  
Fault Injection for Dependability Validation: A Methodology and Some Applications.  
In: *IEEE Transactions on Software Engineering* 16 (1990), Febr., Nr. 2, S. 166–182.  
<http://dx.doi.org/10.1109/32.44380>. –  
DOI 10.1109/32.44380. –  
ISSN 0098–5589
- [3] Cho, H. ; Mirkhani, S. ; Cher, C.-Y. ; Abraham, J. ; Mitra, S. :  
Quantitative evaluation of soft error injection techniques for robust system design.  
In: *Proceedings of the 50th annual Design Automation Conference*, 2013. –  
ISSN 0738–100X, S. 1–10
- [4] Hsueh, M.-C. ; Tsai, T. K. ; Iyer, R. K.:  
Fault Injection Techniques and Tools.  
In: *IEEE Computer* 30 (1997), Apr., Nr. 4, S. 75–82.  
<http://dx.doi.org/10.1109/2.585157>. –  
DOI 10.1109/2.585157. –  
ISSN 0018–9162

- [5] Maiz, J. ; Hareland, S. ; Zhang, K. ; Armstrong, P. :  
Characterization of multi-bit soft error events in advanced SRAMs.  
In: *Proceedings of the IEEE International Electron Devices Meeting (IEDM '03)*.  
New York, NY, USA : IEEE Press, 2003, S. 21.4.1–21.4.4
- [6] Schirmeier, H. ; Borchert, C. ; Spinczyk, O. :  
Avoiding Pitfalls in Fault-Injection Based Comparison of Program Susceptibility to Soft Errors.  
In: *Proceedings of the 45th International Conference on Dependable Systems and Networks (DSN '15)*.  
Washington, DC, USA : IEEE Computer Society Press, Jun. 2015
- [7] Schirmeier, H. ; Hoffmann, M. ; Kapitza, R. ; Lohmann, D. ; Spinczyk, O. :  
FAIL\*: Towards a Versatile Fault-Injection Experiment Framework.  
In: Mühl, G. (Hrsg.) ; Richling, J. (Hrsg.) ; Herkersdorf, A. (Hrsg.): *25th International Conference on Architecture of Computing Systems (ARCS '12), Workshop Proceedings* Bd. 200, Gesellschaft für Informatik, März 2012 (Lecture Notes in Informatics). – ISBN 978-3-88579-294-9, S. 201–210
- [8] Schuster, S. :  
*Control-Flow Monitoring for KESO Applications.*  
Bachelor thesis, University of Erlangen-Nuremberg, Germany, Mai 2015

- [9] Shivakumar, P. ; Kistler, M. ; Keckler, S. W. ; Burger, D. ; Alvisi, L. :  
Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic.  
In: *Proceedings of the 32nd International Conference on Dependable Systems and Networks (DSN '02)*.  
Washington, DC, USA : IEEE Computer Society Press, Jun. 2002, S. 389–398
- [10] Sridharan, V. ; Stearley, J. ; DeBardeleben, N. ; Blanchard, S. ; Gurumurthi, S. :  
Feng Shui of Supercomputer Memory: Positional Effects in DRAM and SRAM Faults.  
In: *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*.  
New York, NY, USA : ACM Press, 2013 (SC '13). –  
ISBN 978-1-4503-2378-9, S. 22:1–22:11
- [11] Ziade, H. ; Ayoubi, R. A. ; Velazco, R. :  
A Survey on Fault Injection Techniques.  
In: *The International Arab Journal of Information Technology* 1 (2004), Nr. 2, S. 171–186