

# Verlässliche Echtzeitsysteme

## Fehlerinjektion

Peter Ulbrich

Friedrich-Alexander-Universität Erlangen-Nürnberg  
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)  
www4.informatik.uni-erlangen.de

19. Mai 2014



## Fragestellungen

- Wie sichert man die **Wirksamkeit von Fehlertoleranzmechanismen**?
  - Hilft uns die vielfältige Redundanz weiter?
    - Information, Daten, Berechnungen und Hardware
- ☞ **Testmethoden** für Fehlertoleranzmechanismen müssen her!
  - ↪ Fehlertoleranzmechanismen „verarbeiten Fehler“
    - man braucht also Fehler, um diese Mechanismen zu testen
  - ↪ Fehler sind in ihrer Handhabung aber nicht so einfach ...
    - sie treten **vergleichsweise selten** auf (auch „häufige“ transiente Fehler)
    - Fehler **verursachen sehr hohe Kosten**
- ☞ **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, ...), Seitenaufprallschutz, 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



## Gliederung

- 1 Überblick
- 2 FARM
- 3 Fehlerinjektionstechniken
- 4 FAIL\*
- 5 Zusammenfassung



## Fehlerinjektion – Was braucht man da alles?

- erstmals festgelegt im **FARM-Modell** [2]
  - die beispielhaften Anmerkungen beziehen sich auf Crashtests (s. Folie VI/3)

**Fault** ∼ Fehlerraum

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

**Activation** ∼ Aktivierungsmuster

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

**Readout** ∼ Messergebnisse

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

**Measure** ∼ Bewertung der Messergebnisse

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



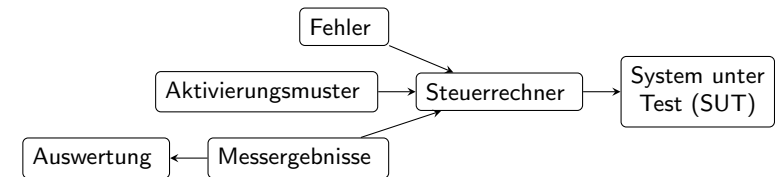
## Fehlerinjektion – Was braucht man da alles? (Forts.)

Irgendwie muss sich das auch noch bewegen ...

- **Auswahl** des zu injizierenden Fehlers
  - unterschiedliche Prüfstände für Frontal- bzw. Seitenaufprall
- **Ausführung** des Aktivierungsmusters
  - Beschleunigung des Fahrzeugs auf die gewünschte Geschwindigkeit
- **Beobachtung** der Folgen der Fehlersituation
  - Sensoren erfassen Beschleunigungen, Verwindungen, Verformungen, ...
- **Auswertung** der Messergebnisse
  - Abgleich mit á-priori Wissen ∼ Schluss auf Verletzungen

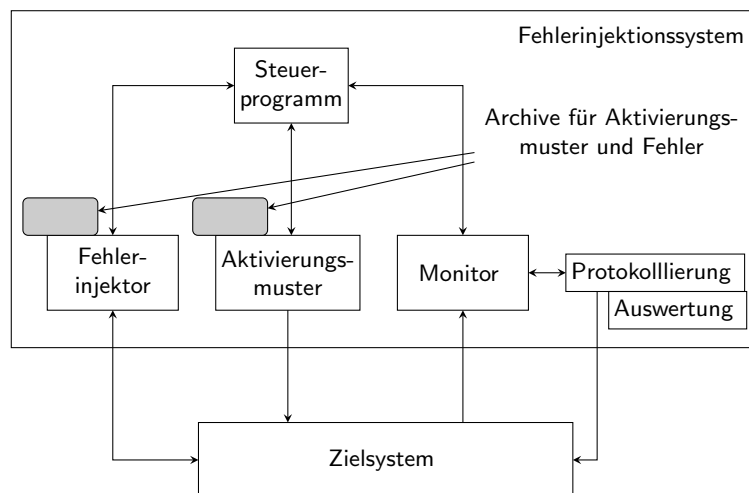


ein **Steuerrechner** übernimmt i. d. R. die Fehlerinjektion in Rechner



## Genereller Aufbau von Fehlerinjektionswerkzeugen

Quelle Grafik: [3]



## Ablauf einer Fehlerinjektion

- Fehlerinjektion besteht aus **Experimenten**:
  - 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
- eine **Kampagne** (engl. *campaign*) beinhaltet mehrerer Experimente
  - der Fehlerraum ist meist sehr groß ∼ eine Vielzahl von Fehlern
  - es gibt mannigfaltig Möglichkeiten für ihre Aktivierung
- Gesamtheit der Messergebnisse  $R$  ∼ **Zuverlässigkeitsmaße**
  - Fehlererkennungslatenz- und rate, Erholungszeit, ...



## Fehlerinjektion – Abstraktionsebenen

- Fehler können auf verschiedenen Ebenen injiziert werden
- axiomatische Modelle
  - analytische Modelle bilden das Verhalten des Systems ab
  - Markov-Ketten, Petri-Netze, Zuverlässigkeitsblockdiagramme

### 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

☞ diese Ebenen haben signifikanten Einfluss auf die Fehlerinjektion

- insbesondere die Mengen  $F$  und  $A$  hängen von ihnen ab

☞ im Folgenden  $\rightsquigarrow$  Konzentration auf empirische/physikalische Modelle

- Fehlerhypothese: durch Software beobachtbare transiente Fehler



## Fehlerraum

- **transiente Fehler** haben ihren Ursprung im physikalischen Modell
  - Umwelteinflüsse bewirken **Zustands-/Ladungsveränderungen**
    - Spannungsschwankungen bei der Datenübertragung
    - Umladen von der **Kondensatoren in Speicherzellen** (EMV, Strahlung)
    - Veränderung der Leitfähigkeit durch **Hitzeinwirkung**
    - **fehlerhafte Berechnungen** in arithmetischen Einheiten
    - ...
  - Annahme: diese Veränderungen werden in der Software sichtbar



als „**Bitkipper**“ im empirischen Modell beobachtbar

- in Registern, Daten, Programmspeicher, ...

- wir interessieren vor allem für diese „Bitkipper“

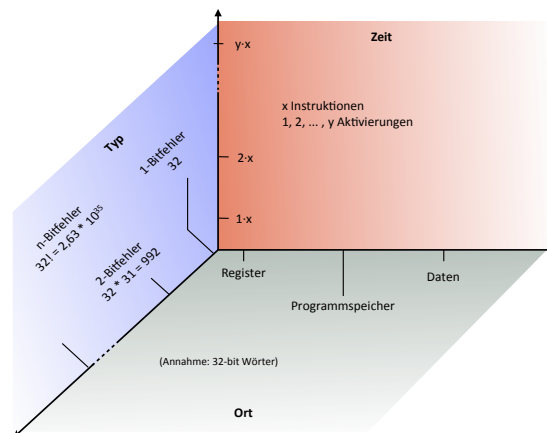
$\rightsquigarrow$  hierfür gibt es immer noch sehr viele Möglichkeiten

- woher sie kommen, ist nicht so sehr von Belang



## Fehlerraum (Forts.)

Ein Musterbeispiel für eine kombinatorische Explosion



- selbst die Beschränkung auf 1-Bitfehler lässt vieles offen
  - In welchem Register möchte man ein Bit kippen lassen?
  - Nach welcher Instruktion soll das Bit gekippt werden?



## Aktivierungsmuster

- Durchführung der zu schützenden Berechnung
  - einfachster Fall: Vektoren von Eingabeparametern
    - i. d. R. nur für einfache Soft- oder Hardwareimplementierungen anwendbar
    - Kombinationen aus Soft- und Hardware erfordern meist auch Kontrollfluss
  - Präparation **anwendungsspezifischer, passender Eingabedaten**
    - Sensordaten, Netzwerkpakete, Unterbrechungen, ...
  - Echtzeitsysteme erfordern eine **Umgebungssimulation** (s. Folie ?? ff.)
    - Durchführung am „realen Objekt“ häufig nicht möglich/zu gefährlich
    - Eingaben müssen das Verhalten des physikalischen Objekts widerspiegeln
- ein **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**

$\rightsquigarrow$  Einbringen des gewünschten Fehlers



## Messergebnisse und ihre Bewertung

- 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?
- ☞ hierraus werden **Maße zur Beurteilung der Fehlertoleranz** bestimmt
  - Rate der Fehlererkennung und Maskierung, Latenz, Erholungszeit



## Physikalisches vs. empirisches Modell

Besteht ein Zusammenhang zwischen diesen beiden Welten?

- uns interessieren eigentlich nur „Bitkipper“
  - ihr Zustandekommen ist hingegen eher uninteressant
  - Verzicht auf eine indirekte physikalische Fehlerinjektion
    - diese ist **sehr aufwendig** und **schlecht kontrollierbar**
    - benötigt beispielsweise EMV-Messkammern oder Ionen-Kanonen
    - welches Bit in welchem Register wurde denn nun beeinflusst ...
- ☞ im Zuge der Fehlerinjektion bringen wir direkt „Bitkipper“ ein
  - statt sie durch physikalische Manipulation zu erzeugen
  - ~> **Simulation von Fehlern auf Registertransferebene**
    - nicht zu verwechseln mit **Fehlersimulation** (engl. *fault simulation*)
    - ~> hier wird ein Schaltkreis in Anwesenheit von Fehlern simuliert
  - man spricht von: **Software Implemented Fault Injection** (SWIFI)
    - falls eine Softwareimplementierung die Verfälschung durchführt
    - alternativ: Verwendung spezialisierter Debug-Schnittstellen (z. B. JTAG)
    - ~> **Scan-Chain Implemented Fault Injection** (SCIFI)



## Gliederung

- 1 Überblick
- 2 FARM
- 3 Fehlerinjektionstechniken
- 4 FAIL\*
- 5 Zusammenfassung



## Fehlerinjektionstechniken

- Injektion von Fehler auf allen Ebenen eines Rechensystems möglich
- ☞ es existiert eine Vielzahl verschiedener Techniken [4]
  - **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 VHDL
  - **emulations-basierte** Techniken
    - beschleunigt Simulation durch Synthetisierung des Modells
  - **hybride Ansätze**
    - vereinigt zwei oder mehr der oben genannten Ansätze



## Hardware-basierte Fehlerinjektion

grundsätzlich stehen folgende Möglichkeiten offen:

mit **Kontakt**  $\leadsto$  direkte Manipulation elektrischer Signale

- Anbringungen aktiver Messfühler an einzelnen Prozessorpins
  - 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**  $\leadsto$  indirekte Manipulation elektrischer Signale

- der Schaltkreis wird physikalischen Phänomenen ausgesetzt
  - radioaktive Strahlung, elektromagnetische Interferenz, Hitze, . . .
  - rufen (relativ unkontrolliert) transiente/permanente Fehler hervor

hardware-basierte Implementierung  $\leadsto$  „Simulation“

- enthaltene Testschaltungen injizieren direkt transiente Fehler
  - basiert auf dem komplett gefertigten Schaltkreis
    - gefertigter und getesteter Schaltkreis verhalten sich identisch
- $\leadsto$  das Verfahren ist **nicht-intrusiv** (engl. *non-intrusive*)



## Hardware-basierte Fehlerinjektion (Forts.)

### Vorteile

- + hohe zeitliche Auflösung der Injektion und Beobachtung
  - $\leadsto$  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  $\leadsto$  geringe Portierbarkeit
- eingeschränkte Kontrollier- und Beobachtbarkeit
  - nur bestimmte Fehlertypen sind injizierbar
  - nicht alle Stellen des Schaltkreises sind direkt zugänglich



## Software-basierte Fehlerinjektion

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
- $\leadsto$  diese aktivieren dann die Fehlerinjektion



## Software-basierte Fehlerinjektion (Forts.)

### 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  $\leadsto$  es beeinflusst das Verhalten



## Simulations-basierte Fehlerinjektion

- ein Modell des zu testenden Systems wird im Simulator ausgeführt
  - das Modell umfasst z. B. Prozessor, Peripherie, Kommunikation, ...

☞ Fehlerinjektion basiert auf

**Modifikation des Systemmodells**  $\leadsto$  vgl. software-basierte Lösung

- **Saboteur:** „boshafte“ in das Modell eingebrachte Komponenten
  - Aktivierung  $\leadsto$  Ausführung der Fehlerinjektion (z. B. Signalstörung)
    - dafür werden sie direkt in den Signalweg eingebracht
  - ansonsten verhalten sie sich unauffällig
- **Mutanten:** „boshafte“ veränderte Komponenten des Modells
  - veränderte, existierende Komponenten injizieren Fehler

**Modifikation der Simulation**  $\leadsto$  vgl. hardware-basierte Lösung

- erfordert keine Veränderung des Modells sondern des Simulators
- Injektion von Fehlern an beliebigen Stellen/Zeitpunkten
  - Modifikation von Zuständen oder Signalen



## Simulations-basierte Fehlerinjektion (Forts.)

### 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 Akkuratheit des Systemmodells ab
  - kein 100%-iges Abbild der Realität ( $\leadsto$  keine „Echtzeitsimulation“)



## Gliederung

- 1 Überblick
- 2 FARM
- 3 Fehlerinjektionstechniken
- 4 FAIL\*
- 5 Zusammenfassung



## Fehlerinjektion: Praktische Probleme

- Validierung von CoRed (s. Folie V/?? ff.) durch Fehlerinjektion
    - Fehlerinjektion über die 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**
    - Beschreibung von Experimenten ist **nicht standardisiert/wiederverwendbar**
    - **hoher zeitlicher Aufwand:** 1s/Experiment  $\leadsto$  110h/400000 Experimente
  - ☞ 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
- $\leadsto$  eine einfache Verwendung „out-of-the-box“ unmöglich



## Lösungsansatz: FAIL\* [1]

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

- vorrangiges Entwurfsziel: Flexibilität bei der Fehlerinjektion

### ☞ Verwendung existierender virtueller Plattformen (Bochs, OVP, ...)

- aktuelle, gewartete Softwarebasis
- schneller Wirtsrechner  $\leadsto$  schnelle Durchführung von Experimenten
- voller Zugriff auf und volle Kontrolle über die Plattform

### ☞ Schaffung einer abstrakten Schnittstelle zu diesen Plattformen

- wiederverwendbare Beschreibung von Experimenten

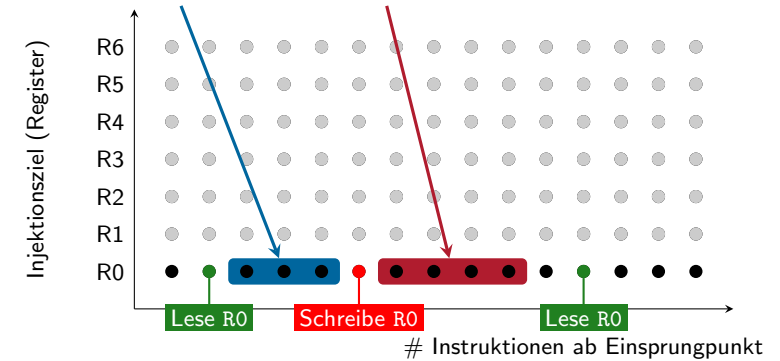
### ☞ mehr dazu in den Übungen!



## 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

$\leadsto$  sie lassen sich hervorragend parallelisieren

- auf mehreren Kernen, Prozessoren, Rechnern, ... in der Cloud



## Gliederung

### 1 Überblick

### 2 FARM

### 3 Fehlerinjektionstechniken

### 4 FAIL\*

### 5 Zusammenfassung



## Zusammenfassung

### FARM-Modell für Fehlerinjektion

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

### Fehlerinjektionstechniken $\mapsto$ grundlegende Kategorisierung

- {hardware, software, simulations, emulations}-basiert

### FAIL\* $\mapsto$ Grundlage für generische Fehlerinjektion?

- basierend auf virtuellen Zielsystemen
- flexible Plattform für Fehlerinjektion
- schnelle Experimentdurchführung durch Parallelisierung



- [1] AL., H. S.:  
**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
- [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] 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



- [4] 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

