Fault Injection: Techniques, Tools and Tricks

EECE 513: Design of Fault-tolerant Systems
Learning Objectives

• Define fault-injection and explain its uses
• Design a fault-injection experiment for measuring reliability
• Apply software and hardware techniques for fault injection
• Apply formal techniques for the assessment of fault tolerance
Dependability Evaluation

Dependability Evaluation Techniques

Model-Based
- Analytical
- Simulation

Measurement-based
- Fault-injection
- Real failures

Design
Prototype
Operational
Fault-injection

• Fault-injection (or fault-insertion) is the act of deliberately introducing faults into the system in a controlled and scientific manner, in order to study the system’s response to the fault
  – Can be used to estimate coverage of dependability mechanisms (e.g., detection, recovery)
  – Also used to understand inherent fault tolerance
  – To obtain reliability estimates of the system prior to deployment (requires statistical projection)
Why fault-injection?

• **Versus Model-based**
  – More realistic, as it evaluates actual system
  – No need to worry about mathematical feasibility
  – No need to supply input parameters

• **Versus operational measurements**
  – Failures take a *long* time to occur and when they do, are often not reproducible or analyzable
  – Failures provide limited insight into what *can* go wrong
  – One has to wait until the system is deployed, which may be too late
Learning Objectives

• Define fault-injection and explain its uses
• Design a fault-injection experiment for measuring reliability
• Apply software and hardware techniques for fault injection
• Apply formal techniques for the assessment of fault tolerance
Fault-Injection Setup

Fault Injection Spec’s
- Injection Strategy
  - Stress-based
  - path-based
  - Random
- Injection Method
  - by hardware
  - by software
- Fault Location
  - CPU
  - Memory
  - disk I/O
  - network I/O
  - Other I/Os
- Injection Time
  - load threshold
  - program execution path
  - fault arrival rate

Workload Spec’s
- Rates and Mixes
- Interaction
- Intensity

System Under Test
- CPU
- MEMORY
- I/O

Fault Injector

Workload Generator

Measure

Load Level
Fault-injection Steps

1. Identify fault-injection points and times
2. Choose workload and platform to inject
3. Start workload on platform with instrumentation
4. Collect the outcome of the expt.
5. Compare the outcome with the correct one
6. Inject fault at the appropriate time and point
Fault-injection: Inputs/Outputs

• **Inputs**
  – Workload and platform to inject?
  – When and where to inject?
  – How many faults to inject (total)?

• **Outputs**
  – How many faults were activated?
  – How many faults cause a deviation of the outcome?
  – What is the latency of manifestation?
Measures to Compute

- What fraction of injected faults are activated?
- What fraction of activated faults manifest as failures?
- What are the average activation and failure latencies?
Assumptions/Requirements

• A representative set of faults must be injected
  – Need to include enough faults to give confidence in
    the measures being studies

• Typically only one fault injected in each run
  – Ability to map the outcome to a set of faults

• Need to have a specification of correct behavior
  to distinguish incorrect outcomes
  – May need to determine golden run ahead of time
Learning Objectives

• Define fault-injection and explain its uses
• Design a fault-injection experiment for measuring reliability
• Apply software and hardware techniques for fault injection
• Apply formal techniques for the assessment of fault tolerance
Levels of Fault-Injection

• Fault-injection can be performed at multiple levels, from hardware to software

• Three things to consider in choosing level
  – Type of fault to inject (e.g., stuck at faults easier to inject in the hardware than in software)
  – Speed of injection (e.g., h/w simulation slower than real execution, though direct h/w probes possible)
  – Intrusiveness (e.g., probing hardware result in physical modifications that change the system’s characteristics)
Fault-Injection and Fault-Models

**Hardware**
- Open
- Bridging
- Stuck-at
- Power Surge
- Spurious Current
- Bit-flip

**Software**
- Storage Data Corruption
  - Registers, Memory, Disk
- Communication data corruption
  - CRC errors, Bus Errors
- Software defect emulation
  - Machine code corruption, source code mutation
Hardware fault-injection

**Contact-based**

- **Active Probes:** Alters the current via probes attached to the pins
  - Usually limited to stuck-at-faults, though bridging faults can also be modeled
  - Care must be taken to not damage the pins

- **Socket based:** Insert a socket between the target hardware and the circuit board
  - Can inject stuck-at or other logical faults

**Non-contact based**

- **Heavy-ion Radiation:** Put the chip in an accelerator beam (e.g., TRIUMF, Los Alamos)
  - Difficult to control and reproduce
  - But injects realistic faults
  - No restriction on where faults can be injected

- **Placing chip in an EM field**
  - Can lead to permanent damage
H/W Fault-Injection: Example (Contact Based)

Messaline from CNRS [Arlat’1990]: Can perform probe-based and socket-based injection. Used for evaluating safety-critical systems such as railway control system.
Hardware Fault-Injection (Non-contact Based)

FIST from Chalmers [Karlson’1995]: Used a Vacuum chamber in which an ionizing source was placed. A second non-faulty processor was used for state comparison.
Software-based Fault-Injection (SWIFI)

**Pros**
- Do not require expensive hardware modifications
- Can target applications and OS errors
- Many hardware faults do not require probes, e.g., register data corruption

**Cons**
- Restricted to inject only faults that S/W can see
- May perturb the workload that is running on the system, resulting in missing many heisenbugs
- Coarser-grained time resolution than h/w
SWIFI: Types

**Compile-time**
- Modify source code or machine code of the program prior to execution
- Can be used to model software defects
- Requires going thro’ compile-run cycle each time

**Runtime**
- Modify the program or its data during runtime
- Can be done through the debugger, kernel or with support from compiler
- No need to go through compile-run cycle each time
Compile-time Injection

• Modify program’s code prior to execution
  – Model hardware transient faults in machine code
  – Also, allows for modeling of software errors

• Example of software errors modeled
  – Missing initialization (corrupt initialized value)
  – Incorrect conditionals (Change $<=$ to $<$)
Example: G-SWIFT

- Injects compile-time faults in the machine code
  - Search Patterns: Patterns of machine code that represent common high-level programming constructs
    - Mutation based on Orthogonal Defect Classification (ODC)
  - Low-level Faults: Faults in a single machine-code insn
    - Mutation based on flipping bits of instructions

Example: G-SWIFT

- **Missing Function Call (Search pattern)**
  - Look through the machine code for patterns corresponding to a function call and replace it with No-ops
  - Need to replace return value with its prior value

- **Missing variable Initialization (low-level)**
  - Find the instruction that assigns a constant to the variable and replace it by a Noop or randomly perturb the constants
G-SWIFT: Results

• Ability to emulate almost source-level faults according to ODC at the machine code level
  – Most discrepancies due to the use of C macros

<table>
<thead>
<tr>
<th>Program</th>
<th>Source-level Faults</th>
<th>Machine-code faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gzip</td>
<td>71</td>
<td>80</td>
</tr>
<tr>
<td>Lzari</td>
<td>110</td>
<td>117</td>
</tr>
<tr>
<td>Camelot</td>
<td>67</td>
<td>75</td>
</tr>
</tbody>
</table>

Results found to hold across a range of compilers and architectures.
Runtime Injection

• Advantages
  – Can inject faults without recompiling - speed
  – Faults can occur deeper in the execution. e.g., one-millionth iteration of a loop
  – Fault can depend on runtime conditions. e.g., if memory usage exceeds a threshold, inject fault

• Examples of faults: Timeouts, dynamic code injection, resource exhaustion, data corruption
Example: NFTAPE

• Framework for conducting automated fault/error injection based dependability characterization

• Enables user to:
  – specify a fault/error injection plan
  – carry on injection experiments
  – collect the experimental results for analysis

• Facilitates automated execution of fault/error injection experiments

• Enables assessment of dependability metrics including availability, reliability, and coverage

• Operates in a distributed environment
NFTAPE Features

• Debugger-based fault injector
  – injection to the target process memory and registers

• Driver-based fault injector
  – injection to memory, registers, OS components

• Use of performance monitors (built into CPUs)
  – trigger fault injection; measure error latency

• Fault injection targets
  – CPU registers, memory, applications, specific OS functions

• Fault injection triggers:
  – random (based on time), application supplied breakpoint, externally supplied breakpoint
# Injection Targets and Outcome Categories

<table>
<thead>
<tr>
<th>Target</th>
<th>User space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>Functions: e.g., main</td>
</tr>
<tr>
<td></td>
<td>Instructions: any or selected subset (e.g., branch, load, store)</td>
</tr>
<tr>
<td>Data</td>
<td>Static data and dynamically allocated memory (heap)</td>
</tr>
<tr>
<td>Stack</td>
<td>Data on an application stack</td>
</tr>
<tr>
<td>CPU Registers</td>
<td>General purpose registers</td>
</tr>
<tr>
<td>Memory range</td>
<td>Any location in application memory space</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outcome Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activated</td>
<td>The corrupted instruction/data is executed/used.</td>
</tr>
<tr>
<td>Not Manifested</td>
<td>The corrupted instruction/data is executed/used, however it does not cause a</td>
</tr>
<tr>
<td></td>
<td>visible abnormal impact on the system.</td>
</tr>
<tr>
<td>Fail Silence Violation</td>
<td>Either operating system or application erroneously detects the presence of</td>
</tr>
<tr>
<td></td>
<td>an error or allows incorrect data/response to propagate out. Workload</td>
</tr>
<tr>
<td></td>
<td>programs are instrumented to detect errors.</td>
</tr>
<tr>
<td>Crash</td>
<td>Application/OS stops working, e.g., bad trap or system panic. Crash</td>
</tr>
<tr>
<td></td>
<td>handlers embedded into operating system are enhanced to enable dump of</td>
</tr>
<tr>
<td></td>
<td>failure data (processor and memory state).</td>
</tr>
<tr>
<td>Hang</td>
<td>System resources are exhausted resulting in a non-operational application</td>
</tr>
<tr>
<td></td>
<td>/system, e.g., deadlock or livelock.</td>
</tr>
</tbody>
</table>
NFTAPE Framework Configuration

Control Host

- Campaign Script
- Control Host
- Log

Error Injection Targets

- Process Manager
  - Injector Process
  - Application Process

- Injector Process
  - Application Process
NFTAPE: Results

- Used to evaluate NASA’s space imaging application: Part of MARS REE project

Fault injections performed in memory:

a) Regular image
b) Image with random noise
c) Application output with no faults
d) Application output with NFTAPE injecting faults

Figure 4. Images from Space Imaging Application.
Learning Objectives

• Define fault-injection and explain its uses
• Design a fault-injection experiment for measuring reliability
• Apply software and hardware techniques for fault injection
• Apply formal techniques for the assessment of fault tolerance
Validation of FTMs

- Detector Validation
  - Fault-injection
  - Formal Verification

Formal Verification – Complements FI by exposing corner-case scenarios in error and attack detectors
SymPLIFIED: Existing Techniques

Program Verification: Finding software bugs in programs

Fault-tolerance Validation: Specific to fault-tolerance mechanism

Hardware Verification: Finding design errors in circuits

Fault-tolerant algorithms: Assumes specific application model

Need for a formal framework to evaluate the effects of hardware errors on arbitrary programs independent of detection mechanism
Formal Framework for Detector Validation

Assembly Language Program

Formal Verification Technique

Enumeration of all errors that escape detection

Input: Application protected with embedded error detectors

SymPLFIED

Model-checking Symbolic Execution

Output: Understanding of the limitations of error detectors

33
SymPLFIED : Approach

- Analyze program written in MIPS assembly language
  - Low-level state made explicit (e.g., stack pointer)
  - Oblivious to post compile-time transformations
  - Both programs and libraries (statically linked)

- Generic representation of error detectors
  - Allow arbitrary error detectors to be specified in program

- Fault Model: H/W transients (memory/register/PC)

- Comprehensive enumeration of undetected errors that lead to program failure → detector defects
SymPLFIED: Symbolic Execution

- Exhaustive enumeration leads to state space explosion
- Represent all error values in program as an abstract symbol
  - Track propagation of errors symbolically
  - Abstraction may lead to false-positives
SymPLFIED: Propagation Example

ori $2$ $0$ #1
read $1$
mov $3$, $1$
ori $4$ $0$ #1

loop: setgt $5$ $3$ $4$
beq $5$ 0 exit
mult $2$ $2$ $3$
subi $3$ $3$ #1
beq $0$ #0 loop

exit: prints "Factorial ="
print $2$
halt

($3 > $4) = true
Exit loop

($3 > $4) = false
Reenter loop and continue

ture => \{N!, (N ! \div 2!), (N ! \div 3!) \ldots 1\}
false => err
SymPLFIED: Detection Example

ori $2 $0 #1
read $1
mov $3, $1
ori $4 $0 #1
loop: setgt $5 $3 $4
beq $5 0 exit
check ($3 >= 1)
mov $6, $2
mul $2 $2 $3
check ($2 <= $6 * $1)
subi $3 $3 #1
beq $0 #0 loop
exit: prints "Factorial = "
print $2
halt

($3 > $4) == false
Exit loop

($3 > $4) == true
(Re)-Enter loop

ture

($2 <= $6 * $1) == true, if (1 <= $3 <= $1)

($2 <= $6 * $1) == false, if (1 <= $1 < $3)
SymPLFIED: Design

- Machine Model
  - (Memory, Registers, Instructions)

- Error Model
  - (Register errors, memory errors, control-flow errors)

- Detector Model
  - (Specification and execution of error detectors)

- Maude Modules
  - (Rewriting logic)

- Verification technique
  - (Model-Checking)

Modular framework allows decoupling of detection mechanism and error class from the machine model and verification technique.
SymPLFIED: Machine Model

\[
\text{eq } \{ C, < \text{addi } rs \text{ rd } v, \text{PC}(pc) \text{ regs}(R) S > \} \\
\{ C, < \text{fetch}( C, \text{next}(pc) ), \text{PC}(\text{next}(pc)) \text{ regs}(R[rd <- (R[rs] + v)]) S > \} \\
\text{eq } \text{next}(pc) = (pc + 1) \text{ if not } \text{terminal}(pc) \\
\text{eq } \text{fetch}( [L \mid I] C, L ) = I \\
\text{eq } \text{fetch}( \text{NoCode}, L ) = \text{throw instException}
\]
SymPLFIED: Error Model

• **Data Errors**
  - $\text{rl } (\text{Err} == \text{Int}) \Rightarrow \text{False}$
  - $\text{rl } (\text{Err} == \text{Int}) \Rightarrow \text{True}$

• **Address Errors**
  - $\text{rl } \text{M}[\text{Err}] \Rightarrow \text{anyAddr(M)}$
  - $\text{rl } \text{M}[\text{Err}] \Rightarrow \text{addrException}$

• **Control Errors**
  - $\text{rl } \text{fetch( C, err )} \Rightarrow \text{anyLabel( C )}$
  - $\text{rl } \text{fetch( C, err )} \Rightarrow \text{instException}$
SymPLFIED: Detector Model

Specified as quintuple:

\[ \text{Det}(\text{Id}, \text{pc}, \text{left expr}, \text{right expr}) \]
Checks when program counter==pc if,
\[ \text{value( left expr)} == \text{value( right expr)} \]

Detector Example

check(4, 10, !($4) + *(#3), @(#5) )
If not, throw checkException(4)
SymPLFIED: Model Checking

Exhaustive search feature (bounded model checking):

\[
\text{search AllRegisterErrors ( start(program, first, input ) =>!} \\
(S:\text{State}) \text{ such that ( getOutput(S) contains 2) } \\
\text{and ( getException(S) == noException) .}
\]

- Timeout (no. of instructions) must be specified
  - Obtained by profiling program with inputs

- Input(s)/Output(s) must be specified for execution
  - Comparison with golden output for determining failure
SymPLFIED: Implementation

• Written using a rewriting logic tool - **Maude**
  – Modular and extensible framework - code reuse
  – Supports direct execution of programs + libraries
  – Wide spectrum of formal techniques can be used

• **Architecture**: Consists of three main parts
  – **Front-end**: Models processor-specific details and interprets assembly language program
  – **Backend**: Performs symbolic/concrete evaluation of program under a given fault model
  – **Interface primitives**: Link front-end to back-end through a generic interface for easy extensibility
SymPLFIED: Implementation contd..

- SS assembly to Maude translator
- DLX to Maude translator

SS instruction set Frontend (Maude)
DLX instruction set Frontend (Maude)

Interface Primitives (Maude)

Concrete evaluation Backend (Maude)
Symbolic Execution Backend (Maude)
Constraint Solving Backend (Maude)
SymPLFIED: Output

• States that lead to the error with info about
  – Program Counter, Registers, Memory state
  – History of branches taken in program
  – Fault-injection and activation logs
  – State of input/output streams, exceptions
  – Constraints on register/memory values

• Correlate with the code to trace how the error happened and improve detection mechanisms
SymPLIFIED: Case Study

- **Tcas: Application Characteristics**
  - FAA mandated Aircraft collision avoidance system
  - Rigorously verified protocol and implementation
  - About 150 lines of C code = 1000 lines of assembly

**Inputs:** Positional parameters of other aircraft (and self)

**Outputs:**
- 0 – Unresolved
- 1 – Ascend
- 2 – Descend
SymPLFIED: Results

- Found one potentially catastrophic output considering all possible register errors
  - Output of 2 (descend) instead of 1 (ascend)
  - Many cases where the output is unresolved (0)

- Highly-parallelizable code of SymPLFIED
  - Took about 4 minutes on a 150 node cluster
  - Total of 600 minutes (10 hours) of machine time

- Not exposed by random fault-injection
  - Used SimpleScalar simulator for experiments
  - Ran for more than 40 hours on single machine
SymPLFIED: Tcas Error

```c
int alt_sep_test()
{
    alt_sep = UNRESOLVED;
    if (enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped)) {
        need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
        ...
        if (need_upward_RA && need_downward_RA)
            alt_sep = UNRESOLVED;
        else if (need_upward_RA)
            alt_sep = UPWARD_RA;
        else if (need_downward_RA)
            alt_sep = DOWNWARD_RA;
    }
}

(1) Assembly-language level reasoning needed to expose error
(2) Random injection needs to hit both type and location of fault

need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();

Non_Crossing_Biased_Climb: Return address in register $31 corrupted by transient error
```
SymPLFIED: Summary

• **Formal framework to evaluate the effects of runtime errors on programs with detectors**
  – Analyze programs directly in assembly language
  – Comprehensive enumeration of failure-causing errors

• **Use of symbolic execution + model-checking**
  – Abstraction techniques to track error propagation

• **Tested on aircraft collision avoidance app.**
  – Found catastrophic error in main controller code
  – Not found using random fault-injection experiment
Learning Objectives

• Define fault-injection and explain its uses
• Design a fault-injection experiment for measuring reliability
• Apply software and hardware techniques for fault injection
• Apply formal techniques for the assessment of fault tolerance