FIG: Fault Injection in glibc

Pete Broadwell, Naveen Sastry
Jonathan Traupman
Presentation Outline

1. Introduction
   - Objective/Motivation
   - Background

2. Methods
   - Implementation
   - Test setup

3. Evaluation
   - Test results
   - Conclusions
Project Objective/Motivation

Objective:
• Develop a tool for injecting faults at the system boundary

Motivation:
• Developers are lazy
• We need testing tools that generate a wide variety of unexpected faults
“Software's Invisible Users”

User Input

User interface

Application

Other libraries

System libraries (libc)

OS

Other apps

Concept: Jim Whittaker, Center for Software Engineering Research, Florida Institute of Technology
Related Testing Methods

1. **CMU’s Ballista project**
   - “Top-down” testing of POSIX-compliant OS and library interfaces

2. **Sandboxing program modules**
   - Modules tested in isolation from rest of system

3. **CSER’s Holodeck**
   - Similar approach to ours, but proprietary and only available for Windows 2000/XP
The Competition: Holodeck
Implementation

- Thin stub library between app & libc
- Traps API calls
  - Logs them
  - Inserts faults
- Can be inserted into any app without modification
  - Uses LD_PRELOAD environment variable

Diagram:

- Application
- libfig.so
- libc.so
- OS

Flow:

- Normal call path
- Injected fault
Test Setup

• Used FIG to inject faults into four common UNIX applications
  - ls
  - emacs
  - Apache
  - Berkeley DB

• Used malloc, open, close, read, write, and select system calls
Extensibility

- API stubs are automatically generated
- Very easy to add new APIs to log
- Fault injection under script control

Sample control file:

```bash
MALLOC_INDEX
interval 82 to infinity return 0
    errno ENOMEM probability 0.03

OPEN_INDEX
// device out of space.
interval 100 to infinity return -1
    errno ENOSPC probability 0.001
// kernel out of memory.
interval 100 to 120 return -1
    errno ENOMEM probability 0.1
// too many files open.
callnumber 108 return -1
    errno EMFILE
    probability 1.0
```
Test Results I

- **emacs:**
  - Much better without X
  - malloc - no end to the misery
  - Typical result: core dump or silent exit
  - No service for file I/O errors

- **ls**
  - Smart about malloc failures
  - Apparently keeps global error count
  - Fails on 6th error
Test Results II

- **Apache**
  - Best of show
  - Upfront resource provisioning
  - Degraded service possible
  - Properly handles EINTR

- **Berkeley DB**
  - Simple non-transactional dictionary application
  - Detects & reports errors
  - On write errors, high chance of database corruption
  - Doesn’t even handle EINTR!
Lessons Learned

• Each application had something interesting to reveal
• Simple tricks help:
  - resource preallocation
  - retries
• Basic Unix tools should be robust
Further Work

• Richer control language
• Interactive GUI
• Better fork / exec handling
  - Goal: do better than strace
• Evaluate software design process with such a tool
• Production use (à la ROC)