Utilizing Inherent Diversity in Complex Software Systems

Peter Okech, Nicholas Mc Guire, Christof Fetzer

ASSC 2014 – Melbourne May 30
Authors

• Peter Okech
  – Tutorial Fellow, Department of Physics, University of Nairobi (UoN)
  – PhD student, UoN and visiting student at TU-Dresden

• Nicholas Mc Guire
  – OpenTech EDV Research GmbH Austria and,
  – Beijing Shenzhou Aerospace Technology Co., Beijing, P.R. China

• Prof. Dr. Christof Fetzer
  – Chair, Systems Engineering Group, TU-Dresden
Computing Platforms
Advances in Technology

• Processor Architecture
  • Sequential ==> Superscalar ==> Multicore/Manycore

• Executive Runtime/Operating System
  • operating systems grow in complexity to provide required abstraction

  Overall complexity is rising!

• Our focus is on system software in safety context
Complexity Challenges

• Complex software
  • Always has residual faults
    – Current techniques cannot eliminate all faults
  • Is difficult to justify
    – How do you demonstrate that the system is capable of achieving tolerable failure rates for safety functions?
Complexity Opportunities

Can we take advantage of complexity to achieve safety?
Our Approach - Key Concepts

• Exploit the non determinism in execution of programs in complex HW/SW platforms
• Use architectural means to protect against systematic residual faults

**We present and argue these on preliminary data**

• Define a strategy for arguing safety based on inherent diversity
Assumptions

- Techniques to provide coverage of HW faults are implemented
- Safety Application: correct and verified
- Kernel is complex, and contains systematic residual faults
Compliance Routes

• Options available to software safety engineers (for example [IEC 61508, 2010])
  • Develop, Verify and Implement (route 1S)
  • Proven in use (route 2S)
  • Assessment of non-compliant development (route 3S)
Design Diversity

• Technique suggested to protect redundant systems from CCFs [Randell, 1975]
• Common approach to generate diverse versions is through N-Version programming [Avizienis & Chen 1977]
• Forced diversity through use of different languages/methodologies [Littlewood, 1996]
• Design diversity is effective [Littlewood et al. 2001]
Design Diversity (2)

• Does the gains match the theoretical expectations?
  – Versions shown to have positive correlation of failure [Knight & Leveson 1986, Eckhardt & Lee 1985, Littlewood & Miller 1989]

• Expensive to generate versions
  – Solution? - Automatically generate the versions, e.g. compiler generated versions
Diversity in Security

- Use of randomization to generate diverse copies of software
- Obscure systematic faults by introducing non-determinism in the execution environment.
- **Take home**: Systematic faults can be made to manifest as random faults through randomization
Inherent Diversity

• Complex systems have large non-predictable internal states
  – non-deterministic paths from input to output
• We expect, that due to inherent non-determinism, a 2oo2 system executing the same application on a complex operating system will diverge in state
Channel A

Application

System call interface

if(state){
    ...
    undetected fault
    ...
} else {
    ... tested path
}

Kernel (state A)

Channel B

Application

System call interface

if(state){
    ...
    undetected fault
    ...
} else {
    ... tested path
    ...
}

Kernel (state B)
Evaluation of Non-Determinism

• Is there non-determinism in program execution in kernel space?

Experiment Description

• Test program
  – **Main task**: read value from a file, increment and store new value.
  – task executed in a loop (20,100 iterations)

• Environment
  – Intel Core 2 Duo
  – Linux kernel 3.12.0
  – GCC 4.4.5

```
BEGIN
  open file
  for count from startValue down to 0
    read data from file
    increment data
    write data to file
    sleep for 10s
  close file
END
```
Evaluation of Non-Determinism (2)

• Traces of kernel functions called by the test program recorded

• System calls in the test-program loop

  \{read, write, lseek, fsync, rt_sigaction, \rt_sigprocmask, nanosleep\}
Execution Variability

- Variations observed both in timing and execution path

<table>
<thead>
<tr>
<th></th>
<th>Duration in us</th>
<th>length of Path</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>max</td>
</tr>
<tr>
<td>read</td>
<td>15.279</td>
<td>23253.259</td>
</tr>
<tr>
<td>write</td>
<td>8.836</td>
<td>407.395</td>
</tr>
<tr>
<td>lseek</td>
<td>1.915</td>
<td>144.872</td>
</tr>
<tr>
<td>fsync</td>
<td>8418.372</td>
<td>753850.994</td>
</tr>
<tr>
<td>nanosleep</td>
<td>9990266</td>
<td>9990638</td>
</tr>
</tbody>
</table>
Time Distribution of \textit{read}
Path Variability

• The characteristics of the paths

<table>
<thead>
<tr>
<th>Function</th>
<th>Call Instances</th>
<th>Distinct Paths</th>
<th>Freq.</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>20101</td>
<td>636</td>
<td>6057</td>
<td>30.13%</td>
</tr>
<tr>
<td>write</td>
<td>20132</td>
<td>768</td>
<td>11748</td>
<td>58.35%</td>
</tr>
<tr>
<td>lseek</td>
<td>40200</td>
<td>33</td>
<td>40164</td>
<td>99.91%</td>
</tr>
<tr>
<td>fsync</td>
<td>20100</td>
<td>16139</td>
<td>36</td>
<td>0.18%</td>
</tr>
<tr>
<td>nanosleep</td>
<td>20100</td>
<td>174</td>
<td>14189</td>
<td>70.59%</td>
</tr>
<tr>
<td>rt_sigaction</td>
<td>20100</td>
<td>22</td>
<td>20078</td>
<td>99.89%</td>
</tr>
<tr>
<td>rt_sigprocmask</td>
<td>40200</td>
<td>81</td>
<td>40100</td>
<td>99.75%</td>
</tr>
</tbody>
</table>
Paths Taken by *read*

- Analysis of the top 14 paths of the read system call instances

<table>
<thead>
<tr>
<th>Top N paths</th>
<th>Number of Paths</th>
<th>Cumulative</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No.</td>
<td>%</td>
</tr>
<tr>
<td>1</td>
<td>6057</td>
<td>30.13%</td>
</tr>
<tr>
<td>2</td>
<td>4911</td>
<td>24.43%</td>
</tr>
<tr>
<td>3</td>
<td>2714</td>
<td>13.50%</td>
</tr>
<tr>
<td>4</td>
<td>2242</td>
<td>11.15%</td>
</tr>
<tr>
<td>5</td>
<td>1018</td>
<td>5.06%</td>
</tr>
<tr>
<td>6</td>
<td>466</td>
<td>2.32%</td>
</tr>
<tr>
<td>7</td>
<td>450</td>
<td>2.24%</td>
</tr>
<tr>
<td>8</td>
<td>306</td>
<td>1.52%</td>
</tr>
<tr>
<td>9</td>
<td>211</td>
<td>1.05%</td>
</tr>
<tr>
<td>10</td>
<td>182</td>
<td>0.91%</td>
</tr>
<tr>
<td>11</td>
<td>152</td>
<td>0.76%</td>
</tr>
<tr>
<td>12</td>
<td>148</td>
<td>0.74%</td>
</tr>
<tr>
<td>13</td>
<td>142</td>
<td>0.71%</td>
</tr>
</tbody>
</table>
Which Path is Taken Next?

Instances in call order

execution paths taken

read system call instances
Safety Justification

• We suggested a justification approach based on inherent diversity

• Has two aspects:
  • Safety argument
  • Procedure for assessment
The Model

• Testable paths
  • safe by testing
• Untestable paths == Rare paths
  • inherent diversity + architectural protection
    − minimum 2oo2 system
The Strategy

- Frequently taken Paths
- Rarely taken Paths

- Statistically Independent
- Not Independent

Faults covered by Verification and Validation

Faults covered by architectural means based on inherent diversity
Requires deeper analysis
Conclusion

• These are our early results
• We intend to run the same experiments on a true 2002 system
• We have suggested the concept of Inherent diversity based on the observations:
  – HW technology is undergoing fundamental changes
  – Complexity of system software is growing
  – Traditional approaches to deriving diversity have limitations
Conclusion (2)

- More work still needed to establish the inherent diversity approach
- Not a silver bullet to the assurance demands for verification of systems utilizing complex hardware/software
  - but inherent diversity shows a promising potential
Thank you