Review
Network reliability
Diameter stability
Multistage networks (butterfly)
extra stage to improve reliability
Computing overall reliability
Computing available bandwidth
Computing availability of processors to memory connections

Redundancy ideas for other networks
2-D Mesh Extra row and column of switches
Extra switches in grid

Redundancy for Hypercube

Redundancy for Rings
Finding path (or end-to-end) reliability in a network
Need to find all possible paths

![Diagram showing network with nodes and paths](image)

Reliability between $N_1$ to $N_4$
Three paths

- $P_1=\{X_1,2,X_2,4\}$
- $P_2=\{X_1,3,X_3,4\}$
- $P_3=\{X_1,2,X_2,3,X_3,4\}$

We cannot simply add the reliabilities of these paths, since there are some common nodes and links and the reliabilities of those will be counted more than once

We can use the idea of representing states for all possible failures of links or nodes
In this example we can also explore mutually exclusive events

Path 1 is operational
Path 1 is not operational but path 2 is operational
Paths 1 and 2 are not operational but path 3 is operational

\[
R_{N_1,N_4} = p_{1,2}p_{2,4} + p_{1,3}p_{3,4} \left[1 - p_{1,3}p_{3,4}\right] + p_{1,2}p_{2,3}p_{3,4} \left[q_{1,3}q_{2,4}\right]
\]

\[
E_1 \cup E_2 \cup \ldots \cup E_m = E_1 \cup (E_2 \cap \overline{E_1}) \cup (E_3 \cap E_1 \cap \overline{E_2}) \cup \ldots \cup (E_m \cap E_1 \cap E_2 \cap \ldots \cap \overline{E_{m-1}})
\]

\[
R_{N_1,N_4} = \text{Prob}[E_1] + \text{Prob}[E_2 \cap \overline{E_1}] + \ldots + \text{Prob}[E_m \cap E_1 \cap E_2 \cap \ldots \cap \overline{E_{m-1}}]
\]
Chapter 5 Software Fault tolerance

Why do software contain bugs
Design failures
Error in coding

Note some consider not meeting requirements as failure – we will not

Testing – can we assure that testing leads to fault free software?

Path testing – test for all possible control flow paths
Integration testing – test all modules
Combinatorial testing → test different combinations of modules

With more complex software, exhaustive testing impractical
Even generating test patterns is complex

Can we mathematically prove a program (at least as designed) is correct?
Again can be very complex

Modeling software using formal logic (Model Checking)

predicate logic and higher logics (and theorem proving tools)
or petri nets
of dataflow graphs

If we have time, I will return to these formal models

how we can make software fault tolerant?

First we need to be able detect failures
incorrect results?
out of range values?

Acceptance tests
How do we know what are acceptable results?
Can use models or “common sense”
-40 C in summer may mean that something is wrong
Timing Checks. If we have a good idea on how long a software system takes, we can use timers to detect errors such as infinite loops or some other reasons for extended execution times.

Random or probabilistic checks

Textbook gives an example

Consider Matrix multiplication: \( A \times B = C \)

Consider a vector \( V \) generated randomly.

Then compute

\[ A \times (B \times V) \quad \text{and} \quad C \times V \]

If these resulting vectors are equal, you have high confidence that \( C \) is the correct result.

But we are doing additional computations

- Original computation = \( O(n^3) \)
- Extra computation = \( O(n^2) \)

Verification of Inputs and Outputs

Make sure inputs are within acceptable values

Check for the acceptability of outputs

Inputs – domain of values

Outputs – Range of values

Can include checks of range of values with all program variables

- bounds on temperature
- bounds on vehicle speeds

Two issues to keep in mind with bounds

Sensitivity: Conditional probability that bounds check fails given that the output is in error – Do we detect all errors (false negatives)

Specificity: conditional probability that output is in error, given that the bounds check failed – Do we miss errors (false positives)
How to implement acceptance tests?
Use wrappers
The wrapper filters inputs and outputs
Textbook gives a few examples

Buffer Overflow:
Some programming languages do not perform range checking on arrays - can cause accidental or malicious damage
Write a large string into a small buffer: buffer overflow - memory outside buffer is overwritten
If accidental – can cause a memory fault
If malicious - overwriting portions of program stack or heap - a well-known hacking technique

Stack-smashing attack:
A process with root privileges stores its return address in stack
Malicious program overwrites this return address
Control flow is redirected to a memory location where the hacker stored the attacking code
Attacking code now has root privileges and can destroy the system

How can we use a Wrapper to protect against these “faults” or attacks
A wrapper around malloc calls from a program
All malloc calls from the wrapped program are intercepted by wrapper
Wrapper keeps track of the starting position of allocated memory and size
Writes are intercepted, to verify that they fall within allocated bounds
If not, wrapper does not allow the write to proceed and instead flags an overflow error
Example 2: Checking correctness of scheduling
(Incorrect scheduling can miss deadlines)

Wrapper around task scheduler in a fault-tolerant, real-time system

Such schedulers may use Earliest Deadline First (EDF) - execute task with earliest deadline among tasks ready to run

Subject to preemptability constraints (tasks in certain parts of execution may not be preemptable)

A wrapper verifies correctness of scheduling algorithm:
- The ready task with earliest deadline was picked
- Any arriving task with an earlier deadline preempts the executing task (if latter is preemptable)

Wrapper needs information about the tasks, their deadlines, and their preemptability

Example 3: It is known that the software fails for certain inputs (or set of inputs)

- known bugs
- why not fix software to eliminate these bugs?

Wrapper intercepts inputs and checks if the input is in set S
If not, forward to software module for execution
If yes, return a suitable exception to system
Alternatively, redirect input to some alternative, custom-written, code that handles these inputs

Likewise, we can design a wrapper to check inputs for range and other properties
Factors influencing capabilities of acceptance tests

Quality of acceptance tests:
Application-dependent - has direct impact on ability of wrapper to stop faulty outputs

Availability of necessary information from wrapped component:
If wrapped component is a “black box,” (observes only the response to given input), wrapper will be somewhat limited
Example: a scheduler wrapper is impossible without information about status of tasks waiting to run

Extent to which wrapped software module has been tested
Extensive testing identifies inputs for which the software fails

Software Rejuvenation
Periodically “update” software
Examples given in book are not great
reboot, reclaim garbage, kill zombie tasks

Example of Rejuvenation: Rebooting a PC
As a process executes
it acquires memory and file-locks without properly releasing them
memory space tends to become increasingly fragmented
The process can become faulty and stop executing
To head this off, proactively halt the process, clean up its internal state, and then restart it
Rejuvenation can be time-based or prediction-based

Time-Based Rejuvenation - periodically
Rejuvenation period - balance benefits against cost

\[ N(t): \text{expected number of errors in } [0,t] \]
\[ C_e: \text{cost due to an error} \]
\[ P: \text{rejuvenation period} \]
\[ C_r: \text{cost of rejuvenation} \]

Total cost \( = N(P)C_e + C_r \)
Rate of rejuvenation \( = \frac{C_{\text{rate}}(P)}{P} = \frac{N(P)C_e + C_r}{P} \)

Rate of rejuvenation = \( 1/P \)
Example 1: Constant failure rate over time - \( R(P) = \lambda P \)

\[ C_{\text{rate}}(P) = \lambda C_e + C_r / P \]

Optimal \( P^* = \infty \) - no software rejuvenation

This equation does not model aging \( \rightarrow \) with time, higher rate of errors

Rejuvenation heads off the increased rate in failure as software ages

Suppose we want to model aging: that is the failure rate increases with time

Let us assume rate increases exponentially with time

\[ \hat{N}(P) = \lambda P^n, \quad n > 1 \]

Now the optimum rejuvenation rate is given by:

\[ P^* = \left( \frac{C_r}{(n-1)\lambda C_e} \right)^{1/n} \]

If \( n = 2 \)

\[ P^* = \sqrt{\frac{C_r}{\lambda C_e}} \]

- Estimating the parameters
  - Can be done experimentally - running simulations on the software
  - System can be made adaptive - some default initial values and adjusting the values as more statistics are gathered

Optimal rejuvenation period

The graph states “arbitrary time units”

So if \( \lambda = 1 \) hour, then rejuvenation periods are in hours
Prediction based rejuvenation. Schedule next rejuvenation based on past observations.

If we have an idea about the last k failures and their inter-arrivals, we can decide an optimal time for next rejuvenation.

Monitoring system characteristics - amount of memory allocated, number of file locks held, etc. - predicting when system will fail.

Example - a process consumes memory at a certain rate, the system estimates when it will run out of memory, rejuvenation can take place just before predicted crash.

The software that implements prediction-based rejuvenation must have access to enough state information to make such predictions.

If prediction software is part of operating system - such information is easy to collect.
If it is a package that runs atop operating system with no special privileges - constrained to using interfaces provided by OS.

For example: Unix provides the following utilities for collecting status information -
- `vmstat` - provides information about processor utilization, memory and paging activity, traps, and I/O
- `iostat` - outputs percentage CPU utilization at user and system levels, as well as a report on usage of each I/O device
- `netstat` - indicates network connections, routing tables and a table of all network interfaces
- `nfsstat` - provides information about network file server kernel statistics

Once we collected the information, we can use least squares fit to determine when to rejuvenation should be scheduled.

You can also explore other curve fitting on the data collected for example weighted least squares (more importance to certain activities)

The level of rejuvenation: Entire system, selected applications.

Can we use rejuvenation as a security effort?
How to eliminate malware? → data below refers to 2013-2014 timeframe

- McAfee's report shows that there are over 100,000 new malware instances detected in a given day.
- Damballa demonstrated that the typical lag time between malware release and detection using anti-malware is 54 days, almost 8 weeks.
- FireEye reported that 75 percent of unique malware were detected in only one environment, and 82 percent of malware binaries disappear within an hour – making difficult to develop signatures


Solutions to eliminating malware

- Reduce the signature generation time, but
  - It’s hard to collect samples in a short time
  - Malware is too entrenched in an environment to be detected
  - There is too many malwares introduced every day

- Periodic rejuvenation can be an alternative way to eliminate malware.
  - Restore to SAFE state
    - Safe state is built on the trust chain, e.g. trusted software, trusted patch, trusted virtual machine, and etc.
  - Can be used in addition to malware scans
  - Can be affordable with proper choice of periodicity of rejuvenation
Rejuvenation for Security

- The rejuvenation can be applied modularly and periodically in order to minimize the downtime of the system.
- Checkpoint good software module

Our experimentation

- Scenario

Malware # i

ClamAV 0.98.3
F-prot 6
Nod32 4

Instance A

Scan & Eliminate

(1)

Instance B

Rejuvenation

(2)

Instance B’

ClamAV 0.98.3
F-prot 6
Nod32 4

Scan & Eliminate

(3)
CSCE 5760: Design or Fault Tolerance

• Simulation Environment Specification.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud platform</td>
<td>Cloud platform</td>
</tr>
<tr>
<td>Flavor</td>
<td>m1.small</td>
</tr>
<tr>
<td>RAM</td>
<td>RAM</td>
</tr>
<tr>
<td>Processor</td>
<td>QEMU Virtual <a href="mailto:1.0@2.33GHz">1.0@2.33GHz</a>(1Core)</td>
</tr>
<tr>
<td>Instance operation system</td>
<td>Ubuntu 12.04</td>
</tr>
<tr>
<td>Instance Size</td>
<td>20 GB</td>
</tr>
<tr>
<td>Application service</td>
<td>Joomla 3.3</td>
</tr>
<tr>
<td>Database</td>
<td>MySQL 5.5.36</td>
</tr>
<tr>
<td>Compiler</td>
<td>PHP 5.4.27</td>
</tr>
<tr>
<td>Web service</td>
<td>Apache 2.4.9</td>
</tr>
</tbody>
</table>

Anti-malware software
- ClamAV 0.98.3
- F-prot 6
- Nod32 4

CSCE 5760: Design or Fault Tolerance

• Simulation results of test malware and vulnerabilities.

<table>
<thead>
<tr>
<th>Malware</th>
<th>Scope</th>
<th>Rejuvenation, Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backdoor.Linux.Ovason</td>
<td>Operating System</td>
<td>Restore, Eliminated</td>
</tr>
<tr>
<td>Backdoor.Linux.Phobi.1</td>
<td>Operating System</td>
<td>Restore, Eliminated</td>
</tr>
<tr>
<td>Backdoor.Linux.Rst.a</td>
<td>Operating System</td>
<td>Restore, Eliminated</td>
</tr>
<tr>
<td>Exploit.Linux.Da2.a</td>
<td>Operating System</td>
<td>Restore, Eliminated</td>
</tr>
<tr>
<td>Exploit.Linux.Race.1</td>
<td>Operating System</td>
<td>Restore, Eliminated</td>
</tr>
<tr>
<td>Net-Worm.Linux.Scalper.b</td>
<td>Operating System</td>
<td>Restore, Eliminated</td>
</tr>
<tr>
<td>Rootkit.Linux.Agent.sm</td>
<td>Operating System</td>
<td>Restore</td>
</tr>
<tr>
<td>Trojan.Linux.Rorkit.n</td>
<td>Operating System</td>
<td>Restore, Eliminated</td>
</tr>
<tr>
<td>Trojan.Tsunami.B</td>
<td>Operating System</td>
<td>Restore, Eliminated</td>
</tr>
<tr>
<td>Virus.Linux.Ox.8759</td>
<td>Operating System</td>
<td>Restore, Eliminated</td>
</tr>
<tr>
<td>Virus.Linux.Radix</td>
<td>Operating System</td>
<td>Restore, Eliminated</td>
</tr>
<tr>
<td>Virus.Linux.Silvio.b</td>
<td>Operating System</td>
<td>Restore, Eliminated</td>
</tr>
<tr>
<td>Virus.Linux.Snoopy.c</td>
<td>Operating System</td>
<td>Restore, Eliminated</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Scope</th>
<th>Rejuvenation, Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2013-1636</td>
<td>Joomla</td>
<td>Restore, Eliminated</td>
</tr>
<tr>
<td>CVE-2014-2440</td>
<td>MySQL</td>
<td>Restore, Eliminated</td>
</tr>
<tr>
<td>CVE-2014-2436</td>
<td>MySQL</td>
<td>Restore, Eliminated</td>
</tr>
</tbody>
</table>

Attack
- Denial of Service (DoS) | Apache | Reboot, Recovered |
- Low-rate Dos             | Apache | Reboot, Recovered |
Analysis and comparisons of rejuvenation and malware scanning for security

- Characteristics comparison

<table>
<thead>
<tr>
<th>Feature</th>
<th>Rejuvenation</th>
<th>Malware Scanning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault avoidance</td>
<td>Partial</td>
<td>No</td>
</tr>
<tr>
<td>Fault tolerance</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Denial of Service (DoS) or Low-rate DoS</td>
<td>Reboot</td>
<td>Log analysis</td>
</tr>
<tr>
<td>Virus elimination</td>
<td>Restore to checkpoint</td>
<td>Scanning</td>
</tr>
<tr>
<td>Trojan horse elimination</td>
<td>Restore to checkpoint</td>
<td>Scanning</td>
</tr>
<tr>
<td>Automated software-patching</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Intrusion detection</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Cost Model (CoMS)**

- Cost of malware scanning ($CoMS$)
  - Instance size ($V$)
  - Scan speed ($SS$)
  - Cloud computing fee ($CCF$)
  - Version storage fee ($SF$)
    - Ex: $0.095$ per GB-month
  - Data transfer fee ($TF$)
    - Ex: $0.12$ per GB

\[
CoMS = \frac{V \times CCF}{SS}
\]

- Ex: To scan 10 GB data of 80 GB instances with the speed of 26.58 MB/sec, the cost is $1.173.
CSCE 5760: Design or Fault Tolerance

Analysis and comparisons of rejuvenation and malware scanning for security

--- Cost Model (CoMS)

- Cost of rejuvenation ($CoR$)
  - Two types of rejuvenation
    - Periodic
    - On demand when an anomaly is detected (ad hoc) or prediction based
  - Downtime ($DT$)
    - Ex: 17 seconds in average
  - Number of transactions lost ($TL$)
    - Ex: Average 355.72/year
  - Potential revenue associated with each transaction ($PR$)
    - Ex: $100,000
  - Version storage fee ($SF$)
    - Ex: $0.095 per GB-month
  - Data transfer fee ($TF$)
    - Ex: $0.12 per GB
  - Monitoring cost ($CoM$)

4. Amazon.

---

\[
CoR_{\text{periodic}}(V,T) = DT \times TL \times PR + m \times V \times SF + V \times TF
\]

\(DT\) = downtime (during rejuvenation),
\(TL\) = lost transactions,
\(PR\) = per transaction loss,
\(m\) = number of checkpoints saved,
\(SF\) = Storage cost
\(TF\) = cost of transferring checkpoints to and from archival storage

- Ad hoc restoration happens when an anomaly is detected
- We will use the probability of an anomaly between periodic rejuvenations

\[
CoR_{\text{ad hoc}}(V,T) = \int_0^T f(t) dt = \left( CoM(V,T) + CoR_{\text{periodic}}(V,T) \right) dt
\]

- Ad hoc rejuvenations are warranted with a probability 10%.
- Monitoring consumes 0.1% of CPU time.
- \(CoR_{\text{ad hoc}}\) (10GB, 1hr) = $1.9161401 + $19.16 = $21.0761401
Analysis and comparisons of rejuvenation and malware scanning for security

--- Cost Model (CoMS)

- Assuming that on average $500,000 is the lost per hour of downtime.
- The cost of our experiment:

<table>
<thead>
<tr>
<th></th>
<th>System rejuvenation</th>
<th>Application rejuvenation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rejuvenation time</td>
<td>54 sec</td>
<td>187.51 sec</td>
</tr>
<tr>
<td>Storage</td>
<td>20 G</td>
<td>327 MB</td>
</tr>
<tr>
<td>Cost of computing</td>
<td>$7500</td>
<td>$26,043</td>
</tr>
<tr>
<td>Cost of image storage</td>
<td>$1.9</td>
<td>$0.12</td>
</tr>
<tr>
<td>Cost of data transfer</td>
<td>$2.4</td>
<td>$0.039</td>
</tr>
</tbody>
</table>

This is just an example, and the cut off point depends on real costs.
CSCE 5760: Design or Fault Tolerance

Analysis and comparisons of rejuvenation and malware scanning for security --- Cost Model (CoMS)

- # of potential undetected malware

<table>
<thead>
<tr>
<th>Day</th>
<th>Scanning method</th>
<th>Rejuvenation method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The signatures of malware are generated at 180th day.

- # of Malware in scanning method\(2\):

\[
S_d = N_d - D_d, 1 \leq d \leq 360
\]

\[
D_d = 0, d \text{ if } (N_d - D_d = 0) \text{ or } (d \text{ mod } 7) = 0
\]

- # of Malware in rejuvenation method (7day/time):

\[
R_d = N_d - D_d, 1 \leq d \leq 360
\]

\[
D_d = 0, d \text{ if } (N_d - D_d = 0) \text{ or } (d \text{ mod } 7) = 0
\]


CSCE 5760: Design or Fault Tolerance

We did not analyze the cost using prediction based rejuvenation

\(\rightarrow\) can be a topic for term project

Another method for software fault tolerance

\(\rightarrow\) data diversity

Think of dividing input space (domain values) into regions

inputs that do not cause errors

inputs that cause errors

Can we modify input – perturb inputs – to nudge the inputs away from error spaces

Again, practicality depends on the application domain

image processing or other statistical applications
Explicit vs implicit perturbation

Explicit - add a small deviation term to a selected subset of inputs
Implicit - gather inputs to program such that we can expect them to be slightly different

Example 1: software control of industrial process - inputs are pressure and temperate of boiler
Every second - \((p_i, t_i)\) measured - input to controller
Measurement in time \(i\) not much different from \(i-1\)

Implicit perturbation may consist of using \((p_{i-1}, t_{i-1})\) as an alternative to \((p_i, t_i)\)

If \((p_i, t_i)\) is in fault region - \((p_{i-1}, t_{i-1})\) may not be

Explicit perturbation

- Example 2: add floating-point numbers \(a, b, c\) - compute \(a+b\), and then add \(c\)
- \(a=2.2E+20, b=5, c=-2.2E+20\)
- Depending on precision used, \(a+b\) may be \(2.2E+20\) resulting in \(a+b+c=0\)
- Change order of inputs to \(a, c, b\) - then \(a+c=0\) and \(a+c+b=5\)

Software Implemented Hardware Fault Tolerance (SIHFT)

Can we overcome hardware failure with software support

*Data diversity and replication*

Think of transforming software to operate with original data and also data that is perturbed
Say for example, if we have software for a function \(f(a)\)
Write a new software that also implements \(f\), but uses input that is transformed – that is \(f(g(a))\)
Consider a simple transformation where the replicated software works on input that is multiplied by a constant $k$.

So the output of the replicated software should be $k$ times the results of original software.

The textbook shows how such an idea can detect stuck at faults.

If the $i$th bit is stuck at zero, and if data transmitted has $i$th bit = zero; an error.

But how do we detect this? Send data and $2^i \cdot d$ (shift).

If $i$th and $(i+1)$ bits are different, then you will notice an error.

---

What if $i$th and $(i+1)$ bits are both zeros?

Try $2^i \cdot d$, $4^i \cdot d$ etc.

May cause an overflow.

Use 2’s complement.

We can also use hardware for the same.

Think of designing an arithmetic unit.

The unit first performs the operation on original data.

Then the data is transformed (shifted or some other operation).

Perform the same arithmetic operation on transformed data.

---

N version programming.

N independent teams of programmers develop software.

Same specification or independent specifications.

different develop tools, algorithms etc.
If we assume failures are statistically independent, the probability that there are no more than \( m \) failures in \( N \) versions is given by

\[
Pr_{\text{ind}}(N, m, q) = \sum_{i=0}^{m} \binom{N}{i} q^i (1 - q)^{N-i}
\]

Comparing results of different versions
- Exact match
- Approximate match

An example from textbook

We have 3 versions computing some function of temperature and pressure of a system
- We compare the output of the function with a constant and take some action
  - \( f(t,p) > C \) do something (say turn furnace off)
  - otherwise add fuel

Say \( C = 1.0000 \); the three versions produce values 1.0001, 0.9999, 0.9998

What action to take?

Correlation between versions
- Need to make them independent
- If not, our reliability formula does not hold

For example we have 3 versions and failure rate of a version is \( q = 0.0001 \)

\[
q^3 + 3q^2(1 - q) \approx 3 \times 10^{-8}
\]

However, suppose 2 versions are correlated – that is if one of them fail both fail

Now we have basically 2 independent version

\[
q^2 + 2^*q^*(1-q) = \text{approximately } 10^{-6}
\]

If we cannot guarantee independence, then we need to change our formula for reliability to include conditional probabilities
CSCE 5760: Design or Fault Tolerance

Causes of correlation among N versions

- Common specification
- Intrinsically difficult problem
- Common algorithms
- Cultural factors
- Common software/hardware platforms

You can try to force independence

- Some experiments on how to do this
- Mostly university based projects
- Divide students into groups, isolate them, no discussion between groups

One study at the Universities of Virginia and California at Irvine

27 students wrote code for anti-missile application
Some had no prior industrial experience while others over ten years
All versions written in Pascal
93 correlated faults identified by standard statistical hypothesis-testing methods: if versions had been stochastically independent, we would expect no more than 5
No correlation observed between quality of programs produced and experience of programmer

To improve confidence in comparing outputs of N versions
may consider comparing intermediate values also

39

40