Rx: Treating Bugs As Allergies—A Safe Method to Survive Software Failures

FENG QIN
The Ohio State University
JOSEPH TUCEK and YUANYUAN ZHOU
University of Illinois at Urbana-Champaign
JAGADEESAN SUNDARESAN
Citadel Investment Group

ACM Transactions on Computer Systems, Vol. 25, No. 3, Article 7, Publication date: August 2007

Presented by Yeh Tsung-Yu

NCHU System & Network Lab
Outline

- Introduction
- Main idea of RX
- RX design
- Design and implementation issues
- Evaluation
- Conclusion and future work
Introduction

• A recent survey from around three hundred companies showed that
  – software defects account for 27% of system failures.
  – *Memory-related* bugs and *concurrency* bugs are common software defects
    • causing more than 60% of system vulnerabilities and accounting for 33–43% of the reported bugs in mature database and OS.
Introduction

• Previous work can be classified:
  – whole program restart, micro-rebooting of partial system components.
    • Handle transient failures and nondeterministic software bugs.
    • Can’t deal with deterministic software bugs.
    • Suffering from high overhead.
  – General checkpointing and recovery
    • Primary and backup. E.g. Nonstop architecture
    • Handle transient failures and nondeterministic software bugs
    • Also provide hardware fault-tolerance
    • Can’t deal with deterministic software bugs.
Introduction

• Previous work can be classified: (Const.)
  – Application-specific recovery mechanisms
    • Multi-process model, e.g. the old version of Apache
    • Spawn a new process for each client, and simply kill failed process and start a new one to handle unanswer request.
    • Can’t deal with deterministic software bugs. (E.g. a malicious request)
  – Failure-oblivious computing
    • Deal with buffer overflow by providing artificial values for out-of-bound reads.
    • But unsafe to use for correctness-critical applications.
Introduction

• Our idea is inspired by real life.
  – When a person suffers from an allergy, the most common treatment is
    • remove the allergens from *living environment*.
  – In software, many bugs resemble allergies. That is, their manifestation
    can be avoided by
    • *changing the execution environment*.

• Our contributions : RX
  – *Comprehensive*
    • Besides nondeterministic bugs, Rx can also survive deterministic bugs.
  – *Safe*
    • Rx does not speculatively “fix” bugs at run time, like previous work.
  – *Non-invasive*
    • Rx requires few to no modifications to applications’ source code.
  – *Informative*
    • Rx does not hide software bugs. Instead, bugs are still exposed.
Main idea of RX

- Upon a software failure, RX roll back the App to a recent checkpoint and re-execute it in a new environment that has been modified.

Fig. 1. Rx: The main idea
Main idea of RX

• Algo.
  – When re-execution still has symptom, RX try another environment to execute again.
  – When symptom is still detected with all those Env. we’ve tried.
    • RX roll back App to “older” checkpoint .
    • do the iteration we just did. (Env. changing)
  – When symptom is still detected with all the checkpoints.
    • RX use “whole program restart”.
  – If re-execution succeed, we can look up table 1 to identify bug and Env. change must be disable to reduce overhead.
# Table I. Possible Environmental Changes and Their Potentially Avoided Bugs

(DM means deterministic).

<table>
<thead>
<tr>
<th>Category</th>
<th>Environmental Changes</th>
<th>Potentially-Avoided Bugs</th>
<th>DM?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Management</td>
<td>delayed recycling of freed buffer</td>
<td>double free, dangling pointer</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>padding allocated memory blocks</td>
<td>heap buffer overflow</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>allocating memory in an alternate location</td>
<td>memory corruption</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>zero-filling newly allocated memory buffers</td>
<td>uninitialized read</td>
<td>YES</td>
</tr>
<tr>
<td>Timing-Related</td>
<td>scheduling</td>
<td>data race</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td>signal delivery</td>
<td>data race</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td>message reordering</td>
<td>data race</td>
<td>NO</td>
</tr>
<tr>
<td>User-Related</td>
<td>dropping user requests</td>
<td>bugs related to the user requests</td>
<td>Depends</td>
</tr>
</tbody>
</table>
RX design

Fig. 2. Rx architecture
RX design

- Sensor: there are two types of sensors.
  - Detect errors such as assertion failures, access violations, divide-by-zero exceptions.
    - Can be implemented by taking over OS-raised exceptions.
    - It’s adopted by RX because of low overhead.
  - Detect bugs such as buffer overflows, accesses to freed memory, before they cause the program to crash.
    - These method has lower detection latency.
    - These previous work has much more overhead.
    - It’s planed to be adopted by RX in the future work.
RX design

- **Checkpoint and Rollback**
  - Similar to the fork(), minimize overhead in a copy-on-write fashion.
  - Other system state
    - keeping a copy of accessed files and file pointers in the beginning of a CK interval and reinstate it for rollback.
    - To simplify it, leverage a *versioning file system* which automatically takes a file version upon modifications.

- Other state can be ignored (deterministic replay isn’t needed)
  - Rx introduces *nondeterminism* into server’s reexecution to avoid the bug.
  - Rx don’t need to remember
    - when an asynchronous event is delivered to the App in the first execution,
    - how shared memory accesses from multiple threads
RX design

• Checkpoint maintenance
  – Multiple CKs could impose a significant space overhead.
  – When every CK is taken,
    • RX compute time to roll back to oldest CK if it’s larger than the time T of restart whole program.
    • If the time is larger than T, oldest CK is deleted.
    • T can be measured by restarting immediately at the first software failure and then be used later.
RX design

• Environment Wrapper
  – Memory wrapper
    • implemented by intercepting memory-related library calls, such as `malloc()`, `realloc()`, `calloc()`, `free()`.
    • Facilities Provided, delaying free, padding buffers (fixed or variable size paddings), allocation isolation, zero-filling.
  – Message wrapper
    • Implemented in the proxy, provide transparent recovery for user and solve output commit problem.
    • Among different connections, it randomly shuffle the order of the requests.
    • But keep requests’ older within each connection for maintaining any possible dependency among them.
(a) Proxy behavior in normal mode

(b) Proxy behavior in recovery mode
RX design

- **Environment Wrapper (Const.)**
  - Process scheduling
    - Rx does this by changing the process’ priority
    - Thus scheduling time quantum is increasing, process is less likely to be switched off in the unprotected critical region.
  - Signal delivery
    - Rx record all signals in a table before delivering them.
    - For hardware interrupts
      - Delivers them at randomly time, but *preserving their order* to maintain any possible ordering semantics.
    - For software timer
      - Ignored because during rollback, the related software timer will also be restored.
RX design

• Environment Wrapper (Const.)
  – Dropping user requests
    • It is a last environmental change before switching to the whole program restart solution.
    • Percentage of dropped requests should be small (e.g., 10%)
    • Rx achieve this by performing a binary search on all recently received requests. (If upper bound is 10%, it takes five iterations of rollback and re-execution.)
RX design

- Control Unit
  - Most important job - Gradually builds up a failure table to capture the recovery experience for future reference

<table>
<thead>
<tr>
<th>Symptom</th>
<th>(1) padding</th>
<th>(2) scheduling</th>
<th>(3) drop reqs</th>
<th>(4) zero-filling</th>
</tr>
</thead>
<tbody>
<tr>
<td>assertion</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Segmentation fault</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Divide by zero exception</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Design and implementation issues

• **Inter-Server Communication**
  – For future work, RX should coordinate checkpoints in all the servers which communicate with each other.

• **Multi-threaded process checkpoint**
  – When checkpointing, some threads may be executing system calls or blocked inside the kernel.
Design and implementation issues

• Multi-threaded process checkpoint (Cont.)
  – Capturing the transient state of such threads could easily lead to state inconsistency upon rollback.
    \* E.g. there can be some kernel locks which have been acquired during checkpoint, and rolling back to such state may cause two processes hold the same kernel locks.
  – In RX, we sending signal to all threads when checkpointing, which make them exit from blocked system call.
  – After checkpoint, library wrapper retry those premature system call.
Evaluation

- **Experiment**
  - Two machine with P4 2.4G, and Ethernet connection with each other.
  - OS: linux kernel 2.6.10
  - Workload: MySQL, Squid, Apache, CVS, ypserv. (see Table 2)

<table>
<thead>
<tr>
<th>App</th>
<th>Ver</th>
<th>Bug</th>
<th>#LOC</th>
<th>App Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>4.1.1.a</td>
<td>data race</td>
<td>588K</td>
<td>a database server</td>
</tr>
<tr>
<td>Squid</td>
<td>2.3.s5</td>
<td>heap buffer overflow</td>
<td>93K</td>
<td>a Web proxy cache server</td>
</tr>
<tr>
<td>Squid-ui</td>
<td>2.3.s5</td>
<td>uninitialized read</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Squid-dp</td>
<td>2.3.s5</td>
<td>dangling pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>2.0.47</td>
<td>stack buffer overflow</td>
<td>283K</td>
<td>a Web server</td>
</tr>
<tr>
<td>CVS</td>
<td>1.11.4</td>
<td>double free</td>
<td>114K</td>
<td>a version control server</td>
</tr>
<tr>
<td>ypserv</td>
<td>2.12.1</td>
<td>memory leak</td>
<td>9.7K</td>
<td>a NIS server</td>
</tr>
</tbody>
</table>
## Evaluation

Table III. Overall Results: Comparison of Rx and Alternative Approaches, Including Whole Program Restart, and Simple Rollback and Reexecution without Environmental Changes (ENV-CHG means environmental changes, FSYM means failure symptoms, AL means alternatives, RS means restart, BO means heap buffer overflow, DR means data race, SO means stack buffer overflow, DF means double free, UR means uninitialized read, DP means dangling pointer, ML means memory leak).

<table>
<thead>
<tr>
<th>Apps</th>
<th>Bugs</th>
<th>FSYM</th>
<th>ENV-CHG</th>
<th>Clients Experience Failure?</th>
<th>Recoverable?</th>
<th>Average Recovery Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AL</td>
<td>Rx</td>
<td>AL</td>
</tr>
<tr>
<td>Squid</td>
<td>BO</td>
<td>SEGV</td>
<td>Padding</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MySQL</td>
<td>DR</td>
<td>SEGV</td>
<td>Scheduling</td>
<td>Yes</td>
<td>No</td>
<td>40%</td>
</tr>
<tr>
<td>Apache</td>
<td>SO</td>
<td>Assert</td>
<td>Drop Reqs</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CVS</td>
<td>DF</td>
<td>SEGV</td>
<td>Delay Free</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Squid-ui</td>
<td>UR</td>
<td>SEGV</td>
<td>Zero All</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Squid-dp</td>
<td>DP</td>
<td>SEGV</td>
<td>Delay Free</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ypserv</td>
<td>ML</td>
<td>SEGV</td>
<td>—**</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
(a) Throughput

(b) Average Response Time
Fig. 5. Throughput and average response time with different bug arrival rates.
Table IV. The Average Space Overhead per Checkpoint.

<table>
<thead>
<tr>
<th>Apps</th>
<th>Rx Space Overhead (kB/checkpoint)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>kernel</td>
</tr>
<tr>
<td>Squid</td>
<td>405.4</td>
</tr>
<tr>
<td>Mysql</td>
<td>300</td>
</tr>
<tr>
<td>Apache</td>
<td>460</td>
</tr>
<tr>
<td>CVS</td>
<td>42.2</td>
</tr>
</tbody>
</table>

Fig. 7. Rx recovery time to avoid the first and subsequent bug occurrences
Conclusion and future work

• Conclusion
  – It can deal with both deterministic and non-deterministic bugs, and increase software’s availability.

• Future work
  – Although the situation never arose during our experiments, it is possible that
    • A bug still occurs during re-execution but its symptoms are not detected in time by the sensors, then Rx’ll claim a false recovery success.
    • Currently an active research area with many recent innovations.