GfK An NIQ Company

FAU

The Fundamentals of Software Aging

Duke University, 2024-03-07

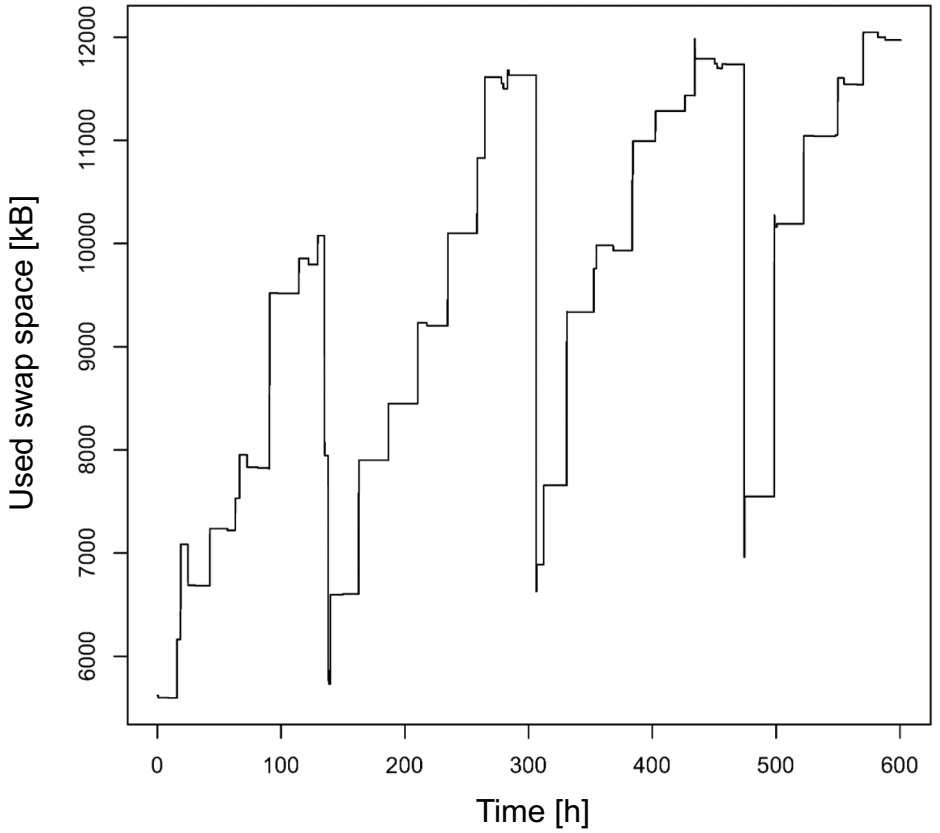
Michael Grottke
GfK and
Friedrich-Alexander-Universität Erlangen-Nürnberg

In our focus: “Fast“ software aging

- The “fast” software aging phenomenon relates to an increasing failure rate and/or a decreasing performance experienced with software systems which have been running continuously for a few days or even just for a few hours.
- It has also been observed for systems that do not use any live patching.
- The fast type of software aging was first studied analytically in 1995 by Huang et al., who referred to it as “process aging.”

Example 1: Apache Web server

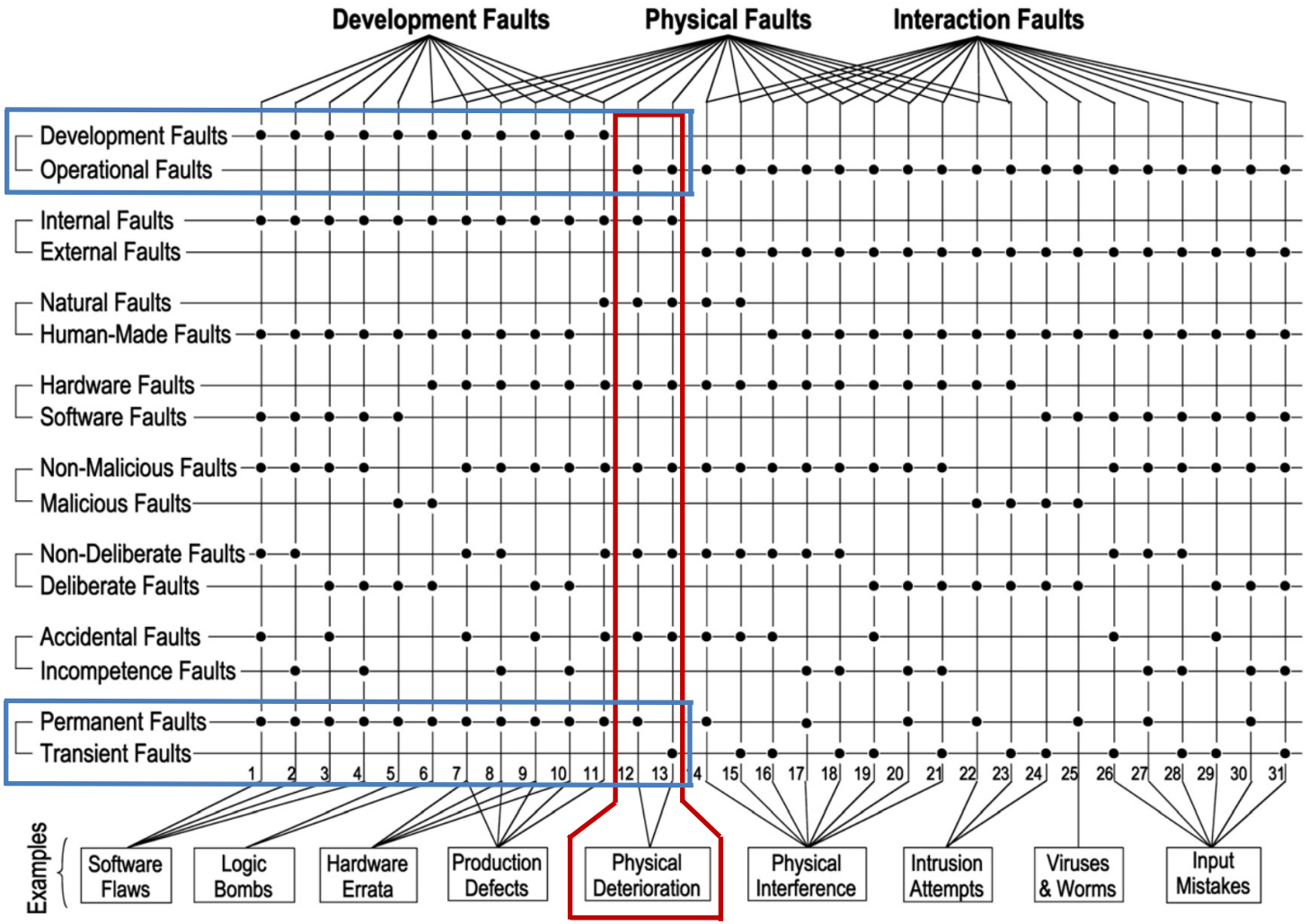
- A Web server running Apache 1.3.14 on a Linux platform was put in an overload condition by synthetic requests.
- For a period of over 3.5 weeks, 400 requests per second were generated.
- Used swap space showed both a statistically significant trend and a seasonal pattern repeating every seven days.
- Ultimately, the depletion of swap space would lead to performance degradation or a complete system failure.



Example 2: Patriot system

- The Patriot system is used for intercepting aircraft and missiles.
- To this end, once it detects an airborne object, the Patriot system calculates a range gate area in which an object of the type to be recognized is to be expected next. Only if the object should then be found in this area, the system fires.
- However, in the wake of updates made in 1988/1990, after the system had been continuously operating for about 20 hours its range gate area calculations were so imprecise that it looked for the target in the wrong place.
- On February 25, 1991, the Patriot system stationed in Dhahran failed to intercept an Iraqi Scud missile after operating continuously for more than 100 hours; 28 U.S. Army Reservists were killed, and 97 were wounded.

Matrix representation of fault types

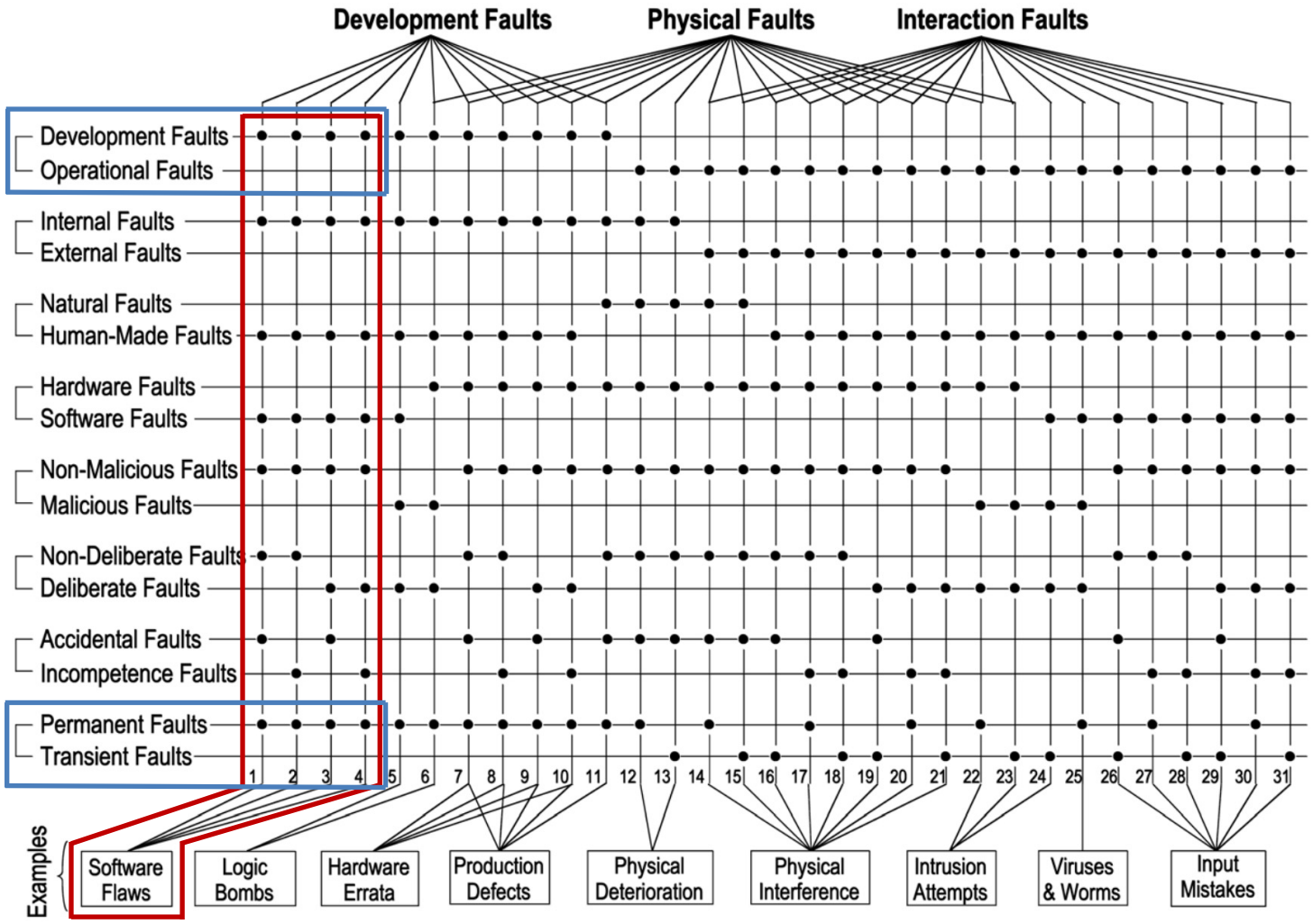


(A. Avižienis et al., 2004)

Natural aging

- Similar to physical deterioration, some software aging effects are a consequence of using the system/application over its lifetime.
- This kind of aging is thus referred to as “natural aging“.
- Examples: fragmentation problems experienced by file systems, database index files, and main physical memory.
- However, in many cases software aging is in fact due to development faults in the software (also referred to as “software flaws“ or “software bugs“).
- But how can a fault that is permanently contained in the software cause aging effects over time?

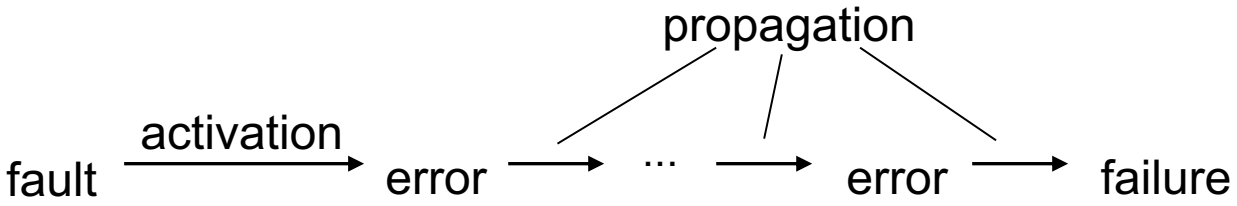
Matrix representation of fault types



(A. Avižienis et al., 2004)

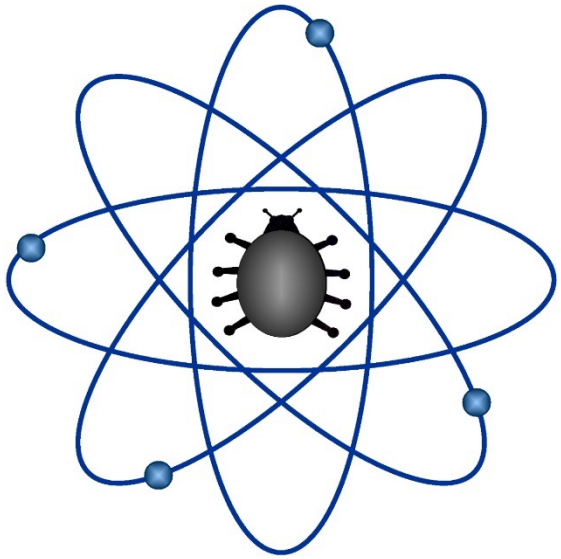
The “chain of threats“

- **Failure:** Deviation of the service delivered by a system from its specification; degraded performance constitutes a *partial* failure.
- **Error:** Part of the internal system state which may lead to a failure occurrence.
- **Error propagation:** Transformation of an error into other errors. A system failure occurs when an error is propagated to the service interface.
- **Fault:** Adjudged or hypothesized cause of an error.
- **Fault activation:** Application of an input to the component that causes the dormant fault to cause an error.



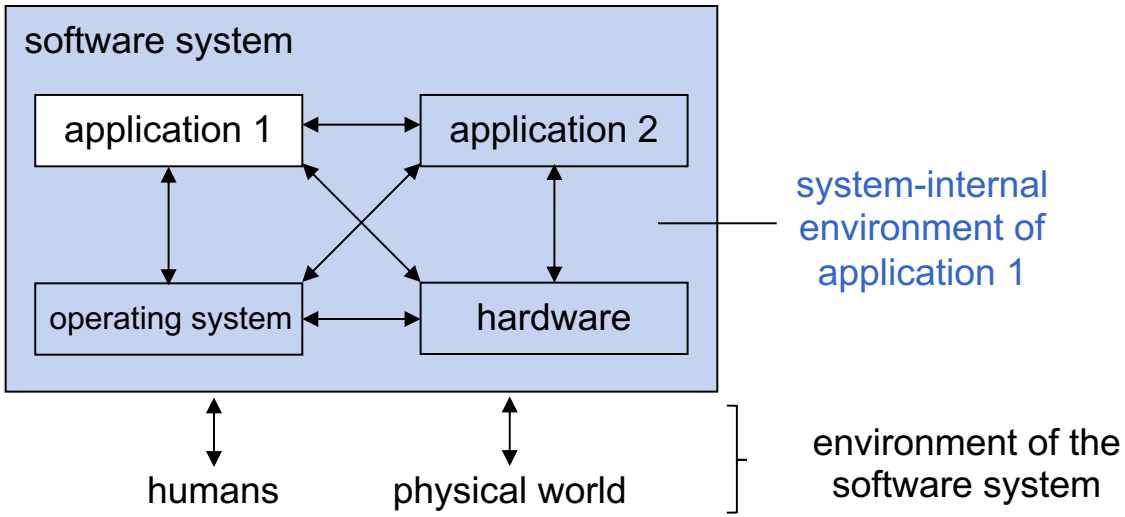
Bohrbug

- A Bohrbug is a fault whose activation and error propagation lack complexity.
- Typically, a Bohrbug is easily isolated, and it manifests consistently under a well-defined set of conditions.
- Example: A bug causing a failure whenever the user enters a negative date of birth
- The term alludes to the physicist Niels Bohr and his rather simple atomic model.



Complex fault activation/error propagation

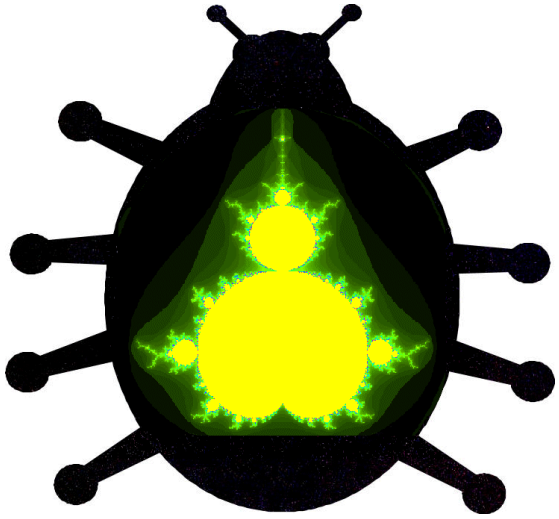
- First source of complexity: Fault activation and/or error propagation depend on the influence of indirect factors, such as
  - Timing/sequencing of inputs and/or operations,
  - System-internal environment of the application.



- Second source of complexity: Time lag between fault activation and failure occurrence, e.g., because several different error states have to be traversed in the error propagation.

Mandelbug

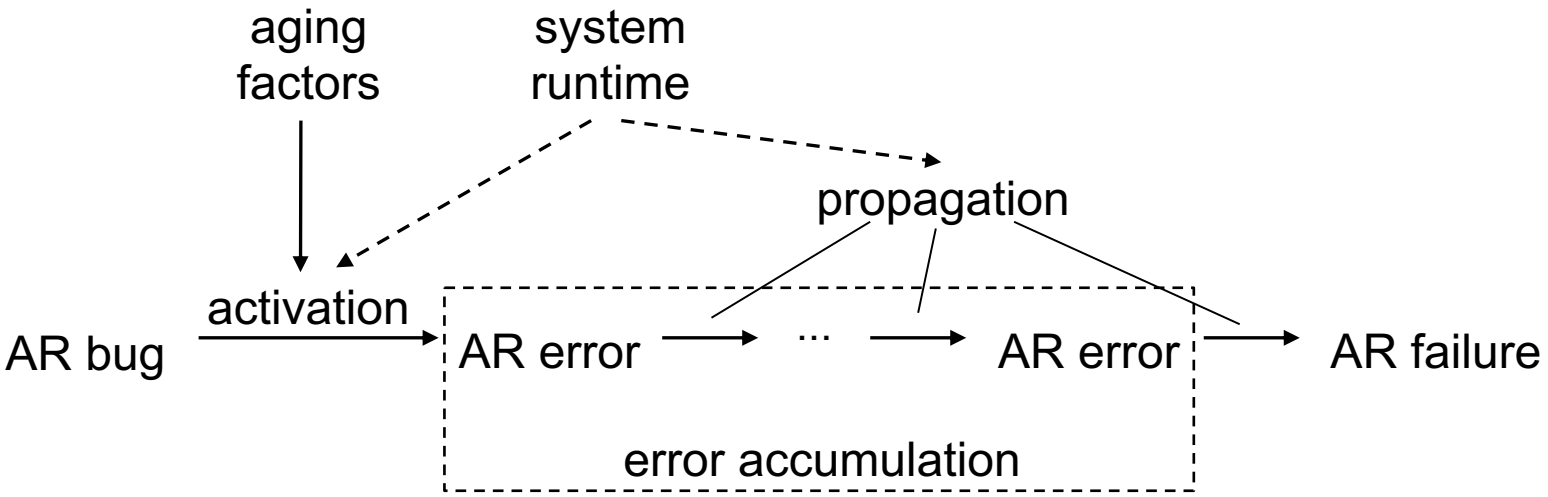
- A Mandelbug is a fault for which either
  1. The activation and/or error propagation is influenced by indirect, difficult to control factors, such as the timing or sequencing of inputs and operations, and aspects of the system-internal environment; or
  2. There is long time lag between fault activation and failure occurrence (e.g., because the error propagation involves several error states and/or subsystems).



- Due to its complex fault activation and/or error propagation, a Mandelbug is typically difficult to isolate, and the failures caused by it are often not systematically reproducible.
- The term alludes to the mathematician Benoît Mandelbrot and his research in fractal geometry.

“Chain of threats“ for an aging-related bug

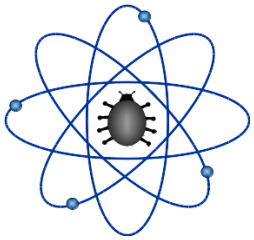
- For an **aging-related (AR) bug** (i.e., a software fault causing aging),
  - Either the errors caused by it accumulate inside the running system;
  - Or the activation or error propagation is influenced by the system runtime.
- **Aging factors:** Activation patterns of an AR bug
- We can distinguish between **internal** aging factors (e.g., function calls) and **external** ones (i.e., triggers related to the system environment).

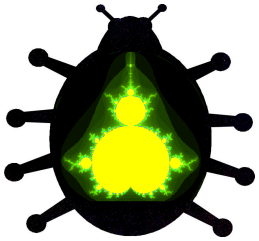


Aging-related bug

- An aging-related bug is a fault that is capable of causing an increasing failure rate and/or a degrading performance during continuous operation of the software.
- Two possibilities:
  - The fault causes the accumulation of errors either inside the running application or in its system-internal environment.
  - The rate of fault activation and/or error propagation is influenced by the total time for which the system has been continuously running.
- Aging-related bugs are Mandelbugs:
  - Either there is a delay between initial fault activation and failure occurrence, because errors first need to accumulate;
  - Or the rate of fault activation and/or error propagation is influenced by the total time for which the system has been continuously running.

Relationships between the bug types





Aging effects

- If the aging is due to AR error accumulation, the **aging effect** is the gradual shift from the correct internal state to a failure-probable one.
- Classes of aging effects:

Basic class	Extension	Examples
Resource leakage	(1) OS-specific (2) App-specific	- Unreleased • Memory (1, 2) • File handlers (1) • Sockets (1) - Unterminated • Processes (1) • Threads (1, 2)
Fragmentation	(1) OS-specific (2) App-specific	- Physical memory (1) - File system (1) - Database files (2)
Numerical error accrual	(1) OS-specific (2) App-specific	Round-off errors (1, 2)
Data corruption accrual	(1) OS-specific (2) App-specific	- File system (1) - Database files (2)

Volatile vs. non-volatile aging effects

- **Volatile aging effects:**
  - Removed by reinitializing the system or process affected.
  - Examples: unreleased memory in the OS, physical memory fragmentation

- **Non-volatile aging effects:**
  - Still exist after re-initializing the system or process affected.
  - Example: file system fragmentation

- Note: Hibernation etc. allow many intrinsically volatile aging effects to persist even after system/process reinitialization.

Pathology of an aging-related failure

Failure as perceived by the user:	File server does not respond
Failure as perceived by the troubleshooter:	Operating system halted
State of system-internal environment required for error propagation into a failure:	Insufficiency of main physical memory (availability of < 350 kB)
Error that leads to the failure:	Leaked memory inside the file server process
Aging effect:	Loss (leakage) of 100 kB per activation of the aging-related fault
Aging-related bug that causes the error:	A wrong value of the parameter used in the free() function in the comm.c program file
Location of the aging-related bug:	(x) Internal / ( ) External
Internal aging factor(s):	A call of the write_record() function
External aging factor(s):	The arrival of a packet carrying out the command SAVE_FILE
Failure mechanics: External aging factor => Internal aging factor => Fault activation => Error accumulation + Required state of system-internal environment => Failure	 [Packet Save File] => write_record() => {free(), Line_200} => Leakage of 100 kB + Availability of < 350 kB of main physical memory => System crash
Error accumulation scale:	1 : 1 : 1 : 100 kB

Software aging and software rejuvenation

- Note: We do not consider **every** performance degradation or **every** increasing failure rate to be caused by software aging.
- Counterexamples:
  - An increasing failure rate due to an increasing workload.
  - An increasing failure rate due to changes in the operational profile.
  - An increasing failure rate due to the queuing of jobs in an overloaded system.
- Important characteristic of software aging:
  The **aging effect** (e.g., the accumulation of internal error states) **is not reversible without external intervention**.
- The technique of proactively reducing the fault activation/error propagation rate, removing internal error conditions, and/or improving performance is called **software rejuvenation**.
- Examples of software rejuvenation:
  System reboots, application restarts, process restarts, garbage collection, flushing operating system kernel tables.

Aspects of software rejuvenation

- Advantages of software rejuvenation:
  - Avoids data loss etc. due to unexpected failures.
  - Can be scheduled conveniently (e.g., for middle of the night).
  - Can be employed if the location of the aging-related fault (or even its existence) is unknown.
- Disadvantage: Incurs costs (e.g., system downtime, higher load for other servers when rejuvenating one server).
- Consequence: Software rejuvenation requires good timing.
- Approaches to determining when to rejuvenate:
  1. Model-based approaches
     - Build analytical model of software aging and rejuvenation.
     - Solve model for metrics and optimal rejuvenation schedule.
  2. Measurement-based approaches
     - Periodically monitor **aging indicators** for **aging effects**.
     - Assess current “health” and trigger rejuvenation if necessary.

Aging indicators

- Aging indicators are explanatory variables that individually or in combination can suggest whether or not the system is healthy.

- **System-wide** aging indicators:
  - Provide information related to subsystems shared by several running applications (e.g., operating system, middleware).
  - Examples: free physical memory, used swap space, system load.

- **Application-specific** aging indicators:
  - Provide specific information about an individual application process.
  - Examples: resident set size of the process, response time.

Revisiting Example 1: Apache Web server

- It turned out that the seasonal pattern was due to the weekly log-rotation causing Apache to kill all of its child processes.
- As a consequence, swap space was released for two reasons:
  1. Swap space occupied by the Apache child processes, including unused memory not released earlier due to **memory leaks**, was released because these processes were killed.
  2. Swap space occupied by low-priority processes not executed but directed to the swap space due to the overload condition was released because these processes were able to execute and terminate as soon as resources were freed by killing the Apache child processes.
- The “external” intervention via the log-rotation **rejuvenated** the system, affecting the **system-wide aging indicator** ”used swap space”.
- Some of the used swap space (most of 2. and some 1.) might have been released by simply decreasing the **workload**.
- However, the increase only counteracted by the external action as well as the one not affected by it (trend!) can be considered **aging effects**.

Revisiting Example 2: Patriot system

- To calculate where to next expect the target, the Patriot system needs a) its velocity and b) the length of the time between subsequent checks.
- Dating back to the 1960s, the Patriot system counts the time since the last reboot (in tenths of seconds) as an integer in a 24-bit register.
- A **faulty** conversion into seconds resulted in an inaccuracy amounting to about 0.0001% of the *time span since the last system reboot* (**error**).
- This inaccuracy led to an imprecision in the calculated range gate area (**error propagation**).
- If the imprecision was small enough, then this error was not propagated into a failure; it became irrelevant, and **errors did not accumulate**.
- However, after 20 hours of continuous operation, the error was large enough to be **propagated into a failure**.
- Only **external intervention** (system reboot) could influence the factor permitting error propagation (**system runtime** kept in the 24-bit register).

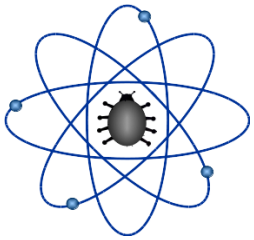
Not in our focus: “Slow“ software aging

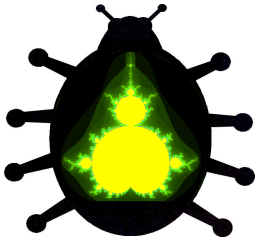
- There is a different type of software aging (first discussed in detail in 1994 by Parnas), which usually takes years or even decades to develop.
- Typical root cause: Changing user requirements and/or environment.
- Failure to account for this would render the software obsolete.
- Trying to adapt and extend the software leads to further problems:
  - Even the original programmers can hardly remember the design concept and the implementation details, especially if proper documentation is lacking.
  - Team composition may have changed substantially over the years.
  - Developers may have to deal with a legacy programming language and old-fashioned design patterns that they are hardly familiar with.
  - This poses the direct risk of introducing bugs when making changes.
  - Moreover, the lack of understanding as well as the usage of hacks results in an erosion of its design and architecture, which reduces the maintainability of the software and increases the risk of introducing faults in the future.

Revisiting Example 2 again: Patriot system

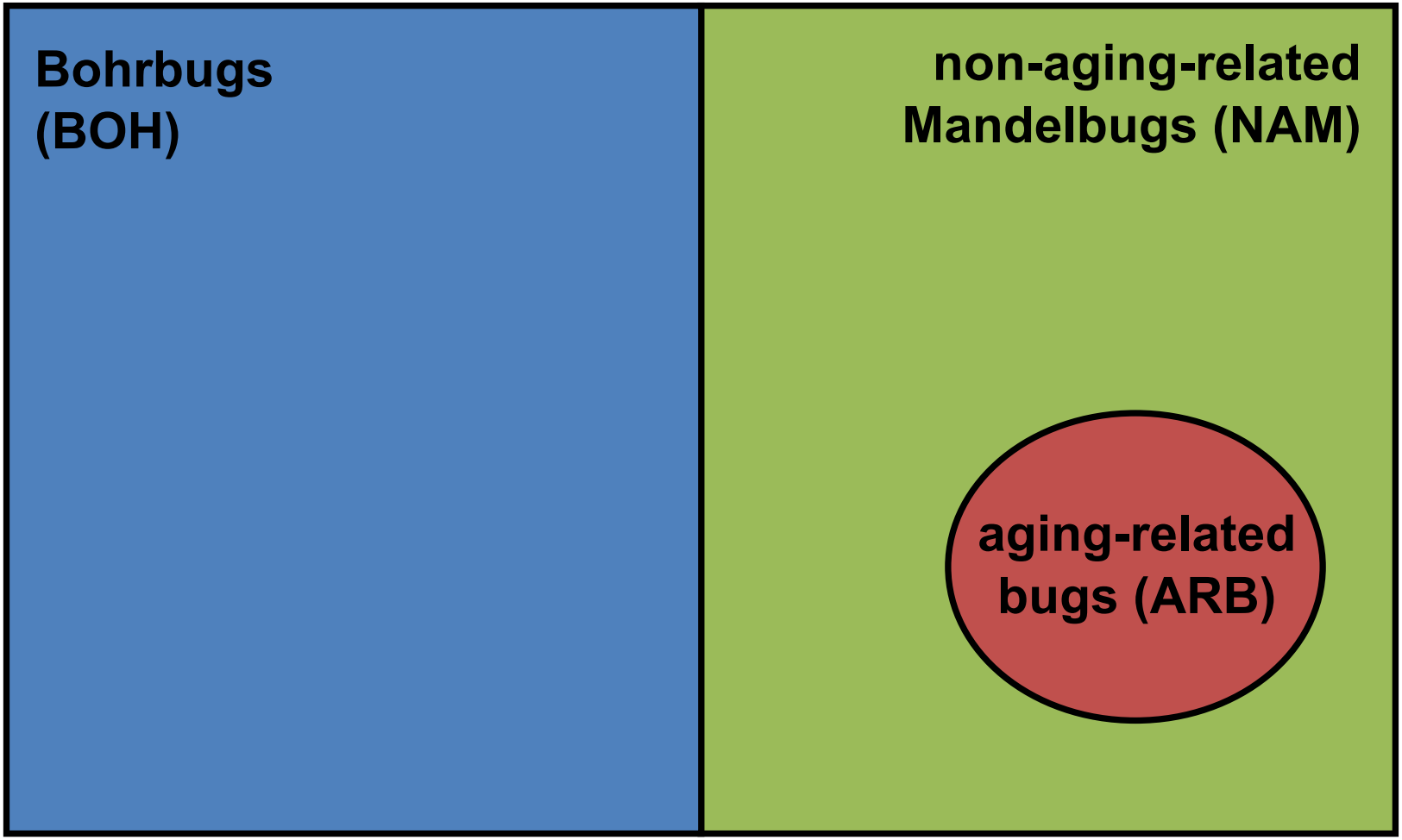
- Several big updates performed in 1988 and 1990 extended the Patriot system to also defend against tactical ballistic missiles.
- Until then, both the beginning and the end of the time interval between successive checks had been converted using a routine underestimating the respective number of seconds by about 0.0001%.
- Subtracting these two numbers resulted in an almost negligible 0.0001% error in the *computed length of the time interval*.
- When changing the 20-year-old assembler code, the developers only replaced the conversion of the end of the time interval with a call to a new routine with improved accuracy.
- As errors could not cancel out anymore, the inaccuracies in the calculated length of a time interval now amounted to 0.0001% of the *time since the last system reboot*.
- Slow software aging had thus given rise to fast software aging.

Relationships between the bug types





Relationships between the bug types



Faults in JPL/NASA flight software

(M. Grottke, A. P. Nikora, and K. S. Trivedi. An empirical investigation of fault types in space mission system software, 2010.)

- For 18 historic and ongoing JPL/NASA missions, we analyzed the anomaly reports recorded after deployment of the respective space system from the development facility.
- Information on these missions:
  - Seven missions were related to earth orbiters.
  - For seven missions, the destination was Mars.
  - Two missions were targeted to comets.
  - One mission returned samples of the Solar wind.
  - One mission was an Outer Planets mission.
- According to the multiple-choice “Cause” field, 653 anomalies were caused by the flight software.
- These anomalies were related to 520 unique faults in the flight software.

Faults in JPL/NASA flight software

Description of incident	Analysis, verification, and/or real time action	Final corrective action	Fault
Initial telemetry from on-board software Version XX indicated that Emergency Antenna was set to value B. This parameter in System Fault Protection should be value A. Uplink and Downlink communication at this point in the mission is not possible with value B. …	A real-time command was sent to the spacecraft to change the RAM configuration to value A. … Both Version YY and Version XX in on-board storage require an update to make the Emergency Antenna value A. On mm-dd-yyyy commands were sent to patch the FSW loads on both onboard storage devices.	All FSW copies now contain the correct Safing antenna value. Update of the Emergency Antenna parameter is being addressed by PFR. …	**BOH**

Faults in JPL/NASA flight software

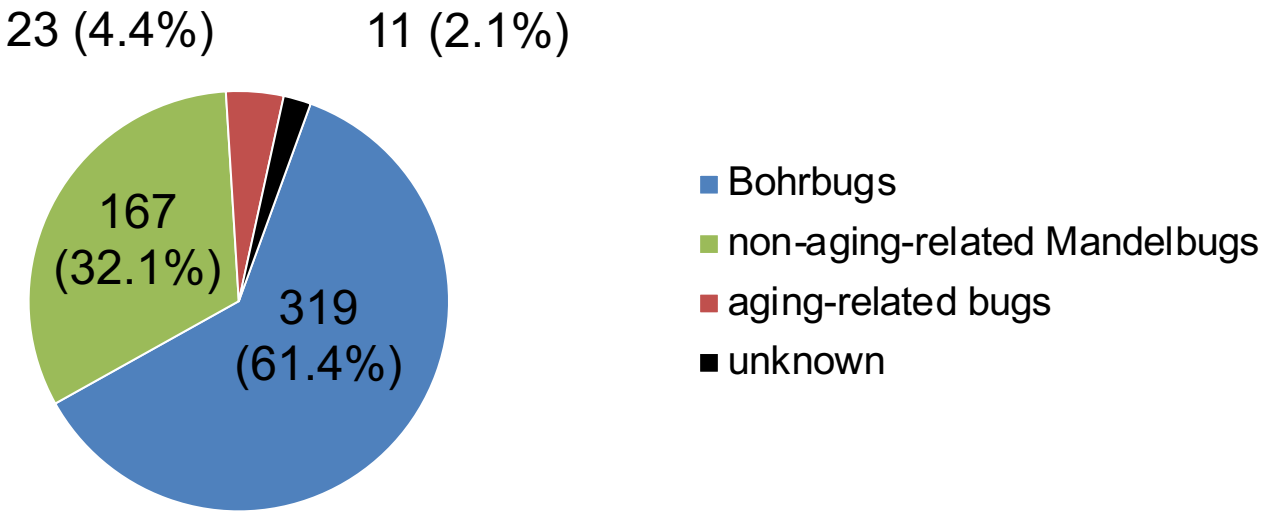
Description of Incident	Analysis, verification, and/or real time action	Final corrective action	Fault
During the attitude control FSW Checkout (YYYY-DOY/HH:MM:SS), telemetry channel reporting the **Number of Stars (NoS) incorrectly reported "X" star counts while the star identification (SID) activity was suspended. The NoS channel should always report 0 stars during periods of SID suspend**. This incident only occurred during the Yth SID Suspend test. **This happened once in flight, and once in a Type Z run with 200 suspends.**	See Corrective Action.	Change request W has been created to document this finding. This happened once in flight, and once in a Type Z run with 200 suspends. This problem occurs when SID SUSPEND or other SID commands cause SID to abort. **There is a possibility that the background task will overwrite the NoS channel after the foreground process updates it.** The change request was rejected by the Change Review Board held on mm-dd-yyyy, "because it is too late to make FSW changes." Low probability of occurrence. NO FIX.	**NAM**

Faults in JPL/NASA flight software

Description of Incident	Analysis, verification, and/or real time action	Final corrective action	Fault
The instrument team discovered that **after running for an extended period a stack pointer in instrument flight software becomes confused** and as a result writes a telemetry packet into flight software memory over code that is used on boot-up.	The instrument team developed a patch to correct the problem with instrument flight software. This patch was uplinked in real time within instrument commanding to flight software during the “A” and “B” sequences as to eliminate the risk.	The new version of instrument flight software fixes the algorithm so that the stack pointer no longer becomes confused and now correctly writes to the data buffer. The instrument team confirmed via testing and a memory dump that the memory overwrite malfunction is no longer occurring.	**ARB**

Checking Gray's conjecture

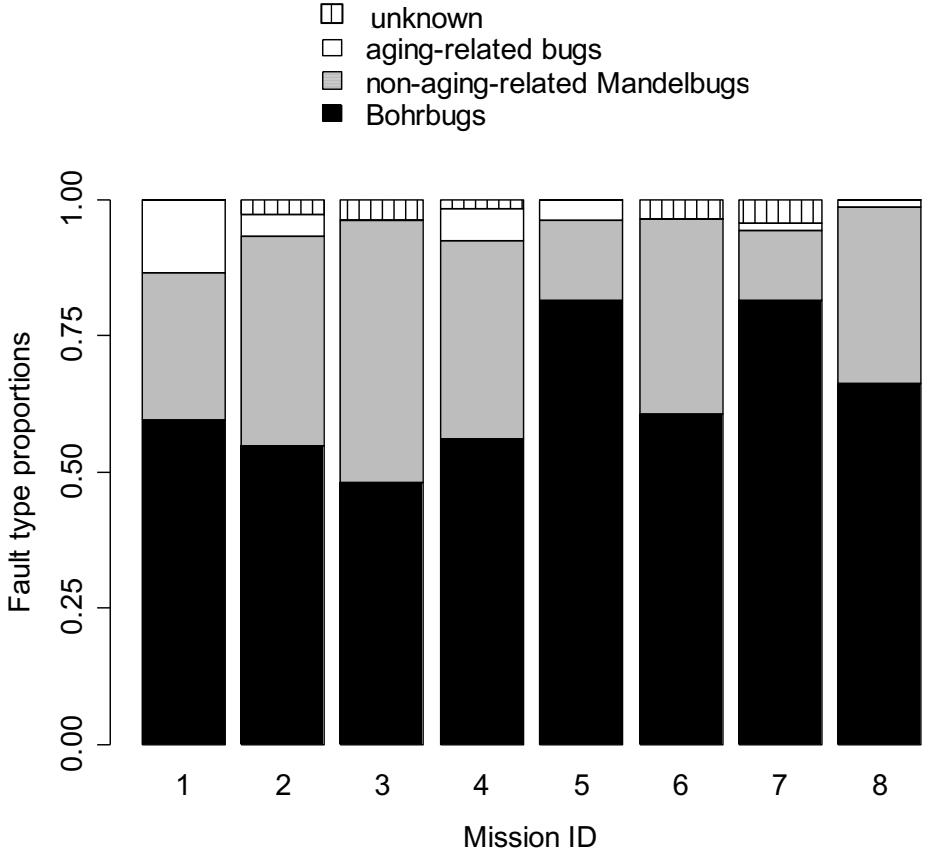
- "Most production software faults are soft. If the program state is reinitialized and the failed operation is retried, the operation will not fail a second time." (J. Gray, Why do computers stop and what can be done about it?, 1986.)
- Are the majority of JPL/NASA flight software faults Mandelbugs?



- For the flight software, Gray's conjecture does not seem to hold.

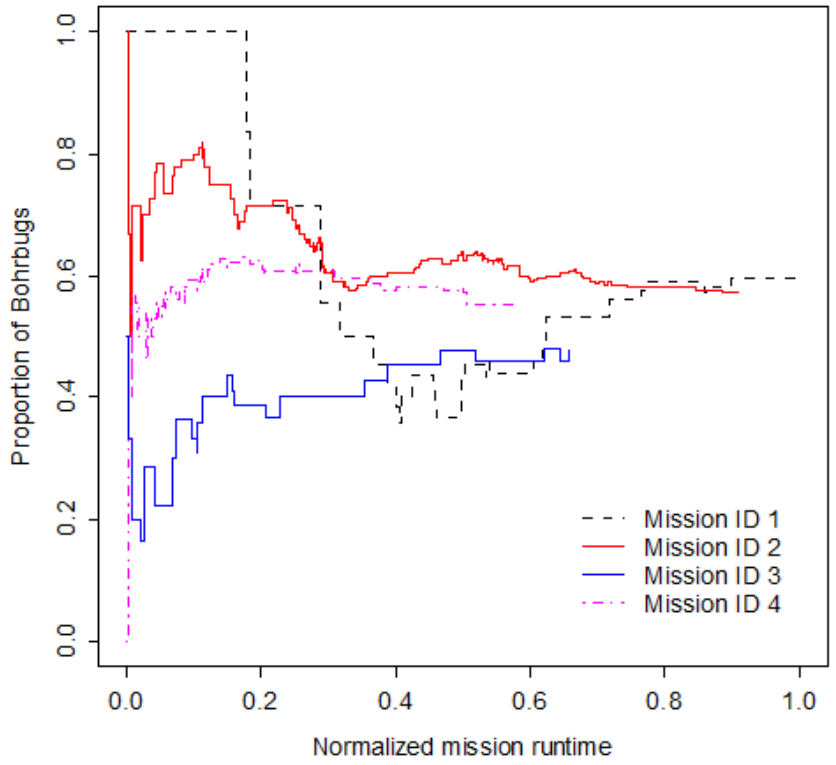
Fault type proportions per mission

- Are the fault type proportions similar across missions?
- Results (omitting ten missions with a total of 39 faults):



Can the differences be explained?

- How do the fault type proportions develop over runtime?
- Bohrbug proportions for the four earliest missions:



Open-source software

(D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi. Fault triggers in open-source software: An experience report, 2013.)

- Analyzed faults in four open-source software projects:

Project	#bugs	%BOH	%NAM	%ARB	%UNK
Linux	267	42.2	41.9	8.3	7.6
MySQL	209	56.6	30.3	7.7	5.4
Apache HTTPD	141	81.1	10.5	7.0	1.4
Apache Axis	199	92.5	3.5	4.0	0.0

- Gray‘s conjecture seems to hold for the Linux project only, but not for the other ones.
- Larger/more complex software seems to contain a higher proportion of Mandelbugs.

Fault types may not differ in severity…

Linux	BOH	MAN
Blocking	9	11
High	18	30
Low	7	4
Normal	88	100

MySQL	BOH	MAN
Critical	28	17
Serious	41	29
Non-Critic.	55	36
Perform.	1	2

Axis	BOH	MAN
Blocker	5	0
Closed	1	0
Critical	6	3
Major	78	6
Minor	15	0
Trivial	1	0

HTTPD	BOH	MAN
Blocker	4	0
Critical	16	4
Major	24	7
Minor	8	0
Normal	62	14
Trivial	2	0

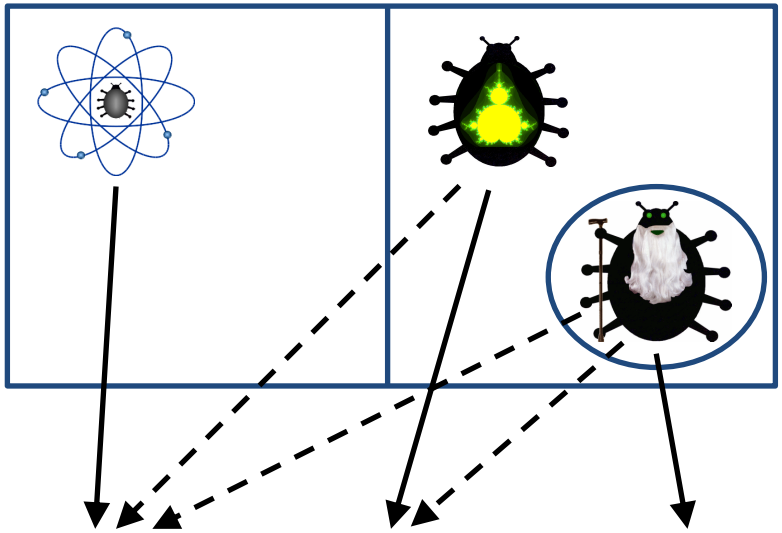
- Null hypothesis: The variables “fault type” and “severity” are independent.
- Fisher’s test of independence with Benjamini-Hochberg procedure for multiple comparison protection
- Confidence level: 90%
- Result: “not rejected” for all projects

… but their fixing times are different

Project	Time to fix: avg. (std. dev.) BOH	MAN
Linux	157.34 (226.21)	229.84 (304.24)
MySQL	107.92 (176.76)	89.45 (109.60)
Apache HTTPD	99.09 (199.44)	116.65 (133.84)
Apache Axis	111.19 (254.99	186.50 (256.18)

- Null hypothesis: For both types of bugs the time to fix is sampled from the same distribution.
- Wilcoxon rank-sum test with Benjamini-Hochberg procedure for multiple comparison protection
- Confidence level: 90%
- Result: In the three projects where a significant difference could be found, it takes *more* time to fix Mandelbugs.

Fault types and mitigation techniques



Fix/Patch	Retry	Rejuvenate
Workaround	Restart	
Use as is	Reboot	
Reconfigure	Failover to identical	
Failover to non-identical		

Conclusions

- “Fast” software aging relates to an increasing failure rate and/or a degrading performance during continuous operation.
- Sometimes, these phenomena are due to “natural aging”.
- Also, aging-related bugs can cause these phenomenona due to the accumulation of the errors caused by them or due to the influence of system runtime on their fault activation and/or error propagation.
- Aging-related bugs are a subset of Mandelbugs, which seemingly cause failures non-deterministically.
- While Gray’s conjecture often does not seem to hold, Mandelbugs represent a substantial share of faults in production software.
- Failures due to Mandelbugs can be recovered using inexpensive environmental diversity (restart, reboot, etc.).
- In particular, failures due to aging-related bugs can be avoided by regularly rejuvenating the system during operation.

References

- A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing, *IEEE Transactions on Dependable and Secure Computing* 1(1):11–33, 2004.
- D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi. Fault triggers in open-source software: An experience report. In *Proc. 24th IEEE International Symposium on Software Reliability Engineering*, pp. 178–187, 2013.
- J. Gray. Why do computers stop and what can be done about it? In *Proc. 5th Symposium on Reliability in Distributed and Database Systems*, pages 3–12, 1986.
- M. Grottke, L. Li, K. Vaidyanathan, and K. S. Trivedi. Analysis of software aging in a web server. *IEEE Transactions on Reliability* 55(3):411-420, 2006.
- M. Grottke, A. P. Nikora, and K. S. Trivedi. An empirical investigation of fault types in space mission system software. In *Proc. 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 447–456, 2010.
- M. Grottke, R. Matias Jr., and K. S. Trivedi. The fundamentals of software aging. In *Proc. 1st International Workshop on Software Aging and Rejuvenation/19th IEEE International Symposium on Software Reliability Engineering*, 2008.
- M. Grottke and K. S. Trivedi. Fighting bugs: Remove, retry, replicate, and rejuvenate. *IEEE Computer* 40(2): 107–109, 2007.
- M. Grottke and K. S. Trivedi. Aging, fast and slow. *IEEE Computer* 55(5):73–75, 2022.
- Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *Proc. 25th Symposium on Fault Tolerant Computing*, 1995, pages 381–390.
- D. L. Parnas. Software aging, In *Proc. 16th International Conference on Software Engineering*, 1994, pages 279–287.
- K. S. Trivedi, M. Grottke, and J. Alonso Lopez. Rethinking software fault tolerance. *IEEE Transactions on Reliability* 73(1):67–72, 2024.