EDITOR **PHIL LAPLANTE**
Penn State; plaplante@psu.edu

Aging, Fast and Slow

Michael Grottke, GfK SE

Kishor S. Trivedi, Duke University

Software can show symptoms of two different types of aging. Sometimes, it is even subject to both types.

In the early decades of software development, engineers were sure that software has one big advantage over hardware: it does not age. While the material used to build hardware components degrades over time, decreasing their reliability, this is not the case for software. Its code statements are either correct or incorrect, and a piece of software that works today will also do so in the future.

However, in the mid-1990s, two publications showed that software *can* age. Indeed, they described two different types of software aging: a fast one and a slow one. Although there has been plenty of research on this topic,[1,7] the notion that "software does not age" is still widespread. In this column, we review the two types of software aging, and we show that both of them played an important role in a well-known and lethal case of a software failure—a fact that has hardly been recognized so far.

FAST SOFTWARE AGING

The *fast* type of software aging was first studied analytically by Huang et al.,[5] who referred to it as "process aging." This phenomenon relates to an increasing failure rate or a decreasing performance experienced with software systems that have been running continuously for a few days or even for just a few hours. It has also been observed for systems that do not use any live patching. This begs the question: Why should the behavior of such a system change over time? After all, its code base remains unchanged, and the introduction of new faults can thus be excluded as an explanation.

To understand this type of software aging, we need to take a closer look at how the static faults in the software code can cause failures (that is, deviations of the dynamic software behavior from the one specified in the requirements). The typical consequence of the activation of a fault during software execution is an internal error state in the running system, for example, a wrong value of a variable kept in the random access memory. Such an error can be propagated into further errors (such as incorrect values of other variables) until it finally hits the front end and causes an incorrect result or any other kind of failure that can be noticed by the user.

Fast software aging is due to specific kinds of software faults, known as *aging-related bugs*.[4] In many cases, such faults cause errors that first need to accumulate inside the

Digital Object Identifier 10.1109/MC.2021.3133121
Date of current version: 6 May 2022

system before they can lead to a failure. Since the probability that a sufficient number of errors has accumulated becomes larger over time, the failure rate increases. Consider the example of a memory leak, that is, a software fault due to which memory that has been reserved and used is not released correctly. Such an individual error of unreleased memory in the running system will hardly cause any immediate problem. However, as increasingly more parts of the memory are affected, the system is likely to slow down until it finally fails because of an out-of-memory condition.

Yet, the accumulation of errors is not the only possible root cause of fast software aging. For some aging-related bugs, the rate at which they are activated or the rate at which the errors caused by them are propagated into failures depends on the system runtime.

In fact, one of the most famous cases of fast software aging falls into this latter category. On 25 February 1991, the Patriot missile defense system operating in Saudi Arabian Dhahran failed to recognize and intercept an Iraqi Scud missile. Twenty-eight U.S. soldiers were killed, and more than 90 were injured when this missile hit their barracks.[3]

The Patriot system worked as follows. After detecting an airborne object, it calculated a range gate area in which an object of the type to be recognized, in this case a Scud missile, was to be expected next. Only if the object should then be found in this area, would this confirm that it is indeed a Scud missile, and the Patriot system would fire. To compute the range gate area, the system used the known velocity of the potential target and the length of time between the last radar detection and the subsequent check. While the length of this time interval was required in seconds, the system internally counted the tenths of seconds that had passed since the last restart. Because of the 24-bit registers used, the conversion resulted in an error that represented about 0.0001% of the time span since the last system reboot (see p. 42 of Huckle and Neckel[6]).

During the first few hours after a restart, such an error was not propagated into a failure because the computed range gate area still contained the Scud missile to be intercepted. However, after a runtime of 20 h, the inaccuracy in the time interval amounted to about 0.0687 s.[3] A Scud missile traveling at Mach 5 (approximately 3,750 mi/h) covers more than 125 yards in this time span, which was enough to make the Patriot system look for the target in the wrong place and thus prevent it from correctly recognizing and intercepting the Scud missile. When the incident happened at Dhahran, the Patriot system located there had been running continuously for more than 100 h.[3]

SLOW SOFTWARE AGING

While the Patriot incident has often been cited as an example of fast software aging, the descriptions typically fail to point out that *slow* software aging played a role as well. This type of software aging, which usually takes years or even decades to develop, was first discussed in detail by Parnas.[8] Its root cause is often the fact that the user requirements for a piece of software as well as the environment in which it operates change over the years. Failure to account for these changes would render the software obsolete. However, if the developers decide to adapt and extend it in response to these changes, this may lead to further problems.

Years after a software was initially coded, even the original programmers can hardly remember the design concept and the implementation details, especially if proper documentation is lacking. If the team composition should have changed completely over the years, none of the original developers would still be around, and it is all the more difficult for new team members to understand the code. Moreover, they may have to deal with a legacy programming language and old-fashioned design patterns with which they are hardly familiar.

On the one hand, this poses the direct risk of introducing bugs when making changes. On the other hand, the lack of understanding as well as the usage of hacks when extending a software often results in an erosion of its design and architecture, such as violations of the "don't repeat yourself" principle (see p. 165 of Foote[2]), according to which the same functionality must not be implemented redundantly at several places in the code. In combination with the fact that the documentation might not be updated to (fully) reflect the changes, this significantly reduces the maintainability of the software while increasing the probability of involuntarily creating faults in the future.[8]

This is what happened with the Patriot missile defense system. It had originally been developed in the 1960s to intercept attacking aircraft.[3] Several big updates performed in 1988 and 1990 extended it to also defend against substantially faster tactical ballistic missiles (see pp. 40–41 of Huckle and Neckel[6]). One of these updates introduced a new routine using a pair of 24-bit registers to improve the accuracy in converting the tenths of seconds counted by the system into seconds. Until then, both the beginning and the end of the time interval during which the potential target was to be tracked had been converted using a routine underestimating the respective number of seconds by about 0.0001%. When these two numbers were subtracted, the inaccuracies partially canceled out, resulting in an almost negligible 0.0001% error in the computed length of the time interval.

Unfortunately, when changing the 20-year-old assembler code, the developers failed to replace all of the occurrences of the original conversion with a call to the new routine (see p. 43 of Huckle and Neckel[6]). Only the end of the time interval, but not its beginning, was now converted more accurately, and therefore errors could no longer cancel out. It was thus only after this update that the inaccuracies in the calculated length of a time interval amounted to 0.0001% of the time since the last system restart. Slow software aging had given rise to a system subject to fast software aging.

Interestingly, Parnas[8] did mention the possibility that slow software aging may lead to the introduction of memory leaks. However, it took one more year before Huang et al.[5] started the research stream investigating the various forms of fast software aging.

ACKNOWLEDGMENT

This work was supported by the Dr. Theo and Friedl Schoeller Research Center for Business and Society.

REFERENCES

1. T. Dohi, K. S. Trivedi, and A. Avritzer, Eds. *Handbook of Software Aging and Rejuvenation: Fundamentals, Methods, Applications, and Future Directions*. Hackensack, NJ, USA: World Scientific, 2020.
2. S. Foote, *Learning to Program*. Upper Saddle River, NJ, USA: Prentice Hall, 2014.
3. "Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia," General Accounting Office, Washington, DC, USA, Rep. no. GAO/IMTEC-92-26, 1992. [Online]. Available: https://www.gao.gov/assets/imtec-92-26.pdf
4. M. Grottke and K. S. Trivedi, "Fighting bugs: Remove, retry, replicate, and rejuvenate," *Computer*, vol. 40, no. 2, pp. 107–109, 2007, doi: 10.1109/MC.2007.55.
5. Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *Proc. 25th Symp. Fault Tolerant Comput.*, 1995, pp. 381–390, doi: 10.1109/FTCS.1995.466961.
6. T. Huckle and T. Neckel, *Bits and Bugs: A Scientific and Historical Review of Software Failures in Computational Science*, Philadelphia, PA, USA: SIAM, 2019.
7. C. Jones, "Geriatric issues of aging software," *CrossTalk*, vol. 20, no. 12, pp. 4–8, 2007.
8. D. L. Parnas, "Software aging," in *Proc. 16th Int. Conf. Softw. Eng.*, 1994, pp. 279–287, doi: 10.1109/ICSE.1994.296790.

MICHAEL GROTTKE is principal data scientist with GfK SE, Nürnberg, 90443, Germany, and an adjunct professor in the Department of Statistics and Econometrics at Friedrich-Alexander-Universität Erlangen-Nürnberg, Nürnberg, 90403, Germany. Contact him at michael.grottke@fau.de.

KISHOR S. TRIVEDI holds the Hudson Chair in the Department of Electrical and Computer Engineering at Duke University, Durham, North Carolina, 27708, USA. Contact him at ktrivedi@duke.edu.

Digital Object Identifier 10.1109/MC.2022.3166406