Root Cause Analysis of Software Aging in Critical Information Infrastructure

Philip König[1], Fabian Obermann[2], Kevin Mallinger[1,2] and Alexander Schatten[1]

[1] SBA Research pkoenig@sba-research.org, kmallinger@sba-research.org, aschatten@sba-research.org

[2] Vienna University of Technology fabian.obermann@tuwien.ac.at

Abstract. This paper examines the role of Software Aging and Rejuvenation and their effect on Critical Information Infrastructure and thus Critical Infrastructure maintainability. Software systems tend to degrade over time by entering a failure-prone state and showing decreased performance. It is suggested that Critical Infrastructures are especially susceptible to the detrimental effects of Software Aging and that common Software Rejuvenation remedies are not suitable in this context. Instead of treating re-emerging symptoms, an alternative approach is presented that seeks to monitor, analyze and, identify potential root causes like underlying architectural problems of software used in Critical Infrastructure. Results of first applications are shown and intended next research and development steps discussed.

Keywords: Software Aging · Critical Information Infrastructure · Maintainabilty · Code Change Analysis

1 Introduction

Critical Infrastructures (CI) span a wide range of sectors, among others, power grids, communication, defense, or finance systems [12]. Critical means that in a case of their incapacitation, corruption, or interference of any sort, the following consequences could be catastrophic for economic and/or social welfare, and national security. In short, CI constitute the foundation on which modern living standards are built. Nowadays these infrastructures are increasingly dependent on information and communication technology [13]. These include among other things software and the hardware it is running on, which in this context are called Critical Information Infrastructure (CII) and thus become a crucial part of CI themselves [8, 9]. This poses a problem as CIIs often show significant security weaknesses and operational problems. Competition between contractors or generally the pressure for a faster time to market interval and new functionality, leads them to offering ever faster development times for their product and to integrate new features on a regular basis [13]. The consequences include neglecting long-term security, stability, maintainability or introduce architectural problems which could compromise modularity of the code and thus impede its own replacement in the future should it become outdated. Such practices may be unnoticed in the short run but can lead to critical points of failure in the future.

2 Software Aging and Rejuvenation

To monitor and deal with changes that software might undergo over time, the field of *Software Aging and Rejuvenation* (SAR) emerged. This research field proposes that software degrades over time or through processing demand [2, 4, 14, 15]. Avizienis et al. [1] identified causes, such as: memory bloat and leakage, unterminated threads, unreleased file locks, data corruption, storage space fragmentation, and accumulation of rounding errors. These phenomena gradually proliferate unnoticed until a critical tipping point is reached, which then leads to failures like increased response time or no service at all [7]. A dramatic example of what can happen when this occurs in critical systems is the 1991 case of a Patriot air defense system malfunctioning and causing the death of 28 servicemen. [11] Patriot's function was to intercept incoming missiles through calculating their flight path. The way in which it's software was written introduced a time lag depending on the running time of the system, which after 100 hours of continuous operation built up to 0.3433 seconds. This resulted in shifting it's scanning area by 687 meters and thus missing an incoming missile. Many of such malfunctions can be mitigated by doing a simple restart of the system, hence these problems seldomly reach that tipping point in systems that are only operated intermittently, like private PCs. CII that is implemented in systems that must continuously be operable – like digital payment providers, emergency services, or power grid management systems – cannot easily use simple restart mitigation measures to enhance functionality [9, 15].

This highlights a problem that Software Rejuvenation, the umbrella term for methods dealing with such software aging related bugs, has. The common methods to remediate the effects of Software Aging include stopping running software, cleaning internal states, and restarting the system to restore it to a known, less failure-prone state [2, 5]. This is precisely what many CII systems are, if at all, only able to do under great expenditures [15]. Further, even if such treatments are successful and re-establish the original failure-free state, they are generally not suited to prevent the resurgence of these failures in the future.

3 Goals

Avizienis et al. [1] stated that such failures stem from the software itself, more precisely the way it is devised. Until rectified, they are permanently written into the source code of a respective program. Therefore, it is impossible to fix them by mentioned techniques of Software Rejuvenation, the purpose of which is more akin to treating symptoms and not addressing the underlying problems that cause them. Lutz [10] came to a similar conclusion, emphasizing that higher code complexity, and thus potential interdependence, in safety critical, embedded systems is a potential safety hazard and must be addressed early on in development. In that vein, Graves et al. [6] devised a fault prediction model on the assumption that code decay occurs when a system becomes so complex that each change to it introduced on average one fault and thus is deemed unstable. Inspired by this line of work, this paper postulates that common methods of Software Rejuvenation need to be supplemented by root cause analysis. To achieve this we derived three different indicators to help assess its maintenance quality: magnitude, quantity and clustering of code changes over time. Considering the example of well implemented modularity, in most cases there should be very little to no required manual changes in other, functionally unrelated, parts of the code when a certain module is modified. In contrast, when this is not done appropriately, modifications could for example manifest as a rat-tail of needed changes in modules that were not part of the original intended one.

Although, one must be cautious to not over-interpret clusters of change in modules in a singular commit. Only when coupled changes occur continuously over time, they become an indicator for further inspection. Such changes would be visible in the history of a given version control system and aggregate at the respective points in time. Complex dependencies of this nature could make future maintenance or replacement of modules very hard or even impossible and thus pose a risk factor if a vulnerability or error is detected in such a module. For this purpose, a proof-of-concept code change analysis methodology was developed, which is able to identify changes in the nature of the mentioned indicators.

4 Methods

Our methodology was conceptualized in a two-step pipeline. In the first step the target software is analyzed and multiple datasets for further processing are generated. In a second step a report from the used data is generated and visualized. A programs git history is used to create the datasets. The target application can be analyzed on three levels of detail: Files, Classes, and, Functions, which will be called scopes from here on. However, Class and Function support is at the moment only implemented for target applications written in Java and C#, in which Classes and File scopes are mostly identical. Changes are clustered by date to allow visualization of how often and intensely a scope changed and to allow the location of change hotspots. The magnitude of a change is calculated as percentual change of the lines modified/added/deleted in comparison of the total lines of the given scope before the change. The application also counts how often each scope was modified in combination with each other scope. The top results are tabled in the report, as these scopes highly depend on each other. For this specific use case, the GUI package of the *checkstyle* repository [3], an open-source java application, was used.

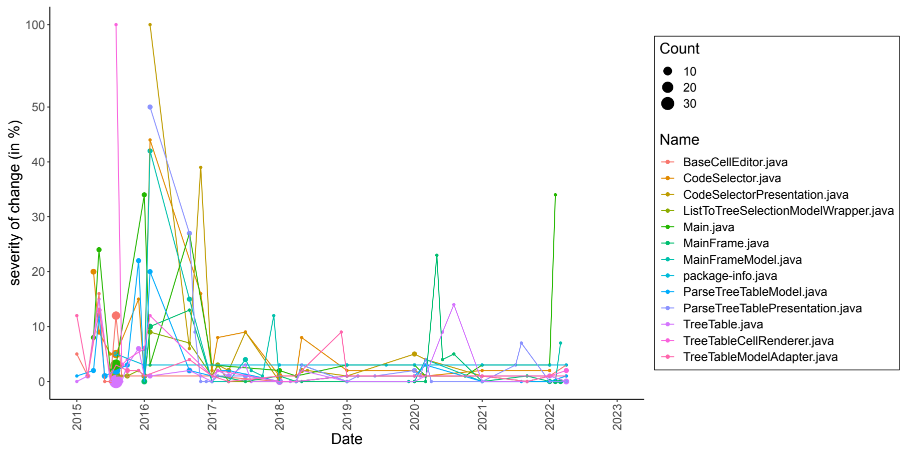

Fig. 1. Number of code changes and their magnitude over time. Quantity of changes represented by the diameter of the corresponding sphere, severity and thus magnitude by its height on y-axis. Code change cluster visualized through grouping of spheres.

5 Results

In Fig. (1) the x-Axis represents the lifetime of the repository in monthly granularity, while the y-Axis shows the magnitude of the change in the given month measured by percent. For multiple changes in one point of time the median was used. The diameter of the spheres corresponds to the number and thus quantity of changes in the given period and the color differentiates between individual files.

The life cycle of single files can be observed, as their corresponding first data point is their creation date and the last one the date of the last update. It can be seen that in recent years, activity increased with small intervals between the data points. Also, most of the code was written or refactored in 2015 and 2016. The granularity in this time period could be further improved by filtering out commits which changed almost every file, as they are most likely refactoring commits that did not alter noncommentary code. Regarding the dates with the most changes, a pattern becomes visible: they tend to cluster at the beginning of the year. The reason being that at the beginning of each year, the copyright headers in all files are adapted by changing the date to the new year. The only other commits that touched nearly every file were the mentioned huge Javadoc refactorings as can be seen in the periods from 2015 to 2016 and 2016 to 2017. Excluding the refactoring commits, the data still shows that commits generally are big, as most of them touch between 4 and 8 of a total of 14 files, potentially indicating that the components might be interdependent. In some cases, this is not necessarily a bad thing, as for example all TreeTable classes are usually modified together because their functionality was extended.

6 Discussion and further research

By using this method, it is possible to extract and visualize when, to what extent, and on which scope, code was changed. When displaying multiple items together over time, it can be observed if and when those were changed simultaneously like in the case of the TreeTable classes. This implicates that this approach is suited to monitor how a project or codebase changes over time and thus provide information about the interdependence of the individual scopes. Although the developed software is fully functional and can be used on real world applications like checkstyle, there are still some manual steps in the setup as well as in the evaluation process. To increase the quality of the output, better metrics for the magnitude of the change could be implemented by not only looking at the code at a line-by-line level, but also consider how much changed in a line. Also changes that consist of only whitespace characters could be ignored. A point that was not considered in this first implementation are comments. In the repositories we analyzed, a lot of changes often came from Javadocs or other general comments, which most likely should not be considered code change in this context. Although Graves et al. [6] mentioned that swapping out lines of code for noncommentary lines of code did not impair the performance of their fault prediction model, so there could be interesting information in doing multiple analyses, with and without noncommentary code and compare those. On top of that, deleted files are currently ignored by this methodology, but might provide valuable data if one wants to analyse how a program changed over time. Future research will also target the generation of an interactive html-based report. This will allow easier evaluation, enabling the generation of drill down charts, and jumping directly into the source code. Such data then could be linked to the history of bug reporting tools or failure data. Not only would a combined analysis of commit and failure history, directly linked to the source code, provide valuable insight in the emergence of failures, in a reverse manner it could also be analyzed which changes to the code were needed to eliminate it.

Conclusion

The continuous embedding of information technologies into critical infrastructures leads to increasingly unpredictable and complex systems. As these components are often developed without proper long-term maintenance or replacement strategies in mind, they introduce security vulnerabilities into all systems they are in contact with. These software systems degrade and age over time and established procedures to counter-act the most common detrimental effects are often not well suited for the kind of systems employed in CI. A supplemental methodology on the basis of three indicators of code change was presented that is not only able to monitor quantity and magnitude of changes in code over time but also track if certain items are periodically changed in tandem. Applied to large codebases or projects such coupled changes could indicate dependencies or modularity flaws and thus help identify potential risks that concern each associated system.

References

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE transactions on dependable and secure computing* 1.1 (2004), pp. 11–33.

[2] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert. "Proactive management of software aging". In: *IBM Journal of Research and Development* 45.2 (2001), pp. 311–332.

[3] *Checkstyle git repository.* `https://github.com/checkstyle/checkstyle/tree/master\\/src/main/java/com/puppycrawl/tools/checkstyle/gui`. Accessed: 2022-04-27.

[4] D. Cotroneo, R. Natella, and R. Pietrantuono. "Predicting aging-related bugs using software complexity metrics". In: *Performance Evaluation* 70.3 (2013), pp. 163–178.

[5] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo. "A survey of software aging and rejuvenation studies". In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 10.1 (2014), pp. 1–34.

[6] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. "Predicting fault incidence using software change history". In: *IEEE Transactions on software engineering* 26.7 (2000), pp. 653–661.

[7] M. Grottke, R. Matias, and K. S. Trivedi. "The fundamentals of software aging". In: *2008 IEEE International conference on software reliability engineering workshops (ISSRE Wksp).* Ieee. 2008, pp. 1–6.

[8] A. Kalashnikov and E. Sakrutina. ""Safety management system" and Significant Plants of Critical Information Infrastructure". In: *IFAC-PapersOnLine* 52.13 (2019), pp. 1391–1396.

[9] J. Lopez, R. Setola, and S. D. Wolthusen. "Overview of critical information infrastructure protection". In: *Critical Infrastructure Protection.* Springer, 2012, pp. 1–14.

[10] R. R. Lutz. "Analyzing software requirements errors in safety-critical, embedded systems". In: *[1993] Proceedings of the IEEE International Symposium on Requirements Engineering.* IEEE. 1993, pp. 126–133.

[11] E. Marshall. "Fatal error: how Patriot overlooked a Scud". In: *Science* 255.5050 (1992), pp. 1347–1347.

[12] J. Moteff, C. Copeland, and J. Fischer. "Critical infrastructures: What makes an infrastructure critical?" In: Library of Congress Washington DC Congressional Research Service. 2003.

[13] E. Nickolov. "Critical information infrastructure protection: analysis, evaluation and expectations". In: *Information and Security* 17 (2006), p. 105.

[14] D. L. Parnas. "Software aging". In: *Proceedings of 16th International Conference on Software Engineering.* IEEE. 1994, pp. 279–287.

[15] M. E. Sabino, M. Merabti, D. Llewellyn-Jones, and F. Bouhafs. "Detecting software aging in safety-critical infrastuctures". In: *2013 Science and Information Conference.* IEEE. 2013, pp. 78–85.