Please hold on: more time = more patches? Automated program repair as anytime algorithms

Authors:

Duc Ly Vu, University of Trento (IT)
Ivan Pashchenko, University of Trento (IT)
Fabio Massacci, University of Trento (IT), Vrije Universiteit Amsterdam (NL)

This paper was written within the H2020 AssureMOSS project that received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 952647. This paper reflects only the author's view and the Commission is not responsible for any use that may be made of the information contained therein.

Assurance and certification in secure Multi-party Open Software and Services (AssureMOSS) No single company does master its own national, in-house software. Software is mostly assembled from "the internet" and more than half come from Open Source Software repositories (some in Europe, most elsewhere). Security & privacy assurance, verification and certification techniques designed for large, slow and controlled updates, must now cope with small, continuous changes in weeks, happening in sub-components and decided by third party developers one did not even know they existed. AssureMOSS proposes to switch from process-based to artefact-based security evaluation by supporting all phases of the continuous software lifecycle (Design, Develop, Deploy, Evaluate and back) and their artefacts (Models, Source code, Container images, Services). The key idea is to support mechanisms for lightweigth and scalable screenings applicable automatically to the entire population of software components by Machine intelligent identification of security issues, Sound analysis and verification of changes, Business insight by risk analysis and security evaluation. This approach supports fast-paced development of better software by a new notion: continuous (re)certification. The project will generate also benchmark datasets with thousands of vulnerabilities. AssureMOSS: **Open Source Software: Designed Everywhere, Secured in Europe**. More information at **https://assuremoss.eu**.

Duc Ly Vu is currently a Ph.D. student at the University of Trento, Italy. He was an Early Stage Researcher at the European Network for Cybersecurity project from 2017 to 2019. As part of the AssureMOSS project, his primary research focuses on automated program repair and mining source code repositories. Contact him at *ducly.vu@unitn.it*.

Ivan Pashchenko (PhD 2019) is a Research Assistant Professor at the University of Trento, Italy. He was awarded a silver medal at the ACM/Microsoft Student Graduate Research Competition at ESEC/FSE. He is UniTrento main contact in *"Continuous analysis and correction of secure code"* work package for the AssureMOSS project. Contact him at *ivan.pashchenko@unitn.it*.

Fabio Massacci (Phd 1997) is a professor at the University of Trento, Italy, and Vrije Universiteit Amsterdam, The Netherlands. He received the Ten Years Most Influential Paper award by the IEEE Requirements Engineering Conference in 2015. He is the the European Coordinator of the AssureMOSS project. Contact him at *fabio.massacci@ieee.org*.

How to cite this paper:

- Vu, D.L. and Pashchenko, I. and Massacci, F. *Proceedings of ACM/IEEE International Conference on Software Engineering - Automated Program Repair (APR) workshop (ICSE-APR 2021)*. IEEE Press.

Please hold on: more time = more patches? Automated program repair as anytime algorithms

Duc-Ly Vu
University of Trento (IT)
ducly.vu@unitn.it

Ivan Pashchenko
University of Trento (IT)
ivan.pashchenko@unitn.it

Fabio Massacci
University of Trento (IT), Vrije Universiteit Amsterdam (NL)
fabio.massacci@ieee.org

***Abstract*—Current evaluations of automatic program repair (APR) techniques focus on tools' effectiveness, while little is known about the practical aspects of using APR tools, such as how long one should wait for a tool to generate a bug fix. In this work, we empirically study whether APR tools are any time algorithms (e.g., the more time they have, the more fixes they generate, so it makes sense to trade off longer time for better quality). Our preliminary experiment shows that the amount of plausible patches, given exponentially greater time, only increases linearly or not at all.**

***Index Terms*—Automated Program Repair, Anytime Algorithms, Empirical Software Engineering**

I. Introduction

Automated program repair (APR) techniques were designed to automatically search for (i.e., generate) candidate patches for software bugs that might be plausible ones (passing all tests) and then possibly correct ones (actually fixing the bug). An important class of search-based techniques is that of *anytime algorithms* [1] which tries to capture an important trade-off:

> Anytime algorithms give intelligent systems the capability to trade deliberation time for quality of results. This capability is essential [when] it is not feasible (computationally) or desirable (economically) to compute the optimal answer [...].

Our long-term **Goal** is to understand whether APR techniques are anytime algorithms so that it makes sense to wait longer to obtain better results. Empirically, Durieux et al. [2] studied 11 APR tools but only showed that most of the repair attempts resulted in an error or terminated with a timeout. By constraining the search space, Qi et al. [3] showed that the number of patch candidates can vary significantly, which might impact the number of plausible patches. In this direction, Martinez and Monperrus [4] showed that an additional time budget might result in a higher number of plausible patches. The authors did not investigate *how long one needs to wait*. To the best of our knowledge, we do not find a study investigating the impact of different time budgets.

In this work, we report the answer to the first question:

- **RQ:** *By doubling the time budget of an APR tool, do we get twice more plausible patches?*

We follow the methodology used to evaluate anytime algorithms [5] to compare the number of patches generated by five open-source APR tools on a set of 5 benchmarks.

This work was partly funded by the European Commission under the grants n. 952647 (H2020-AssureMOSS) and n.830929 (H2020-CyberSec4Europe).

TABLE I: Benchmarks

Benchmark	Project	#Bugs
Defects4J	Chart	26
Bears	spring-projects-spring-data-commons	15
Bugs.jar	Accumulo	88
IntroClassJava	smallest	52
QuixBugs	40 projects	40

Our preliminary results show that having exponentially more time, APR techniques produce only a linear or no increase in plausible patches and so do not seem to have the trade-off ability for being anytime algorithms.

II. Experiments

For the APR tool selection, we used the *RepairThemAll* framework [2] and selected five tools for Java programs belonging to three repair technique families: *generate-and-validate*, *semantic-driven*, *metaprogramming-based*. The legend of Figure 2 shows the final set of APR tools.

We selected the five most popular benchmarks having both a set of buggy programs with known locations of software bugs and a set of test cases to validate the generated patches. For each benchmark, we randomly selected a project and extracted all its bugs. For the QuixBugs benchmark, we selected all available projects as they contain only one bug per project. This selection resulted in 221 bugs from 44 projects (Table I).

To benchmark the APR tools, we give each tool exponentially longer deadlines and count the corresponding cumulative patches. A generated patch is measured successful if it passes all the specified test cases for the particular software program. An anytime algorithm should provide more successful patches in proportion to the increased amount of available time [5]:

- each tool takes as same input set of the studied programs with known bugs and same starting parameters for each run (i.e., the predefined seed)[1];
- we terminate the repair process if a given time budget is exceeded or the first successful patch is generated;
- after each run, we double the time interval that a tool has for generating plausible bug fixes.

In our experiments, we used an Ubuntu server with eight CPU cores and 62 GB RAM. Each trial run of an APR tool and a buggy program was executed in a separate CPU core. We use the following time intervals: 1 minute (2^0), 2 minutes

[1] *NPEFix* was an exception as it does not support such an option.

(2^1), 4 minutes (2^2), 16 minutes (2^4). When possible, we used the *maxtime* option provided by the framework to limit the execution time. In some cases, we had to kill the repair process after exceeding the timeout.

III. Preliminary findings

Figure 1 shows the fractions of the total number of patches generated by the selected APR tools. We observe that within one minute, the tools fixed 9% of the total number of bugs. When we doubled the time (e.g., two minutes), the number of plausible patches increased to 13.6%. This already deviates from the expected increase in the number of patches: if the APR tools are anytime algorithms, the expected fraction of patches after two minutes is 18%. The difference between the expected fraction of patches and the actually registered ones increases with time. We observe 19% of bugs fixed after running the tools for four minutes (36% is expected) and 28% of bugs fixed after 16 minutes (72% is expected). Hence, we observe a sub-linear increase in the fraction of patches while the tool running time increased exponentially.

Figure 2 shows the fraction of patches each tool generated after a particular time interval. We observe that Nopol fixed the biggest fraction of bugs: already after the first minute, it generated patches for 5% of bugs. However, each consecutive increase of the tool's time budget reduces the number of newly generated patches: +2% after running for the second minute, +2% after four minutes and +1% after running for 16 minutes. Similarly, jKali generated 2% of patches after running for one minute and then was able only to double the number of fixes (5% of bugs) while had 16x more time.

jGenProg and NPEFix fixed a small fraction of bugs after running for one minute (1% of total bugs). After waiting 4x time for jGenProg and 16x time for NPEFix, we obtained additional fixes. However, the number of patches increased modestly while the tools required a substantially longer time to generate them (5x patches required 16x time).

DynaMoth demonstrated a somewhat different behavior. It produced no patches after running for one minute then generated patches for 2% of bugs after two minutes. The next doubling of the time interval leads to just one additional bug fix. After 16 minutes, DynaMoth patched 7% of bugs, which is the second-best result within the evaluated tools. Even if we observe a stepped increase in the number of fixes, it still does not correspond to the exponential growth in the amount of time required for the tool to produce them.

Summary: We evaluated the automated software repair tools with different time budgets. We found that by giving exponentially more time, the number of patches only increases linearly or not at all. If no quick fix is generated (within the first four minutes), one is unlikely to be generated. To conclude this work in progress and really identify if APR tools are anytime algorithms, we plan to find the correct patch out from all plausible patches (the 'better solution') as plausible patches may not correctly fix the related bug [10]. Hence, we plan to i) run the APR tools on a larger dataset and ii) try other more advanced APR techniques, iii) analyze when plausible patches are actually correct patches. Our code and replication data are available in https://github.com/assuremoss/Automated-Program-Repair/tree/main/Anytime-Algorithm-2021.

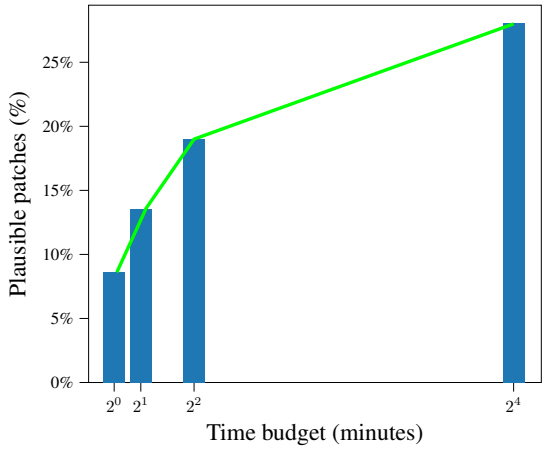

Fig. 1: Total generated patches by all APR tools

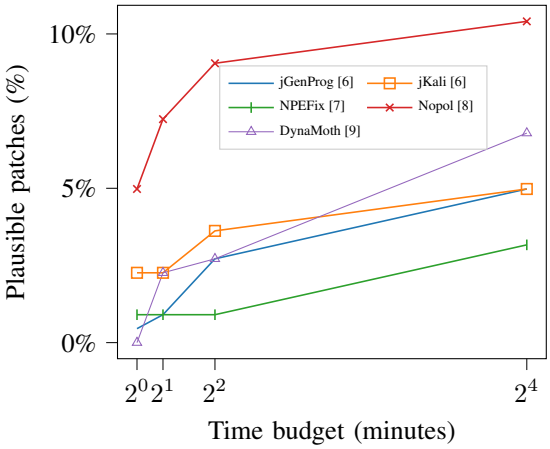

Fig. 2: Generated patches by individual APR tool

References

[1] S. Zilberstein, "Using anytime algorithms in intelligent systems," *AI magazine*, 17(3), 1996.

[2] T. Durieux, et al., "Empirical review of java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *Proc. of ESEC/FSE'19*, 2019.

[3] Y. Qi, et al., "The strength of random search on automated program repair," in *Proc. of ICSE'14*, 2014.

[4] M. Martinez and M. Monperrus, "Ultra-large repair search space with automatically mined templates: The cardumen mode of astor," in *Proc. of SBSE'18*, 2018.

[5] E. A. Hansen and S. Zilberstein, "Monitoring and control of anytime algorithms: A dynamic programming approach," *AI magazine*, 126(1-2), 2001.

[6] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *Proc. of ISSTA'16*, 2016.

[7] B. Cornu, et al., "Npefix: Automatic runtime repair of null pointer exceptions in java," *arXiv*, 2015.

[8] J. Xuan, et al., "Nopol: Automatic repair of conditional statement bugs in java programs," *TSE*, 43(1), 2016.

[9] T. Durieux and M. Monperrus, "Dynamoth: dynamic code synthesis for automatic program repair," in *Proc. of AST'16*, 2016

[10] Z. Qi, et al., "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proc. of ISSTA'15*, 2015.