Interactive Patch Generation and Suggestion

Xiang Gao
National University of Singapore, Singapore
gaoxiang@comp.nus.edu.sg

Abhik Roychoudhury
National University of Singapore, Singapore
abhik@comp.nus.edu.sg

ABSTRACT

Automated program repair (APR) is an emerging technique that can automatically generate patches for fixing bugs or vulnerabilities. To ensure correctness, the auto-generated patches are usually sent to developers for verification before applied in the program. To review patches, developers must figure out the root cause of a bug and understand the semantic impact of the patch, which is not straightforward and easy even for expert programmers. In this position paper, we envision an interactive patch suggestion approach that avoids such complex reasoning by instead enabling developers to review patches with a few clicks. We first automatically translate patch semantics into a set of *what* and *how* questions. Basically, the *what* questions formulate the expected program behaviors, while the *how* questions represent how to modify the program to realize the expected behaviors. We could leverage the existing APR technique to generate those questions and corresponding answers. Then, to evaluate the correctness of patches, developers just need to ask questions and click the corresponding answers.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

ACM Reference Format:
Xiang Gao and Abhik Roychoudhury. 2020. Interactive Patch Generation and Suggestion. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3387940.3392179

1 INTRODUCTION

Automated program repair (APR) can reduce the cost of bug fixing. One of the most challenging problems in today's automated program repair research is the weak specifications. The most commonly studied test-driven APR take a test suite T as program oracles, and find a change to a buggy program P to make it pass T. However, the automatically generated patch may overfit the test data, meaning that the patched program P' passes T but still fail on program inputs/tests outside of T. The overfitting problem is one of the main obstacles that prevent APR being deployed in practice.

Patch suggestion & Challenges To overcome the overfitting issues, instead of directly applying the patches on programs, existing techniques integrate the APR into the development environment [1, 5, 6]. When a bug is triggered, APR tools generate patch candidates, which will be then sent to developers for verification. Ideally, the APR can help developers save precious time by fixing bugs with a single click. However, it is not always true since developers usually have struggled with understanding the cause of the software behavior [4], and hence the semantic impact of a patch. This situation becomes worse if APR suggests more than one patches (APR usually generates a set of plausible patches) for developers.

Debugging reinvented To save the programming time spent on debugging, Ko et al. [4] propose a new program understanding and debugging approach, which translates the program behaviors into a set of *why did* and *why didn't* questions according to program code and execution. Developers are then enabled to speculate about program behavior by selecting questions about program output, such as "why did variable *color=red* at line 15?". Then, this tool generates explanations for the output in question via program analysis and visually provides the answers to developers. This technique can help developers find the underlying cause of a bug more easily. This is because developers usually define program correctness in terms of outputs, and they are usually better at reasoning about program outputs. On the same debugging task, their evaluation shows that novice programmers with this tool are twice as fast as expert programmers without it. These successes inspire us to extend this idea to bug-fixing.

Proposed technique We envision a new approach to allow developers to review and apply patches via a few clicks, instead of directly analyzing the root cause of a bug and the semantic impact of auto-generated patches. To realize it, we propose an algorithm with two steps: offline patch generation and online patch suggestion. In the first step, given the test suite T and buggy program P, we generate a set of patch candidates using existing APR tools, like Fix2Fit [3]. The semantics of patches are then translated into a set of *what* and *how* questions and answers. As complementary to the *why* question, the *what* question asks for the expected output to fix a bug, such as "what should the value of variable *color* be at line 15 to pass the failing test?". The answer to those *what* questions are a set of Angelic values [2], the values that fix the failing test while do not break the passing tests. The *how* questions formulate how to change the program to realize the correct answer to the what questions, such as "how to change the value of variable *color* to blue at line 15?". The answers to those *how* questions are the auto-generated patches that can realize the expected behaviors. In the second step, we provide the pre-generated questions and answers to developers in an interactive way. The users just need to ask questions and select corresponding answers to review the patches. With our approach, users avoid evaluating the semantic effects of a large number of plausible patches. Instead, they review the correctness of much fewer patches in a more direct way. Meanwhile, the slow patch generation is conducted offline, which enables real-time online interactions with developers.

ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3392179

2 METHODOLOGY

The proposed approach could be applied in the continuous integration (CI) or the development process, e.g. IDE plugin. By regularly building, testing, and deploying a program, CI provides the prerequisites for APR tools that use test suites as correctness specifications. Once a bug is detected, i.e. some tests fail, we will generate a set of plausible patches in a form of questions and answers which are then sent to developers via an interactive GUI. Ideally, developers can figure out the correct patches with a few clicks. Overall, the proposed approach consists of the following four main steps.

Offline Patch Generation Given a detected bug and a test suite, we first use existing APR tools to generate a set of plausible patches. Conceptually, we could use any kind of APR tool, either search-based repair or semantic-based repair. Except for the patches, we will also record some intermediate results, e.g. the value of the patch expression on each test, which will be used to generate questions and answers in the following steps.

***What* Questions & Answers Generation** Once the plausible patches are generated, we then set the patched expressions at each location as holes H={h_1, h_2, ... h_i,... h_n}. For each hole h_i, we draw up a what question which asks "what is the expected output of the hole that can make the failing test pass?". These questions formulate the expected behaviors of these holes. For the what question at hole h_i, we then generate the answers by iterating the angelic values V={v_1, v_2, ... v_i,... v_m}. The angelic values are the produced values by plausible patches applied at h_i. For instance, replacing h_i with either plausible patches p_1 or p_2 can pass the failing test t. Under t, the values produced by patch p_1 and p_2 are 2 and 3, respectively. Then, the answers to this what questions will be 2 or 3. Note that, there could be multiple what questions and answers. The task of picking the correct one (indicate the expected program behaviors) is left to developers. To help developers understand the program behaviors, we could also integrate the *why* question [4].

***How* Question & Answer Generation** Once developers select the correct answer (v_i) to the what questions, we then draw up how questions to realize the expected behaviors. Typically, the how question asks how to fill the holes such that the program can behave as expected. We then select the plausible patches that can generate the expected values (v_i). The plausible patches will be then translated into the answers to the how questions. Just like the what questions, there could be multiple answers (p_1, p_2, p_3 ...) to these how questions. The developer will play the role of selecting the correct answer from multiple plausible patches.

Patch Suggestion The generated questions and answers will be sent to developers in an interactive way (e.g. interactive GUI). The role of developers is twofold: (1) ask what question and select the corresponding answer to indicate the expected program behaviors (2) ask how questions and choose correct answers to select patches. This will finally lead developers to find a correct patch.

3 AN EXAMPLE

Listing 1 shows a *FFmpeg* buffer overflow vulnerability [1] which was reported by OSS-Fuzz in 2017. This vulnerability is caused by incorrect bound checking when parsing media files. If *remaining_space* is equal to *width* (line 3), an invalid buffer access will occur at line 10, since it will overwrite the memory locations after *frame_end*. One correct patch for this vulnerability could be modifying the assignment at line 2 from *frame_end-frame-3* to *frame_end-frame-4*.

```
1   int remaining_space = frame_end-frame - 3;
2   //correct patch: "frame_end-frame-3" → "frame_end-frame-4"
3   if (remaining_space < width)
4       return AVERROR_INVALIDDATA;
5   frame[0] = frame[1] = frame[width] =
6           frame[width+1] = bytestream_get_byte(gb);
7   frame += 2;
8   frame[0] = frame[1] = frame[width] =
9   //buffer overflow location
10  frame[width +1] = bytestream_get_byte(gb);
11  frame += 2;
```

Listing 1: Buffer overflow vulnerability in *FFmpeg*

Given the failing test case that can trigger this bug, we first use existing APR tools, e.g. Fix2Fit, to generate a set of patch candidates. For instance, replacing the assignment at line 2 with *frame_end−frame−4*, *frame_end−frame−5*, or *frame_end−frame−6* can fix the failing test. Then, we set the right expression at line 2 as a hole and draw up the what question: "What is the expected value of the hole to make the failing test pass?". The answers to the what question are the angelic values produced by plausible patches. In this case, the angelic values are 8, 7 or 6, all of which can make the failing test pass. If developers indicate 8 is the expected value, we then create the how question: "How to fill the hole to generate the expected value (8)?". The answer to the how question is the patch candidates (in this case, only *frame_end−frame−4*) that can realize the expected value. Developers will play the role of selecting the correct patches from those candidates. The proposed approach provides a way to alleviate the over-fitting problem. Instead of directly applying the patches, we involve developers in the patch generation process in an interactive way, such that, the overfitted patches (e.g. *frame_end−frame−5*) can be filtered out.

4 CONCLUSION

Automated program repair may generate low-quality patches, which prevents them being directly applied in the program. Usually, patches are sent to developers to review. In this project, we envision an interactive patch suggestion approach, which enables developers to select and review patches via a few clicks. The potential benefits of this approach include increasing the efficiency of auto-generated patch review and reducing the burden of developers. We invite the community to consider the deployment of automated program repair in the interactive setting proposed in this position paper.

REFERENCES

[1] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* 3, OOPSLA (2019).

[2] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. 2011. Angelic debugging. In *International Conference on Software Engineering*. 121–130.

[3] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 8–18.

[4] Andrew Ko and Brad Myers. 2008. Debugging reinvented. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 301–310.

[5] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. Sapfix: Automated end-to-end repair at scale. In *ICSE-SEIP (2019)*. IEEE, 269–278.

[6] Martin Monperrus, Simon Urli, Thomas Durieux, Matias Martinez, Benoit Baudry, and Lionel Seinturier. 2019. Repairnator patches programs automatically. *Ubiquity* 2019, July (2019), 1–12.

[1] https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=1345