2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)

Characterizing Developer Use of Automatically Generated Patches

José Pablo Cambronero[*1], Jiasi Shen[*1], Jürgen Cito[*1], Elena Glassman[2], and Martin Rinard[1]

[1]MIT, Cambridge, MA

[2]Harvard University, Cambridge, MA

***Abstract*—We present a study that characterizes the way developers use automatically generated patches when fixing software defects. Our study tasked two groups of developers with repairing defects in C programs. Both groups were provided with the defective line of code. One was also provided with five automatically generated and validated patches, all of which modified the defective line of code, and one of which was correct. Contrary to our initial expectations, the group with access to the generated patches did not produce more correct patches and did not produce patches in less time. We characterize the main behaviors observed in experimental subjects: a focus on understanding the defect and the relationship of the patches to the original source code. Based on this characterization, we highlight various potentially productive directions for future developer-centric automatic patch generation systems.**

I. Introduction

Software defects have been a known problem ever since the inception of the field of software development. Recent research in automatic patch generation has produced systems that have been shown to be capable of generating correct patches for a significant fraction of the considered defects [12], [15], [17], [13]. Many successful automatic patch generation systems for real-world applications use a generate-and-validate approach — the system generates candidate patches that it then validates against a test suite containing sample inputs and outputs. While this approach has been shown to successfully generate correct patches, it has also been shown to generate many more so-called plausible patches that produce correct outputs for all inputs in the test suite, but incorrect outputs for at least some other input [22], [14]. For this reason, the generated patches should be examined by a developer before integration into the source code base. Despite the need for developer involvement, there has been little research characterizing the developer workflow and potential productivity improvements of automatic patch generation in comparison with other alternatives.

Automatic Patch User Study. We present a study that characterizes and compares the developer process in automatic patch generation and manual patch generation aided by defect localization. We compared two populations of developers: one provided with the location of the defective line of code and asked to manually develop a patch, and one provided with five automatically generated patches, all of which validate against the test suite, all of which modify the defective line of code, but only one of which is correct.

Our study provides a qualitative analysis of the recurring behaviors observed in the experimental population.

We summarize our contributions in the following:

User Study: To the best of our knowledge, this paper presents the first study that 1) asks developers to produce correct patches for application logic defects and 2) provides the experimental group with multiple plausible automatically generated patches.

Experimental Results and Qualitative Analysis: We conduct a qualitative analysis on developer interactions with the patches and programs to identify challenges in the program repair process. We characterize the main behaviors observed and use this to derive implications for developer-centric patch generation systems.

Future Directions: Based on the results of our analysis, we formulate potential directions for future patch generation research.

II. Research Method

We present an abridged version of our design methodology and refer interested readers to [4] for details. Materials can be downloaded from [3].

We recruited a total of 12 developers from the doctoral program in computer science at MIT with programming experience in C (either through industry or academia). Subjects were identified as experts or not based on a survey of key C concepts [16].

Subjects were assigned to balance experts across control and experimental cohorts. We carried out two variants of the study: one gave participants 80 minutes, another gave participants 45 minutes. Of the twelve subjects, eight were assigned to the long study and four to the short study. These time allotments are consistent with previous studies in active bug repair, which have allocated a maximum of two hours for five bugs [28]. We recruited the participants telling them the study "evaluates bug fixing tools."

Experimental subjects were given five validated patches generated by Prophet [15] for each bug, one of which was correct. Subjects were not told whether any of the patches were

[*]Equal contribution

978-1-7281-0810-0/19/$31.00 © 2019 IEEE

```
} else if (
  td->td_nstrips > 1 td->td_nstrips > 2
  && td->td_compression == COMPRESSION_NONE
  && td->td_stripbytecount[0] != td->td_stripbytecount[1]
  )
  {
  TIFFWarning(module,
        "%s: Wrong \"%s\" field, ignoring and calculating
             from imagelength",
        tif->tif_name,
                _TIFFFieldWithTag(tif,
                     TIFFTAG_STRIPBYTECOUNTS)->field_name);
                if (EstimateStripByteCounts(tif, dir,
                     dircount) < 0)
                    goto bad;
        }
```

Fig. 1: `libtiff-d13be-ccadf`: The branch condition at line 589 of `libtiff/tif_dirread.c` needs to tighten the first predicate from `td->td_nstrips > 1` to `td->td_nstrips > 2` to successfully repair the bug, which manifests itself as a custom error when `libtiff` tries to estimate the strip byte counts for images that don't satisfy the correct condition.

```
if (len > s1_len - offset) {
  len = s1_len - offset;
}

cmp_len = (uint)
  (len ? len : MAX(s2_len, (s1_len - offset)));
...
```

Fig. 2: `php-309892-309910`: The branch statement starting at line 5255 in `ext/standard/string.c` should be removed to successfully repair the bug, which manifests itself by producing incorrect output for PHP's `substr_compare`.

```
else {
    if ((td->td_nstrips > 1 && td->td_compression == 1 &&
         td->td_stripbytecount[0] != td->td_stripbytecount
         [1]) && !((td->td_nstrips == 2)) )
{  ... }
```

Fig. 3: The generated patch that correctly repairs `libtiff-d13be-ccadf` adds a predicate equivalent to the developer fix, which modifies `td->td_nstrips > 1` to `td->td_nstrips > 2`, highlighted in pink.

correct. All patches modified the same line of code. Control subjects were not given automatically generated patches but received the exact location of the defect.

Study participants were not informed of cohort assignments (until after the experiment) and given tailored instruction manuals and tutorials, including relevant background for each bug. We carried out the study on a pre-configured virtual machine with standardized project structure and utility scripts. Participants also had the option of using a (customized) set of commands in the popular C IDE CLion. No additional tooling was permitted.

Subjects were tasked with repairing two application logic bugs. The bugs were chosen based on constraints on the number of validating patches (at least five), and the need for these to modify the same line of code (in order to effectively provide the same error localization information to control and experimental subjects.) We chose to provide five patches, rather than fewer, as existing state-of-the-art patches tend to produce multiple plausible patches for each bug they are tasked with repairing.

This criteria yielded two bugs for the experiment: `libtiff-d13be-ccadf`, an error-checking bug in the popular TIFF library, and `php-309892-309910`, a bug in PHP's standard library function `substr_compare`. The relevant portions of the source code for each bug, along with the appropriate fix, are presented in Figure 1 and Figure 2, respectively. The developer patches for `libtiff-d13be-ccadf` and `php-309892-309910` are available at [1] and [2], respectively.

In the long study, we allocated 40 minutes for each of these bugs. In the short study, we allocated 25 minutes for `libtiff-d13be-ccadf` and 20 minutes for `php-309892-309910`.

The virtual machines used to carry out the experiments were set up to record all on-screen activities using `avconv` [20]. This totaled nine hours of on-screen activity across the 12 subjects and two study tasks. These videos facilitated our qualitative analysis. In a first pass, two researchers independently collected notes on subject behavior and identified shared behaviors for use as emergent qualitative coding [26].

III. Findings

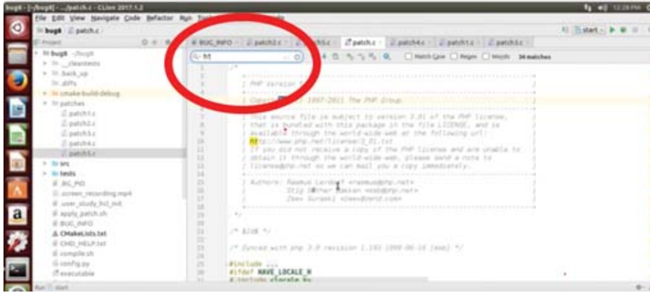
Subjects in the experimental group did not display a significant improvement relative to the control group. Both groups submitted roughly the same number of correct patches. The differences in time-to-first-submission across groups were negligible. We analyzed the experimental subjects' recordings and find key behaviors that we believe drove this outcome and present key challenges for developer use of automatic patch generation systems.

1) **Understanding Code Context:** Experimental subjects on average spent 7.5% (sd=5.6%) of the allocated time per bug searching source code that contained variables used in the automatically generated patches. They searched for declarations, definitions, and uses of variables across the entire

```
if ((len > s1_len - offset) && !(1) ) {
  len = s1_len - offset;
}

cmp_len = (uint) (len ? len : MAX(s2_len, (s1_len - offset)
    ));
```

Fig. 4: The generated patch that correctly repairs `php-309892-309910` adds a false conjunct, highlighted in pink, that is equivalent to the developer fix, which removes the branch statement.



(a) Subject 9 searches for occurrences of variable `ht` in entire source file of `php-309892-309910`

```
if ((len > s1_len - offset) && !((ht == 4))) {
  len = s1_len - offset;
 }

cmp_len = (uint) (len ? len : MAX(s2_len, (s1_len - offset)
     ));
```

(b) Patch provided for `php-309892-309910` that uses variable `ht`

Fig. 5

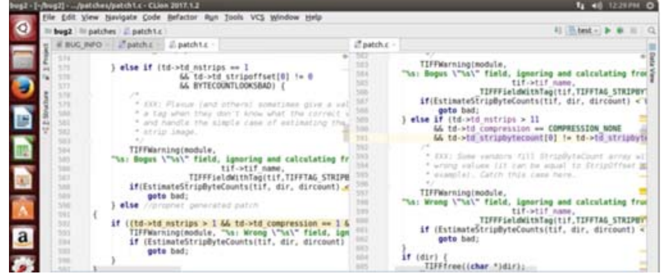


Fig. 6: Subjects used ad-hoc approaches to compare provided patches and the original source code. Subject 10, shown here, viewed provided patches side-by-side. Other subjects switched back and forth between files.

```
if ((len > s1_len - offset) && !(1)) {
  len = s1_len - offset;
  }

cmp_len = (uint) (len ? len : MAX(s2_len, (s1_len - offset)
     ));
```

Fig. 7: Patches that made significant code changes to the original buggy code were inspected less than patches that made incremental changes. We found that patch 1 for `libtiff-d13be-ccadf` and patch 3 for `php-309892-309910` (the correct patch), both of which remove a branch, were only inspected for 3.13% (sd=2.6%) of the allocated time per bug.

file and project, not simply in the area surrounding the defect location.

Figure 5 shows a subject searching for instances of `ht` in the source file that contains the bug. `ht` is one of the variables used in a patch provided to the subject.

Experimental subjects spent 21% (sd=15.8%) of the allocated time per bug inspecting the provided patches. The remainder of the time was divided between reading and modifying the original buggy source code, inspecting tests, and reading the supplemental bug information.

We attribute both of these behaviors to experimental subjects attempting to understand both the overall code they were trying to repair and the interactions between the patch and the application. This highlights the difficulty of patching application-logic bugs, which require understanding the application semantics.

We expected subjects to leverage the patches more heavily, given that the codebases for both bugs were unfamiliar to the subjects. The relatively small fraction of time spent inspecting patches may indicate that subjects prioritized understanding the application over picking a repair that worked, but which they did not understand.

2) **Patch Comparisons:** Reviewing videos of experimental subjects showed that the subjects used a variety of ad-hoc approaches to compare different patches provided. Some participants, such as subject 10 shown in Figure 6, placed patches side-by-side and inspected the source code in both files. Other subjects quickly navigated back and forth between files.

The lack of infrastructure dedicated to comparing differences across patches may have complicated the code understanding task for subjects as they explored patches. We observed subjects spent time switching their screen use between multiple patches, and did so frequently for patches that differed less between each other. The difference between many of the patches provided to subjects was a single predicate.

3) **Code Anchoring:** When comparing patches to the original buggy source code, we found that subjects on average spent only 21% (sd=15.8%) of the allocated time per bug inspecting patches that we provided. Subject 9 spent the highest fraction, 52% of the allotted time on `libtiff-d13be-ccadf`.

Subjects spent less time inspecting patches that make changes that may be considered significant. Patch 1 in `libtiff-d13be-ccadf` and patch 3 in `php-309892-309910`, shown in Figure 7, both add `false` to the branch condition, which is equivalent to removing the branch. For `php-309892-309910` this patch is the correct patch. We found that on average experimental subjects only inspected these two patches 3.13% (sd=2.6%) of the allocated time per bug.

We repeatedly informed experimental subjects that the provided patches all passed the test suites (i.e. validated), but that they were not guaranteed to be correct, and that their task was to produce a correct repair. We also did not inform the experimental group that there was a correct patch amongst the five validated patches they received. However, in five of the 12 submissions from experimental subjects, subjects did not apply additional patches nor make any changes after selecting an initial patch that validated. The single submission that modified a validating patch did so by changing a comparison operation from `!=` to `<` and submitted the new variant. As a result of inspecting the patches for a relatively short period of time, avoiding patches that seemed “unlikely”, and not exploring the application of multiple patches, we believe the experimental subjects made relatively uninformed decisions

for patch selection. This highlights the challenges in selecting from several plausible patches.

IV. Implications For Developer-Centric Patch Generation Systems

Our study illustrates that solely providing subjects with automatically generated patches may not be sufficient to see an effect in terms of patch integration productivity (measured by number of correct bug repairs or time to first patch). Subjects spent most of their time trying to understand the defect and the way the provided patches related to the original source code containing the defect.

Based on our qualitative analysis, we formulate concrete directions for future patch generation systems research. Particularly, providing mechanisms that help developers better understand defects, and the relationship of the candidate patch to the defects, can improve patch integration.

Variable Instrumentation. Our qualitative analysis indicates that developers often spend time investigating the roles that variables from the generated patches play in the original defective code. Future systems could provide enhanced information about variables that occur in the generated patches, for example by providing program slices containing the code that affects the values of these variables [29] or by providing dynamic information flow data that characterizes how these variables influence program outputs [23], [24].

Successful Patch Characteristics. Developers in our study had difficulty identifying correct patches in a set of plausible patches. Consequently, they spent a lot of time trying to find contextual information to assess the correctness of the given candidate patches. Machine learning has previously been successfully used to identify characteristics of correct patches and provide a probabilistic assessment of the viability of a patch [15]. Providing developers with this information can aid patch integration as they inspect candidates, helping them to more quickly distinguish correct from incorrect patches.

Trace and Influence Summaries. Providing the developer with information about how the patch affects program execution characteristics, such as the flow of control and data through the program and output values, may make the potential impact of the patch clearer. This information would be collected during the runs of the original unpatched program and during the validation runs for candidate patches.

Invariants. Previous systems have inferred invariants that characterize successful executions [5], [11]. Providing invariants within the patch integration process that involve variables occurring in the patch may help developers better understand the roles that these variables play in the overall computation.

V. Scope and Threats to Validity

This study models what we believe are a crucial set of circumstances that affect real-world use of a patch generation system. However, the scope of our conclusions is limited by various design choices.

We tasked subjects with repairing two unfamiliar bugs in unfamiliar codebases. We believe this difficult task commonly arises in industry, where new and lateral hires are pervasive. However, this may mean our characterization does not generalize to subjects who have deep codebase knowledge and may be better prepared to identify correct patches.

Subjects were not given access to additional debugging tools in order to control for tooling experience. We chose bugs that we judged were reasonable without additional tools (e.g. no memory bugs). The lack of tools may affect how subjects approached the tasks.

Finally, control subjects were given an ideal case: perfect error localization. Error localization, however, does not always produce the exact location of a defect [21].

VI. Related Work

We present key related work in the area of automatic patch generation research and developer-centric software systems.

Automatic Patch Generation. A significant body of work in the software engineering community has developed new techniques for automatically generating patches [15], [17], [13], [12]. However, a smaller subset of these studies have included meaningful evaluation of developer interaction with such patches through user studies. Previous work, has explored patch understandability and maintainability using questions [6], comparing automatically generated patches between two systems [8], or based on providing users with no patches or a single (possibly) correct patch [28]. Our own work suggests that user studies can reveal new directions for automatic patch generation research to improve useability.

Developer-Centric Software Systems. Research in the HCI community has developed frameworks to understand software errors from the perspective of the developer [9], [10] and investigate developer behavior and tool usage [19], [25]. Incorporating these insights into automatic patch repair could improve the successful adoption of this technology. For example, systems that allow quick diagnoses of errors through inspection [27], code-embedded visualizations [7], or dedicated micro-versioning systems [18] could potentially improve the ability of a developer to quickly identify key variables involved in a bug/patch or compare competing patches.

VII. Conclusion

Automated patch generation systems have been designed to solve the long-lasting problem of software bugs. However, humans remain an important component in integrating the final patch to be applied. Thus a key area of research for automatic patch generation is developer usability and productivity. We provide an initial study to characterize the way developers use automatically generated patches. Based on this we formulate possible research directions to improve developer adoption.

Acknowledgment

The authors would like to thank all people of the user study for their participation.

REFERENCES

[1] libtiff-d13be-ccadf developer patch. https://github.com/vadz/libtiff/commit/3edb9cd. Accessed: 2017-08-25.

[2] php-309892-309910 developer patch. https://github.com/php/php-src/commit/5a8c917. Accessed: 2017-08-25.

[3] Submission online materials. https://drive.google.com/drive/folders/0B9IR8VjNetPYNVVJc0ktVC0tRmc. Accessed: 2017-08-25.

[4] José Pablo Cambronero, Jiasi Shen, Jürgen Cito, Elena Glassman, and Martin Rinard. Characterizing developer use of automatically generated patches. https://arxiv.org/abs/1907.06535, 2019.

[5] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. Science of Computer Programming, 69(1):35–45, 2007.

[6] Zachary P Fry, Bryan Landau, and Westley Weimer. A human study of patch maintainability. In Proceedings of the 2012 International Symposium on Software Testing and Analysis, pages 177–187. ACM, 2012.

[7] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. Augmenting code with in situ visualizations to aid program understanding. In Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, page 532. ACM, 2018.

[8] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In Proceedings of the 2013 International Conference on Software Engineering, pages 802–811. IEEE Press, 2013.

[9] Andrew J Ko and Brad A Myers. A framework and methodology for studying the causes of software errors in programming systems. Journal of Visual Languages & Computing, 16(1-2):41–84, 2005.

[10] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Transactions on software engineering, 32(12):971–987, 2006.

[11] Jens Kruger, Jens Schneider, and Rudiger Westermann. Clearview: An interactive context preserving hotspot visualization technique. IEEE Transactions on Visualization and Computer Graphics, 12(5), 2006.

[12] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms and search spaces for automatic patch generation systems. 2016.

[13] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pages 166–178. ACM, 2015.

[14] Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems. In Proceedings of the 38th International Conference on Software Engineering, pages 702–713. ACM, 2016.

[15] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In ACM SIGPLAN Notices, volume 51, pages 298–312. ACM, 2016.

[16] Sandra Loosemore, Richard M Stallman, Andrew Oram, and Roland McGrath. The gnu c library reference manual. 1993.

[17] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on, pages 691–701. IEEE, 2016.

[18] Hiroaki Mikami, Daisuke Sakamoto, and Takeo Igarashi. Micro-versioning tool to support experimentation in exploratory programming. In Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, pages 6208–6219. ACM, 2017.

[19] Brad A Myers, Andrew J Ko, Thomas D LaToza, and YoungSeok Yoon. Programmers are users too: Human-centered methods for improving programming tools. Computer, 49(7):44–52, 2016.

[20] Jan Newmarch. Ffmpeg/libav. In Linux Sound Programming, pages 227–234. Springer, 2017.

[21] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In Proceedings of the 39th International Conference on Software Engineering, pages 609–620. IEEE Press, 2017.

[22] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, pages 24–36. ACM, 2015.

[23] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In ACM SIGPLAN Notices, volume 50, pages 43–54. ACM, 2015.

[24] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In ACM Sigplan Notices, volume 50, pages 473–486. ACM, 2015.

[25] Jamie Starke, Chris Luce, and Jonathan Sillito. Working with search results. In Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation, pages 53–56. IEEE Computer Society, 2009.

[26] Steve Stemler. An overview of content analysis. Practical assessment, research & evaluation, 7(17):137–146, 2001.

[27] Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, Loris D'Antoni, and Bjoern Hartmann. Tracediff: Debugging unexpected code behavior using trace divergences. In Visual Languages and Human-Centric Computing (VL/HCC), 2017 IEEE Symposium on, pages 107–115. IEEE, 2017.

[28] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. Automatically generated patches as debugging aids: a human study. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 64–74. ACM, 2014.

[29] Mark Weiser. Program slicing. In Proceedings of the 5th international conference on Software engineering, pages 439–449. IEEE Press, 1981.