Understanding Automatically-Generated Patches Through Symbolic Invariant Differences

Padraic Cashin*, Carianne Martinez†, Westley Weimer‡, Stephanie Forrest*
* Arizona State University, Tempe, AZ
† University of New Mexico, Albuquerque, NM
‡ University of Michigan, Ann Arbor, MI

***Abstract*—Developer trust is a major barrier to the deployment of automatically-generated patches. Understanding the effect of a patch is a key element of that trust. We find that differences in sets of formal invariants characterize patch differences and that implication-based distances in invariant space characterize patch similarities. When one patch is similar to another it often contains the same changes as well as additional behavior; this pattern is well-captured by logical implication. We can measure differences using a theorem prover to verify implications between invariants implied by separate programs. Although effective, theorem provers are computationally intensive; we find that string distance is an efficient heuristic for implication-based distance measurements. We propose to use distances between patches to construct a hierarchy highlighting patch similarities. We evaluated this approach on over 300 patches and found that it correctly categorizes programs into semantically similar clusters. Clustering programs reduces human effort by reducing the number of semantically distinct patches that must be considered by over 50%, thus reducing the time required to establish trust in automatically generated repairs.**

I. Introduction

Overview. Developer trust is a major barrier to the deployment of automatically-generated patches. Understanding the effect of a patch is a key element of that trust. We propose to explain patches in terms of how they change formal program invariants. We find that changes in invariants characterize patch differences and that implication-based distances in invariant space can quantify patch similarity. We also present a scalable string-based approximation to implication distance.

Automated Patching. Many tools have been developed to improve the efficiency of software engineering, but the process of patching code to repair bugs remains a dominant cost. To address this issue, many methods have been proposed to automate the patching process, e.g., [10], [11], [13], [16], and over the past decade automated software repair has been an active area of research. Generally, these methods fall into two classes: semantically-sound methods that produce patches that are correct by construction, and methods that generate random mutations and test them for correctness using a supplied test suite. Search-based methods can be applied to legacy software even when they lack a formal specification, they perform well on large code bases, and they generalize across many classes of bugs.

However, these methods have been criticized because they produce multiple patches that are correct only with respect to available tests. Patches that are not trusted are not deployed [1], [17]. Initial industrial deployments have focused on aspects of automatic patching that are simple and easy for developers to assess and trust [7], [12]. These issues suggest the need for methods that can reduce the cost of evaluating automatically-generated patches as produced by search-based methods.

Our Approach. We propose to address this issue by incorporating formal methods into the repair process, using them to show that automatically-generated patches preserve certain aspects of required functionality and to highlight the key semantic changes implied by a proposed patch. Conceptually, we achieve this by generating and comparing relevant program invariants for the original program and the proposed repair. This comparison determines if the patched program violates an important known property of the original program, and it identifies invariants that describe the repair's corrected functionality. We thus avoid an impediment to using formal methods in search-based program repair—a buggy program is incorrect in at least some of its behavior, and therefore does not (currently) represent a correct specification.

We use dynamic invariant generation to create a set of invariants (logical formulae over program variables encoding relationships that are true on indicative runs), one for the buggy program and one for each proposed repair. Since most software bugs are revealed through a failing input, we use the failing tests, together with the supplied test suite, to generate invariants that are most relevant to the repair. We group repairs into quasi-equivalence classes using one of two distance metrics, one based on logical implication and one based on syntactic comparison of the invariant sets. Each class is made up of invariant sets that are distance zero from each other under the chosen metric. We use hierarchical clustering to highlight how the classes are related.

In addition, we explicitly compare the invariants of each patch to those of the original buggy program. This approach highlights the key semantic differences between the original program and each patch, and it allows a developer to quickly consider multiple possible patches by evaluating only one element from each semantic class.

Preliminary Findings. We hypothesize that *dynamic invariants can detect changes within a program.* Invariants are effective because functional correctness relates to the final result of a function rather than any specific implementation. We propose the use of *implication relationships* between

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00046

```
+ ssize = ((a->_mp_size) >= 0 ? (asize != 0) : -1;
---
- ssize = ((a->_mp_size) >= 0 ? 1 : -1;
```

(a) Tool 1 Patch

```
+ ssize = SIZ (a) >= 0 ? (asize != 0) : -1;
---
- ssize = SIZ (a) >= 0 ? 1 : -1;
```

(b) Official Patch

```
+  if (1) return (-1);
...
+ _gmpn_cmp (ap, bp, asize) < 0
---
- mpn_cmp (ap, bp, asize) < 0
```

(c) Tool 2 Patch (Fragment)

Fig. 1: Three patches for GMP-14166-14167 [9], which returns an incorrect answer from the `gmp_ext` function.

sets of invariants to measure pairwise patch similarity and group invariant sets into quasi-equivalence classes according to similarities between their implication relationships. Because it is expensive to compute implication distance, we propose *string distance as an effective approximation for implication distance.* A formulation involving Levenshtein edit distance is significantly more scalable than a formal approach based on Satisfiability Modulo Theories (SMT) decision procedures. We hypothesize this approach, which we call PATCHPART, to be efficient and effective.

II. MOTIVATING EXAMPLE

This section considers candidate patches produced by two different program repair tools and shows how PATCHPART identifies semantically-meaningful clusters of patches and reports their semantic differences. Results of earlier studies of search-based repair tool quality [17], [18] have been mixed. Some papers have identified flaws in particular examples of automatically-generated patches [17]. We consider one of these latter cases here.

We take as an example one defect in the GNU Multiple Precision Arithmetic Library [9], and two tool repairs (referred to as Tool 1 [10] and Tool 2 [17] respectively). The defect is located in the `gcdext` function and involves the incorrect resolution of `gcd(0,0)`. An incorrectly-constructed conditional causes the return value to be 1. We apply PATCHPART to four versions of GMP: the original buggy version and three patches (from Tool 1, Tool 2, and the official human GMP developers). In this instance, the Tool 1-generated patch matches the developer version more closely than Tool 2. We used Daikon [5] to generate invariant sets for each variant.

Figure 1 shows the three different patches. Tool 2 inserts a conditional in a helper function, which causes it to always return -1, bypassing most subsequent calculations (including a later comparison predicate). Although this is sufficient to pass the given test suite, the modified conditionals cause `gcd()` to return incorrect results on other inputs. The modification to the helper function could have additional consequences not immediately apparent to developers only considering the effects on the main `gcd` function.

Tool 1 and the official patch each update a conditional expression involving size comparisons. This change causes `ssize` to be set to 0 in the case where `asize = 0`. Although, these patches caused the smallest modifications to the program syntax, it is not clear if they are ideal.

We used PATCHPART with Daikon-produced invariant sets to partition the repaired program variants, which produced three semantic classes. PATCHPART correctly isolates the buggy version in its own cluster and finds two semantically-distinct clusters for the repairs: one containing Tool 2, and the other containing Tool 1 and official patch. PATCHPART highlights key differences between the clusters. For example, in this case, the invariant `temp2.mp_d != a.mp_d` is a property of the Tool 2 patch but not the others. The invariant indicates that the values of `mp_d` members for `temp2` and `a` always remain different, formalizing the effects of that patch's `return`.

Although the patches are syntactically distinct, Tool 1 and the official maintainer patches generate identical sets of invariants. A developer need only inspect two patches—one from the $\{$ Tool 2 $\}$ cluster and one from the $\{$ Tool 1, Official $\}$ cluster—rather than all three. Note that non-functional quality properties, such as readability, are not considered here: the developer patch makes use of the `SIZ` macro while the tool patch uses its definition. Various approaches have been proposed for improving readability in search-based methods; we view readability concerns as orthogonal and focus on functional aspects of patch inspection and trust.

This example shows how PATCHPART identifies semantic clusters of patches, detects important differences between the original program and different repairs, and reduces the number of candidate patches a developer must manually review.

III. TECHNICAL APPROACH

We require an approach that is accurate enough to distinguish between different patches, fast enough to be applied as part of a patch cycle, and that requires minimal human intervention. We present PATCHPART, a method for partitioning programs into semantic clusters, which addresses these requirements.

*A. Patch Partitioning (*PATCHPART*)*

PATCHPART takes as input a set of programs. It then computes the pairwise semantic distance between them (in invariant space) and clusters the programs into semantically-similar classes based on those distances. More specifically, PATCHPART constructs a set of pre-conditions and post-conditions for each function in each considered program, uses off-the-shelf methods (such as Daikon [5]) to produce dynamically-generated program invariants, and then calculates the invariant distance between each program pair. The dynamic invariants are generated from the test suite and the bug-inducing test case.

We consider two methods for computing the pairwise distance between sets of invariants: formal implication between logical formulae [3] and Levenshtein edit distance [15] between their string representations. Theorem proving, using an SMT solver, determines whether one invariant is logically implied by another, while Levenshtein distance compares two sets of invariants syntactically. Finally, we use hierarchical clustering to group programs based on a distance calculation.

B. Implication Distance

Semantic similarity between two programs can be captured by computing the logical similarity between their corresponding sets of program invariants. We thus lift a comparison on sets of invariants to a comparison on programs.

We define the *implication distance* (ID) between two programs, A and B, to be the cardinality of $\mathsf{Inv}(B, \mathcal{T}) - \mathsf{ImplInv}(A, B)$, where $\mathsf{Inv}(X, \mathcal{T})$ is the set of dynamic invariants generated from tracing program X on test suite $\mathcal{T}$ and $\mathsf{ImplInv}(A, B) = \{b \mid b \in \mathsf{Inv}(B, \mathcal{T}) \wedge \exists a \in \mathsf{MinTerms}(\mathsf{Inv}(A), \mathcal{T}).\ a \Rightarrow b\}$ where Inv is a function mapping programs to generated invariants (e.g., such as via Daikon) and $\mathsf{MinTerms}$ is a function mapping sets of predicates to all subsets of a particular size. We restrict minterms to a maximum size of three to minimize computation time following established best practices from predicate abstraction [2, p. 112]; this size is known to be efficient while retaining enough information to prove program correctness.

Our definition of ID relaxes the standard notion of set difference from requiring logical equivalence to requiring only an implication relation. Intuitively, the implication distance represents a measure of the number of invariants in program B that are not implied by the invariants of program A.

As a simple example of why we propose implication (not equivalence) to capture program relationships, consider two programs, each with a loop which increments a value x: `for(i=0; i<N; ++i) { x++; }`. If, for program A, $x = 0$ at the start of the loop, then we should find the invariant $x \geq 0$ at the end of the loop. However, if $x = 1$ initially in program B, then we will find $x \geq 1$ post-loop for B instead. Such programs are not strictly equivalent, but the implication $(x \geq 1) \Rightarrow (x \geq 0)$ holds. This definition allows us to use hierarchical clustering to partially order programs.

Given the restriction to minterms, we can directly compute ID from its definition. We consider all minterms of a given size and interate, querying a prover to check implications. We use the usual conjunction interpretation when determining if a set of predicates implies another predicate. Our implementation uses the Z3 theorem prover [3] to determine satisfiability; our approach is thus sound with respect to that tool.

C. Levenshtein Edit Distance

A direct calculation of ID requires many queries to an expensive external SMT solver. As an efficient approximation for ID, we propose a string distance measure between invariant sets. Levenshtein edit distance (LD) [15] measures the number of character swaps, additions, and deletions needed to convert one string into another. We map each invariant to a single logical alphabet symbol, representing syntactically-identical invariants with the same symbol. For example, $x = 2$ and $x = 2$ both map to the same symbol, but $x = 1 + 1$ maps to a different symbol even though it is semantically equivalent—*not* determining semantic equivalence is the heart of our efficient approximation. We then compute the distance between the two induced strings of symbols.

The intuition behind this approximation follows from our use case of comparing multiple automated program repair patches. Most program repair patches are small and thus they leave the majority of the original program textually unchanged [9], [16]. Textually identical program fragments often yield invariants that are not just semantically equivalent but are directly syntactically equivalent (cf. [19]). Thus, an approximation based on syntactic equivalence has the potential to be more accurate in this use case than it would be in general.

D. Hierarchical Clustering

We propose the use of the Unweighted Pair Group Method with Arithmetic Mean (UPGMA) clustering algorithm [6]. UPGMA takes as input a distance matrix and identifies clusters that minimize the average cluster diameter. Critically, since the UPGMA algorithm does not require the distance matrix to be symmetric, we can use either LD or ID measurements to group programs. A cluster of identical patches can be inspected for a functional trust assessment by inspecting any representative of it. On the other hand, the distances between clusters and the invariants on which they differ can communicate the effects of a patch to developers.

IV. Preliminary Results

We evaluated a prototype of PatchPart on a set of 5 programs from the ManyBugs [10] and 7 programs from the Defects4J [8] benchmarks. We used 50 GenProg [10] patches for each C defect and 20 ARJA [20] patches for each Java defect (repair tools are often language-specific but our approach is agnostic). Daikon was used to find dynamic invariants. By computing the pairwise distances between individuals we are able to place patches into a hierarchy based on functional similarities.

For each program we studied, candidate patches were easily distinguished from the original program and relatively few invariants differentiated the repairs from the original, supporting our hypothesis that PatchPart can provide a concise assessment of the key semantic elements of a proposed repair, allowing a developer to quickly check that a repair retains required functionality. When a human-generated repair is not available, the intended use case for automated bug repair, PatchPart can provide the developer with a small set of semantically-distinct patches, highlighting the semantic differences between them.

We found that Levenshtein Distance performs almost as well as Implication Distance for our use case. However, we do not attribute a measure of importance to each invariant, and instead assume each invariant is equally meaningful. An investigation of this assumption in this context is left to future work.

These preliminary results suggest that PATCHPART can successfully categorise patches based on the invariants detected by Daikon. For each group of patches we can determine a hierarchy relationships, with each layer of the hierarchy containing more patches and representing a broader, more abstract set of features. For most defect scenarios this led to a reduction of patch classification effort by approximately 50% (i.e., even when requiring distance-zero invariant sets, the clusters were large enough to save developer inspection effort). Although these results are preliminary, we expect this approach to find useful groupings among other patches.

V. RELATED WORK

The two most relevant areas of related work are, broadly, invariant detection and automated program repair. For reasons of space we elide the rich literature on invariant detection (our approach is agnostic; our preliminary evaluation uses Daikon) and program repair (our approach is agnostic; the reader is directed to Monperrus [14] for a survey).

We focus on one particular related paper to place this work in context. Recent work by Ding et al. [4] explored the use of invariants to measure diversity among patched programs in the context of a multi-objective genetic algorithm. Their work uses invariants as part of the repair process to generate a wider diversity of repairs than those typically found by search-based methods. Similar to PATCHPART, they rely on dynamically-generated invariants. In contrast, however, they measure the frequency with which invariants appear across different test cases and use that information to guide the repair. Our approach takes already-produced repairs, uses invariant information to guide human understanding, does not rely on frequencies, and proposes the Levenshtein Distance approximation to improve scalability.

VI. CONCLUSION

Automated testing and repair methods are advancing rapidly and transitioning to industrial practice. Search-based repair methods often produce many candidate patches which must be manually inspected and understood before being deployed. By focusing on differences, both among candidate patches and between patches and the original program, our PATCHPART approach offers an intermediate path that supports the use of formal invariants in settings without formal specifications.

By using an efficient string-based distance measures on dynamic invariant sets, PATCHPART can detect and enumerate the relevant semantic differences between programs. In our preliminary results, the detected invariants correctly separated the original buggy program from patched programs that pass the failing test suite, and in most cases, semantically-identical program variants were correctly placed in clusters. Providing the developer with this additional information should simplify the task of inspecting candidate repairs, reducing that burden by about 50% in our preliminary investigation.

VII. ACKNOWLEDGMENTS

The work reported here was partially supported by NSF (CCF1908633,CCF1763674), AFRL (FA8750-19-2-0006, AFRL), and DARPA (FA8750-19C-0003).

REFERENCES

[1] G. M. Alarcon, L. G. Militello, P. Ryan, S. A. Jessup, C. S. Calhoun, and J. B. Lyons. A descriptive model of computer code trustworthiness. *Journal of Cognitive Engineering and Decision Making*, 11(2):107–121, 2017.

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Workshop on Model Checking Software (SPIN)*, pages 103–122, 2001.

[3] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[4] Z. Y. Ding, Y. Lyu, C. S. Timperley, and C. L. Goues. Leveraging program invariants to promote population diversity in search-based automatic program repair. In *Genetic Improvement*, 2019.

[5] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[6] I. Gronau and S. Moran. Optimal implementations of upgma and other common clustering algorithms. *Information Processing Letters*, 104(6):205–210, 2007.

[7] S. O. Haraldsson, J. R. Woodward, A. E. I. Brownlee, and K. Siggeirsdottir. Fixing bugs in your sleep: How genetic improvement became an overnight success. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1513–1520, 2017.

[8] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[9] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.

[10] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A genetic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, Jan 2012.

[11] F. Long and M. Rinard. Automatic patch generation by learning correct code. *ACM SIGPLAN Notices*, 51(1):298–312, 2016.

[12] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott. Sapfix: Automated end-to-end repair at scale. In *International Conference on Software Engineering (ICSE), Software Enginering in Practice (SEIP) track*, 2019.

[13] M. Martinez and M. Monperrus. Astor: A program repair library for java. In *Proceedings of the 25th Intl. Symp. on Software Testing and Analysis*, pages 441–444. ACM, 2016.

[14] M. Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)*, 51(1):17, 2018.

[15] G. Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001.

[16] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781. IEEE Press, 2013.

[17] Z. Qi, F. Long, S. Achour, and M. 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.

[18] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543. ACM, 2015.

[19] X. Yin, J. C. Knight, E. A. Nguyen, and W. Weimer. Formal verification by reverse synthesis. In *Computer Safety, Reliability, and Security*, pages 305–319, 2008.

[20] Y. Yuan and W. Banzhaf. ARJA: automated repair of java programs via multi-objective genetic programming. *CoRR*, abs/1712.07804, 2017.