Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2015-019 May 26, 2015

Prophet: Automatic Patch Generation via Learning from Successful Human Patches

Fan Long and Martin Rinard

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Prophet: Automatic Patch Generation via Learning from Successful Human Patches

Fan Long and Martin Rinard
MIT CSAIL
{fanl, rinard}@csail.mit.edu

Abstract

We present Prophet, a novel patch generation system that learns a probabilistic model over candidate patches from a large code database that contains many past successful human patches. It defines the probabilistic model as the combination of a distribution over program points based on error localization algorithms and a parameterized log-linear distribution over modification operations. It then learns the model parameters via maximum log-likelihood, which identifies important characteristics of the successful human patches. For a new defect, Prophet generates a search space that contains many candidate patches, applies the learned model to prioritize those potentially correct patches that are consistent with the identified successful patch characteristics, and then validates the candidate patches with a user supplied test suite.

1. Introduction

Automatic patch generation holds out the promise of correcting defects in production software systems without the cost for human developers to diagnose, understand, and correct these defects. The standard *generate-and-validate* techniques start with a test suite of inputs, at least one of which exposes a defect in the software [7, 8, 12, 13, 17]. The patch generation system then applies program modifications to generate a space of candidate patches, then searches the generated patch space to find *plausible* patches - i.e., patches that produce correct outputs for all inputs in the test suite.

Recent work shows that generate-and-validate techniques can suffer from *weak test suite problem*, i.e., these techniques often generate plausible but *incorrect* patches that pass the supplied test suite but produce incorrect outputs for other inputs [8, 13]. These plausible but incorrect patches often have negative effects such as eliminating desired functionalities or introducing security vulnerabilities in an application [13]. This indicates that additional information other than test suites is desirable for the patch generation techniques.

We present Prophet, a new generate-and-validate patch generation system that learns from past successful human patches. Prophet analyzes a large database of human revision changes collected from open source project repositories to automatically learn characteristics of successful patches. It then uses these learned characteristics to recognize and prioritize correct patches within a larger space of plausible patches. To the best of our knowledge, Prophet is the first program repair technique that uses a probabilistic model to identify important patch characteristics and a machine learning algorithm to learn the model from past successful human patches.

2. Basic Approach

Prophet first performs an offline training phase to learn important characteristics (features) of successful human patches from a large revision database. For a new defect, Prophet then generates a search space of potential useful patches for the defect and uses the learned knowledge to prioritize the patches that tend to be successful, i.e., the learned knowledge guides the exploration the search space.

2.1 Training

The input to the Prophet training phase is a large revision change database $D = \{\langle p_1, p\prime_1\rangle, \ldots, \langle p_n, p\prime_n\rangle\}$, where each element of D is a pair of defective and repaired programs p_i and $p\prime_i$. Prophet learns a probabilistic model such that, it assigns a high conditional probability to $p\prime_i$, denoted as $P(p\prime_i \mid p_i)$, for each pair $\langle p_i, p\prime_i\rangle \in D$. Note that there are other possible repaired programs $p\prime$ given program p_i, the probability thus can be interpreted as a normalized score (that is, $\sum_{p\prime} P(p\prime \mid p_i) = 1$) which prioritizes the correct repaired program $p\prime_i$ among all possible candidates (in the search space Prophet generates).

Probabilistic Model: More precisely, the probabilistic model of Prophet assumes that each candidate patch modifies one program point. Specifically, the model assumes that a repaired program $p\prime$ can be derived from a given defective program p by first localizing a program modification point

ℓ and then performing an AST modification operation m at this point. Based on this assumption, Prophet factorizes the probability $P(p\prime \mid p)$ as follow:

$$\begin{aligned} P(p\prime \mid p) &= P(m, \ell \mid p) \\ &= P(\ell \mid p) \cdot P(m \mid p, \ell) \qquad \text{(chain rule)} \end{aligned}$$

$P(\ell \mid p)$ is a distribution that corresponds to the probability of modifying the program point ℓ given the defective program p. Prophet runs an error localization algorithm [8] to obtain a ranked list of potential program points to modify to generate patches. Prophet empirically defines $P(\ell \mid p)$ as a normalized geometric distribution,

$$P(\ell \mid p) = \frac{(1-\alpha)^{r-1}\alpha}{Z}$$

where Z is the normalization divisor, r is the rank of the program point ℓ determined by the error localization algorithm, and α is the probability of each coin flip trial (which Prophet empirically sets to 0.08).

$P(m \mid p, \ell)$ is a parameterized log-linear model,

$$P(m \mid p, l) = \frac{\exp\left(\phi(p, m, \ell) \cdot \theta\right)}{\Sigma_{m\prime \in M} \exp\left(\phi(p, m\prime, \ell) \cdot \theta\right)}$$

where M is the set of all candidate modification operations (including m) given p and ℓ, $\phi(p, \ell, m)$ is the feature vector that Prophet extracts for the triple of p, ℓ, and m, and θ is the feature weight parameter vector Prophet learns from the revision change database.

Learning Steps: Prophet learns θ by maximizing the average log likelihood of the observed pairs of the defective and the repaired programs in D. Specifically, Prophet performs the following steps:

- **AST Structured Diff:** For each pair $\langle p_i, p\prime_i \rangle$ in D, Prophet performs a structured diff between the ASTs of p_i and $p\prime_i$ to determine the corresponding modification operation m_i and the modified program point ℓ_i for $\langle p_i, p\prime_i \rangle$.
- **Generate Repair Space:** For each pair $\langle p_i, p\prime_i \rangle$ in D, Prophet then generates a search space that contains a set of candidate repairs for p_i around the identified program point ℓ_i. Each of the candidate repairs correspond to a modification operation and this gives a set of modification operations M_i. Note that one of the candidate repairs in the generated search space corresponds to the identified modification operation m_i for the repaired program $p\prime_i$ (i.e., $m_i \in M_i$).
- **Maximum Loglikehood:** Prophet initializes θ with zeros and runs the iterative algorithm to maximize:

$$\frac{1}{|D|} \cdot \Sigma_{i=1}^{|D|} \log(P(m_i \mid p_i, \ell_i))$$

For each iteration Prophet calculates the average log likelihood over the collected training pairs in D; Prophet then updates the θ with the gradient decent algorithm.

- **Regularization and Validation Set:** To determine when the iterative algorithm stops, Prophet splits the training data set and reserves 15% of the training pairs for validation only. The algorithm records the θ that produces the best results on the validation set and it stops when the result on the validation set is no longer improved for 200 iterations. To avoid the overfitting problem, Prophet uses both L1 and L2 regularizations during training.

2.2 Feature Selection

Prophet extracts two types of features, modification features and semantic features. In our current implementation, the feature vector ϕ in Prophet contains 3425 elements.

Modification Features: Prophet defines a set of modification features to characterize the type of the modification operation m given the surrounding code. Specifically, Prophet uses the same set of transformation schemas in SPR [8] to generate candidate patches. For each transformation schema, there is a binary value in $\phi(p, \ell, m)$ that indicates whether m is generated from this specific schema. For each pair of a transformation schema and a statement type (e.g., assign statement, branch statement), there is a binary value in $\phi(p, \ell, m)$ which equals one if and only if m is generated from the specific schema and the original code around ℓ contains a statement that is the specific type.

Semantic Features: Prophet also defines a set of semantic features to capture the common semantic characteristics of successful or unsuccessful human patches. Specifically, for each syntactic value v (e.g., a local variable v), Prophet defines a set of binary atomic semantic features $a_i(v, p, \ell, m)$. Examples of such atomic features are whether v is dereferenced in the original code around ℓ or whether v is used as the left operand of a less than operation in the new code after applying the modification m.

For each pair of such atomic semantic features a_i and a_j, there is a binary value in $\phi(p, \ell, m)$ which equals one if and only if there is a syntactic value v such that both $a_i(v, p, \ell, m)$ and $a_j(v, p, \ell, m)$ are one.

A key benefit of defining semantic features in this way is that it abstracts (potentially application specific) syntactic elements away from the extracted feature vectors. This abstraction enables Prophet to learn semantic features from one application and then applies the learned knowledge to another application.

2.3 Apply to New Defect

Given a program p that contains a defect and a test suite that contains at least one test case which exposes the defect,

Prophet performs the following steps to generate a patch for p:

- **Error Localization:** Prophet runs all test cases in the test suite with an error localization algorithm to produce a ranked list of potential program points (statements) that may correspond to the root cause of the error. Prophet uses the same error localization algorithm as SPR [8]. Prophet uses the error localization results to compute $P(\ell \mid p)$.
- **Generate Search Space:** Prophet then generates a search space that contains candidate patches for all of the program points returned by the error localization algorithm. Each candidate patch corresponds to a pair $\langle m, \ell \rangle$, where m is the modification operation and ℓ is the program point to modify.
- **Rank Candidate Patch:** For each candidate patch, Prophet uses the learned θ to compute $P(m \mid p, \ell)$ and, in turn, computes $P(m, \ell \mid p)$ as the product of $P(\ell \mid p)$ and $P(m \mid p, \ell)$. Prophet uses $P(m, \ell \mid p)$ as the score for the candidate patch. Prophet then sorts all of the candidate patches in the search space according to this score, prioritizing candidate patches with higher probability.
- **Validate Candidate Patch:** Prophet tests all of the candidate patches one by one in the ranked order with the supplied test suite. Prophet outputs the first candidate patch that passes the test suite. Note that Prophet uses all existing optimization techniques in SPR to speed up this validation step, including the staged condition synthesis technique [8].

3. Experimental Results

We evaluated Prophet on 69 real world defects in seven large open source applications, libtiff, lighttpd, the PHP interpreter, gmp, gzip, python, wireshark, and fbc. These defects are from the same benchmark set of GenProg, AE, and SPR [7, 8, 17]. Note that we exclude 36 cases from the original benchmark set because those 36 cases correspond to functionality changes rather than defects in the application repositories [8].

3.1 Methodology

We perform all of our experiments except those of fbc on Amazon EC2 Intel Xeon 2.6GHz machines running Ubuntu-64bit server 14.04. The benchmark application fbc runs only in 32-bit environments, so we use a virtual machine with Intel Core 2.7Ghz running Ubuntu-32bit 14.04 for the fbc experiments. We perform our experiments as follows:

Collect Successful Human Patches: We collect in total more than 20000 revision changes from eight open source project repositories. To control the noise from the training data, we filtered a significant part of the collected revisions because either 1) these revisions do not correspond to a patch for a defect, 2) we cannot compile these collected revisions in our experiment environment, or 3) the AST modifications of these collected revision changes are outside the defined search space of Prophet. After filtering, we use 669 revisions in total for the Prophet training. Figure 1 presents the statistics of the collected code database.

Project	Revisions Used for Training
apr	12
curl	47
httpd	72
libtiff	11
php	162
python	96
subversion	199
wireshark	70
Total	669

Figure 1. Statistics of Collected Code Database

System	Correct	Avg. Rank in Search Space
Baseline	5	Around 50%
Prophet	14	11.3%
SPR	11	27.2%
GenProg	1	N/A
AE	2	N/A

Figure 2. Experimental results of Prophet, SPR, randomized search, GenProg, and AE.

Train Prophet with Collected Database: We train Prophet with the collected database. Note that our collected code database and our evaluation benchmark share four common applications, libtiff, php, python, and wireshark. For each of the four applications, we train Prophet separately and exclude the revision changes of the same application from the training data. The goal is to measure the capability of Prophet to apply the learned characteristics of successful human patches across different applications.

Apply Prophet to Defects: We then use the trained Prophet to generate patches for each defect. For comparison, we also run SPR and a baseline random search algorithm to generate patches for each defect as well. We terminate a patch generation system execution if the system fails to generate a plausible patch (i.e., which passes the supplied test suite) within 12 hours.

3.2 Result Summary

Figure 2 presents the summary of the experimental results of Prophet, in comparison with the random search baseline

algorithm, SPR, GenProg, and AE. Note that we collect GenProg and AE results from previous work [8, 13].

The first column (System) presents the evaluated system name. Prophet, SPR, and the baseline random search system operate on the same search space with different candidate patch test orders. Prophet uses the learned characteristics of successful human patches to determine the test order; SPR uses a set of empirical deterministic rules to determine the order [8]; the baseline algorithm randomly shuffles all candidate patches to determine the test order. Note that we found that the staged condition synthesis technique [8] can significantly improve the efficiency of the patch generation systems for generating and validating candidate patches that manipulate conditions. For a fair comparison, the staged condition synthesis technique [8] is enabled in both of the baseline, SPR, and Prophet in our experiments.

The second column (Correct) presents the number of defects for which each system generates a correct patch as the *first* generated plausible patch. Our results show that Prophet generates correct patches for 14 defects, 9 more than the baseline random search and 3 more than SPR. GenProg and AE generate correct patches for only 1 and 2 defects respectively. One potential explanation is that the search space of these systems often does not contain correct patches [8, 13].

The third column (Avg. Rank in Search Space) presents an percentage number, which corresponds to the rank, normalized to the size of the search space per defect, of the first correct patch in patch test order of each system. This number is an average over the 19 defects for which the search space of these systems contains at least one correct patch. Prophet ranks correct patches as top 11.3% among all candidate patches on average. In contrast, SPR, which uses a set of empirical deterministic rules, ranks correct patches as top 27.2% on average. This result highlights the capability of Prophet to prioritize the correct patches based on the learned knowledge.

We attribute the success of Prophet to the learned characteristics of successful human patches that enable Prophet to prioritize a correct patch among multiple plausible patches that pass a supplied test suite. Note that the search space of Prophet is identical to that of SPR, which contains correct patches for 19 out of the 69 evaluated defects [8]. Prophet generates correct patches for over 70% of the defects for which the Prophet search space contains any correct patch.

3.3 Correct Patch Results

Figure 3 presents the detailed results of each of the 19 defects for which the Prophet search space contains correct patches. The first column (Defect) is in the form of X-Y-Z, where X is the name of the application that contains the defect, Y is the defective revision in the repository, and Z is the revision in which developer repaired the defect. The second, third, and forth columns present the result of Prophet, SPR, and the baseline random search system on each defect. "Correct" indicates that the system generates a correct patch (as the first generated plausible patch). "Plausible" indicates that the system generates a plausible but incorrect patch. "Timeout" indicates that the system does not generate a plausible patch in 12 hours.

The fifth column (Search Space) presents the number of candidate patches in the search space for each defect. The sixth and seventh columns present the rank of the first correct patch among all candidate patches in the patch test orders determined by Prophet and SPR respectively. The eighth column (Time) presents the execution time of Prophet for generating the first plausible (and often correct) patch.

We acknowledge that, in general, determining whether a specific patch corrects a specific defect can be difficult (or in some cases not even well defined). We emphasize that this is not the case for the patches and defects that we consider here. The correct behavior for all of the evaluated defects is clear, as is patch correctness and incorrectness. Furthermore, our manual code analysis show that each of the generated correct patches in our experiments is, in general, semantically equivalent to the corresponding developer patch in the repaired revision.

4. Related Work

SPR: SPR is the current state-of-the-art search-based patch generation system [8]. For the same benchmark set as GenProg and AE, the SPR search space contains correct patches for 19 defects. With a staged repair algorithm that enables SPR to efficiently explore patches that manipulate branch conditions, SPR is able to generate correct patches for 11 out of the 19 defects (10 more than GenProg and 9 more than AE). Prophet differs from SPR because Prophet learns from the past successful human patches and uses this learned information to guide the search space exploration. Our results show that Prophet outperforms SPR in generating correct patches for 3 more defects in the same benchmark set.

CodePhage: CodePhage automatically locates correct code in one application, then transfers that code into another application [16]. This technique has been applied to eliminate otherwise fatal integer overflow, buffer overflow, and divide by zero errors. Prophet differs from CodePhage, because CodePhage relies on the existance of a specific donor application that contains the exact program logic to fix an error, while Prophet learns general characteristics of successful patches to guide the search space exploration of many candidate patches for a defect.

GenProg, RSRepair, AE, and Kali: GenProg [7, 18] uses a genetic programming algorithm to search a space of patches, with the goal of enabling the application to pass all considered test cases. RSRepair [12] changes the GenProg algo-

Defect	Prophet	SPR	Baseline	Search Space	Correct Rank	SPR Correct Rank	Time
php-307562-307561	Correct	Correct	Correct	15733	1285	6253	10m
php-307846-307853	Correct	Correct	Correct	62494	9825	22247	264m
php-308734-308761	Correct	Correct	Correct	6459	1397	2717	105m
php-309516-309535	Correct	Correct	Timeout	64767	9003	21837	145m
php-309579-309580	Correct	Correct	Correct	10831	362	604	40m
php-309892-309910	Correct	Correct	Plausible	16582	388	68	114m
php-310991-310999	Correct	Correct	Timeout	555622	10838	16563	333m
php-311346-311348	Correct	Correct	Plausible	69738	14	1042	55m
php-308262-308315	Correct	Plausible	Timeout	17232	641	6492	63m
php-309688-309716	Plausible	Plausible	Plausible	8018	881	2986	63m
php-310011-310050	Plausible	Plausible	Plausible	7220	2270	7163	118m
php-309111-309159	Plausible	Plausible	Plausible	31245	4090	31005	36m
libtiff-ee2ce-b5691	Correct	Correct	Timeout	477646	10548	184789	21m
libtiff-d13be-ccadf	Correct	Plausible	Plausible	815190	2539	8453	17m
libtiff-5b021-3dfb3	Plausible	Plausible	Plausible	268497	110159	62775	10m
gmp-13420-13421	Correct	Correct	Correct	31570	9392	8585	116m
gzip-a1d3d4-f17cbd	Correct	Correct	Plausible	101651	6224	8211	27m
lighttpd-2661-2662	Plausible	Plausible	Plausible	77555	13323	3674	19m
fbc-5458-5459	Correct	Plausible	Plausible	1104	16	46	27m

Figure 3. Statistics of the 19 defects for which the Prophet search space contains correct patches

rithm to use random modification instead. AE [17] uses a deterministic patch search algorithm and uses program equivalence relations to prune equivalent patches during testing.

Previous work shows that, contrary to the design principle of GenProg, RSRepair, and AE, the majority of the reported patches of these three systems are implausible due to errors in the patch validation infrastructure [13]. Further semantic analysis on the remaining plausible patches reveals that despite the surface complexity of these patches, an overwhelming majority of these patches are equivalent to functionality elimination [13]. The Kali patch generation system, which only eliminates functionality, can do as well [13].

PAR: PAR [5] is another prominent automatic patch generation system. PAR is based on a set of predefined human-provided patch templates. We are unable to directly compare PAR with Prophet because, despite repeated requests to the authors of the PAR paper over the course of 11 months, the authors never provided us with the patches that PAR was reported to have generated [5]. Monperrus found that PAR fixes the majority of its benchmark defects with only two templates ("Null Pointer Checker" and "Condition Expression Adder/Remover/Replacer") [10]. In general, PAR avoids the search space explosion problem because its human supplied templates restrict its search space. However, the PAR search space (with the eight templates in the PAR paper [5]) is in fact a subset of the SPR search space [8].

JSNICE: JSNICE [14] is a JavaScript beautification tool that can automatically predicts variable names and generates comments to annotate variable types for a JavaScript program. JSNICE first learns, from a "big code" database, a probabilisitic model that captures the common relationships between the syntactic elements (e.g., the variable names) and the semantic properties (e.g., variable types and operations) of JavaScript programs. Then for a new JavaScript program, it produces a prediction that maximize the probability in the learned model.

Unlike JSNICE, the goal of Prophet is to find a correct patch for a given defect of a program, which is a deep semantic-level task to modify the program logic. Prophet therefore works with a probability model 1) that abstracts away potentially application-specific syntactic-level features and 2) that focuses on powerful application-independent semantic-level features.

ClearView: ClearView is a generate-and-validate system that observes normal executions to learn invariants that characterize safe behavior [11]. It deploys monitors that detect crashes, illegal control transfers and out of bounds write defects. In response, it selects a nearby invariant that the input that triggered the defect violates, and generates patches that take a repair action to enforce the invariant.

Failure-Oblivous Computing: Failure-oblivious computing [15] checks for out of bounds reads and writes. It discards out of bounds writes and manufactures values for out of bounds reads. This eliminates data corruption from out of bounds writes, eliminates crashes from out of bounds ac-

cesses, and enables the program to continue execution along its normal execution path.

Bolt: Bolt [6] attaches to a running application, determines if the application is in an infinite loop, and, if so, exits the loop. A user can also use Bolt to exit a long-running loop. In both cases the goal is to enable the application to continue useful execution.

RCV: RCV [9] enables applications to survive null dereference and divide by zero errors. It discards writes via null references, returns zero for reads via null references, and returns zero as the result of divides by zero. Execution continues along the normal execution path.

DieHard: DieHard [1] provides probabilistic memory safety in the presence of memory errors. In stand-alone mode, DieHard replaces the default memory manager with a memory manager that places objects randomly across a heap to reduce the possibility of memory overwrites due to buffer overflows. In replicated mode, DieHard obtains the final output of the application based on the votes of multiple replications.

APPEND: APPEND [4] proposes to eliminate null pointer exceptions in Java by applying recovery techniques such as replacing the null pointer with a pointer to an initialized instance of the appropriate class. The presented examples illustrate how this technique can effectively eliminate null pointer exceptions and enhance program survival.

Data Structure Repair: Data structure repair enables applications to recover from data structure corruption errors [3]. Data structure repair enforces a data structure consistency specification. This specification can be provided by a human developer or automatically inferred from correct program executions [2].

5. Conclusion

Generate-and-validate patch generation systems rely solely on the user supplied test suite to validate a candidate patch for a defect. This inevitably causes these systems to generate many plausible but incorrect patches. Prophet is a novel patch generation system that exploits additional information other than the test suite, i.e., the characteristics of past successful human patches, which Prophet automatically learns from a large code revision change database. Our experimental results show that, in comparison with previous systems, the learned information significantly improves the Prophet ability to generate correct patches. These results also highlight how learning characteristics of successful patches from one application can improve automatic patch generation for potentially other applications.

References

[1] E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06', pages 158–168. ACM, 2006.

[2] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. C. Rinard. Inference and enforcement of data structure consistency specifications. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 233–244, 2006.

[3] B. Demsky and M. C. Rinard. Goal-directed reasoning for specification-based data structure repair. *IEEE Trans. Software Eng.*, 32(12):931–951, 2006.

[4] K. Dobolyi and W. Weimer. Changing java's semantics for handling null pointer exceptions. In *19th International Symposium on Software Reliability Engineering (ISSRE 2008), 11-14 November 2008, Seattle/Redmond, WA, USA*, pages 47–56, 2008.

[5] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13', pages 802–811. IEEE Press, 2013.

[6] M. Kling, S. Misailovic, M. Carbin, and M. Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12', pages 431–450. ACM, 2012.

[7] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 3–13. IEEE Press, 2012.

[8] F. Long and M. Rinard. Staged Program Repair in SPR. Technical Report MIT-CSAIL-TR-2015-008, 2015.

[9] F. Long, S. Sidiroglou-Douskos, and M. Rinard. Automatic runtime error repair and containment via recovery shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14', pages 227–238, New York, NY, USA, 2014. ACM.

[10] M. Monperrus. A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 234–242, New York, NY, USA, 2014. ACM.

[11] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 87–102. ACM, 2009.

[12] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 254–265, New York, NY, USA, 2014. ACM.

[13] Z. Qi, F. Long, S. Achour, and M. Rinard. An anlysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2015*, 2015.

[14] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15', pages 111–124, New York, NY, USA, 2015. ACM.

[15] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, pages 303–316, 2004.

[16] S. Sidiroglou, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by multi-application code transfer. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2015.

[17] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE'13*, pages 356–366, 2013.

[18] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09', pages 364–374. IEEE Computer Society, 2009.