Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2015-027 July 13, 2015

Prophet: Automatic Patch Generation via Learning from Successful 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 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 database of past successful patches. Prophet defines the probabilistic model as the combination of a distribution over program points based on defect 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 previous successful patches in the database. 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. The experimental results indicate that these techniques enable Prophet to generate correct patches for 15 out of 69 real-world defects in eight open source projects. The previous state of the art generate and validate system, which uses a set of hand-code heuristics to prioritize the search, generates correct patches for 11 of these same 69 defects.

1. Introduction

We present Prophet, a new generate-and-validate patch generation system that automatically learns features of patches that successfully correct defects in existing programs. Prophet leverages the availability of large publicly available open-source software repositories that contain many successful patches. Given a database of such successful patches, Prophet learns a probabilistic model that characterizes the features that these patches exhibit. It then applies this model to recognize and prioritize correct patches within its larger set of candidate patches.

Generate-and-validate systems start with a program and a test suite of test cases, at least one of which exposes an defect in the program. They then generate a space of candidate patches and search this space to find *plausible patches* that produce correct outputs for all test cases in the test suite. Unfortunately, the presence of plausible but incorrect patches (which produce correct outputs for all of the test cases in the test suite but incorrect outputs for other inputs) has complicated the ability of previous generate-and-validate systems to find correct patches [8, 11–13, 18–20, 25].

Prophet uses its learned probabilistic patch correctness model to rank the plausible patches in its search space. The hypothesis is that successful patches share common characteristics which, if appropriately extracted and integrated into the patch generation system, will enable Prophet to automatically recognize and rank correct patches above the other plausible patches that it generates.

1.1 Prophet

Probabilistic Model: Prophet operates with a parameterized probabilistic model that, once the model parameters are determined, assigns a probability to each candidate patch in the search space. This probability indicates the likelihood that the patch is correct. The model is the product of a geometric distribution determined by the Prophet defect localization algorithm (which identifies target program statements for the generated patch to modify) and a log-linear distribution determined by the model parameters and the feature vector.

Maximum Likelihood Estimation: Given a training set of successful patches, Prophet learns the model parameters via maximizing the likelihood of observing the training set. The intuition behind this approach is that the learned model should assign a high probability to each of the successful patches in the training set.

Patch Generation: Given a program with an defect and a test suite that exposes the defect, Prophet operates as follows:

- **Defect Localization:** The Prophet defect localization algorithm analyzes execution traces of the program running on the test cases in the test suite. The result is a sequence of target program statements to patch (see Section 3.5).
- **Search Space Generation:** Prophet generates a space of candidate patches, each of which modifies one of the statements identified by the defect localization algorithm.
- **Feature Extraction:** For each candidate patch, Prophet extracts features that summarize relevant patch characteristics. These features include *program value features*, which capture relationships between how variables and constants are used in the original program and how they are used in the patch, and *modification features*, which capture relationships between the kind of program modification that the patch applies and the kinds of statements that appear near the patched statement in the original program. Prophet converts the extracted features into a binary feature vector.
- **Patch Ranking and Validation:** Prophet uses the learned model and the extracted binary feature vectors to compute a correctness probability score for each patch in the search space of candidate patches. Prophet then sorts the candidates according to their scores and validates the patches against the supplied test suite in that order. It returns the first patch that validates (i.e., produces correct outputs for all test cases in the test suite) as the result of the patch generation process.

A key challenge for Prophet is to identify and learn application-independent characteristics from successful patches. Many surface syntactic elements of successful patches (such as specific variable

names) may be application-specific and therefore may not generalize to other applications.

The Prophet program value features address this challenge as follows. Prophet uses a static analysis to obtain a set of application-independent *atomic characteristics* for each program value (i.e., variable or constant) that the patch manipulates. Each atomic characteristic captures a role that the value plays in the original or patched program (for example, a value may occur in the condition of an if statement or be returned as the value of an enclosing function).

Prophet then defines program value features that capture relationships between different roles that the same value plays in the original and patched programs. Because the program value features are derived from application-independent roles, they generalize across different applications.

To the best of our knowledge, Prophet is the first program repair system to use an automatically learned probabilistic model to identify and exploit important patch characteristics to automatically generate correct patches.

1.2 Experimental Results

We evaluate Prophet on 69 real world defects drawn from eight large open source applications. Our results show that, on the same benchmark set, Prophet automatically generates correct patches for significantly more defects than previous generate-and-validate patch generation systems, specifically SPR [12, 13], GenProg [11], and AE [25]. The Prophet search space contains correct patches for 19 defects. Prophet generates correct patches for 15 of these 19 defects. SPR, which uses a set of hand-coded heuristics to prioritize its search of the space of candidate patches, generates correct patches for 11 defects while GenProg and AE generate correct patches for 1 and 2 defects, respectively.

Our results also show that the learned model significantly improves the capability of Prophet to identify correct patches among the candidate patches in the search space. On average, Prophet prioritizes the first correct patch as one of top 11.7% in the patch prioritization order. In comparison, SPR prioritizes the first correct patch as one of the top 17.5% on average. A baseline algorithm without learning generates correct patches for only 8 defects and prioritizes the first correct patch as one of the top 20.8% on average.

Our results also highlight how program value features are critical for the success of Prophet. A variant of Prophet that disables program value features generates correct patches for only 10 defects. A common scenario is that the search space contains multiple plausible patches that manipulate different program variables. The extracted program value features often enable Prophet to identify the correct patch (which manipulates the right set of program variables) among these multiple plausible patches.

1.3 Contributions

This paper makes the following contributions:

- **Probabilistic Model:** It presents a novel parameterized probabilistic model for correct patches. This model assigns a probability to each candidate patch. This probability indicates the likelihood that the patch is correct. It also presents an algorithm that learns the model parameters via a training set of successful patches collected from open-source project repositories.
- **Feature Extraction:** It presents a novel framework for encoding program value features. Because these features successfully abstract away application-specific surface syntactic elements (such as variable names) while preserving important structural patch characteristics, they significantly improve the ability of Prophet to learn application-independent characteristics of successful patches.
- **Patch Generation with Learning:** It presents the implementation of the above techniques in the Prophet automatic patch generation system. Prophet is, to the best of our knowledge, the first automatic patch generation system that uses a machine learning algorithm to automatically learn and exploit characteristics of successful patches.
- **Experimental Results:** It presents experimental results that evaluate Prophet on 69 real world defects in eight large open source applications. Prophet generates correct patches for 15 of the 69 defects. The previous state of the art system (SPR) generates correct patches for 11 defects.

  The results show that the learned model significantly improves the ability of Prophet to identify correct patches among the candidate plausible patches and highlight how the program value features are critical for the success of Prophet — with these features disabled, Prophet generates correct patches for only 10 of the 69 defects.

The rest of this paper is organized as follows. Section 2 presents an example that illustrates how Prophet generates a patch that corrects an defect in the PHP interpreter. Section 3 presents the technical design of Prophet. Section 4 presents the experimental results. We discuss related work in Section 5 and conclude in Section 6.

2. Example

We next present an example that illustrates how Prophet corrects a defect in the PHP interpreter. The PHP interpreter (before version 5.3.5 or svn version 308315) contains a defect (PHP bug #53971) in the Zend execution engine. If a PHP program accesses a string with an out-of-bounds offset, the PHP interpreter may produce spurious runtime errors even in situations where it should suppress such errors.

Figure 1 presents (simplified) code (from the source code file $\mathrm{Zend/zend\_execute.c}$) that contains the defect. The C function at line 1 in Figure 1 implements the read operation that fetches values from a container at a given offset. The function writes these values into the data structure referenced by the first argument (result).

When a PHP program accesses a string with an offset, the second argument ($\mathrm{container\_ptr}$) of this function references the accessed string. The third argument (dim) identifies the specified offset values. The code at lines 17-18 checks whether the specified offset is within the length of the string. If not, the PHP interpreter generates a runtime error indicating an offset into an uninitialized part of a string (lines 27-29).

In some situations PHP should suppress these out-of-bounds runtime errors. Consider, for example, a PHP program that calls $\mathrm{isset(str[1000])}$. According to the PHP specification, this call should not trigger an uninitialized data error even if the length of the PHP string str is less than 1000. The purpose of $\mathrm{isset()}$ is to check if a value is properly set or not. Generating an error message when $\mathrm{isset()}$ calls the procedure in Figure 1 is invalid because it interferes with the proper operation of $\mathrm{isset()}$.

In such situations the last argument (type) at line 3 in Figure 1 is set to 3. But the implementation in Figure 1 does not properly check the value of this argument before generating an error. The result is spurious runtime errors and, depending on the PHP configuration, potential denial of service.

```
1  static void zend_fetch_dimension_address_read(
2      temp_variable *result, zval **container_ptr,
3      zval *dim, int dim_type, int type)
4  {
5    zval *container = *container_ptr;
6    zval **retval;
7    switch (Z_TYPE_P(container)) {
8      ...
9      case IS_STRING: {
10       zval tmp;
11       zval *ptr;
12       ...
13       ALLOC_ZVAL(ptr);
14       INIT_PZVAL(ptr);
15       Z_TYPE_P(ptr) = IS_STRING;
16
17       if (Z_LVAL_P(dim) < 0 ||
18           Z_STRLEN_P(container) <= Z_LVAL_P(dim)) {
19 //      A plausible but incorrect patch that validates
20 //      if (!(type == 3)) return;
21
22 //      The guard that the correct Prophet patch inserts
23 //      before the following error generation statement.
24 //      This Prophet patch is identical to the (correct)
25 //      developer patch.
26 //      if (!(type == 3))
27         zend_error(E_NOTICE,
28                    "Uninitialized string offset: %ld",
29                    (*dim).value.lval);
30       Z_STRVAL_P(ptr) = STR_EMPTY_ALLOC();
31       Z_STRLEN_P(ptr) = 0;
32     } else {
33       Z_STRVAL_P(ptr) = (char*)emalloc(2);
34       Z_STRVAL_P(ptr)[0] =
35           Z_STRVAL_P(container)[Z_LVAL_P(dim)];
36       Z_STRVAL_P(ptr)[1] = 0;
37       Z_STRLEN_P(ptr) = 1;
38     }
39     AI_SET_PTR(result, ptr);
40     return;
41   } break;
42   ...
43 }
44 }
```

Figure 1. Simplified Code for PHP bug #53971

Offline Learning: Prophet works with a database of previous successful human patches to obtain a probabilistic model that captures why these patches were successful. We obtain this database by collecting revision changes from open source repositories. In our example, we train Prophet with revision changes from seven open source projects (apr, curl, Apache httpd, libtiff, python, subversion, and wireshark). Although revision changes for PHP are available, we exclude these revision changes from this training set. During the offline learning phase, Prophet performs the following steps:

- **Extract Features:** For each patch in the database, Prophet analyzes a structural diff on the abstract syntax trees of the original and patched code to extract both 1) features which summarize how the patch modifies the program given characteristics of the surrounding code and 2) features which summarize relationships between roles that values accessed by the patch play in the original unpatched program and in the patch.
- **Learn Model Parameters:** Prophet operates with a parameterized log-linear probabilistic model in which the model parameters can be interpreted as the weights that capture the importance of different features. Prophet learns the model parameters via maximum likelihood estimation, i.e., the Prophet learning algorithm attempts to find parameter values that maximize the probability of observing the collected training database in the probabilistic model.

Apply Prophet: We then apply Prophet to automatically generate a patch for this defect. Specifically, we provide Prophet with the PHP source code that contains the defect and a test suite that contains 6957 test cases. One of the test cases exposes the defect (i.e., the unpatched version of PHP produces incorrect output for this test case). The remaining 6956 test cases are to prevent regression (the unpatched version of PHP produces correct outputs for these test cases). Prophet generates a patch with the following steps:

- **Defect Localization:** Prophet first performs a dynamic analysis of the execution traces of the PHP interpreter on the supplied test suite to identify a set of candidate program points for the patch to modify. In our example, the Prophet defect localization algorithm observes that the negative test case executes the statement at lines 27-29 in Figure 1 while the positive test cases rarely execute this statement. Prophet therefore generates candidate patches that modify this statement (as well as candidate patches that modify other statements).
- **Search Space Generation:** Prophet works with the SPR search space [12, 13], which uses *transformation schemas* and *staged condition synthesis* to generate candidate patches. Some (but by no means all) of these candidate patches add an if statement to guard (conditionally execute) the statement at lines 27-29 in Figure 1.
- **Rank Candidate Patches:** Prophet computes a feature vector for each candidate patch in the search space. It then applies the learned model to the computed feature vector to obtain a probability that the corresponding patch is correct. It then ranks the generated patches according to the computed correctness probabilities.

  In our example, the model assigns a relatively high correctness probability to the candidate patch (line 26 in Figure 1) that adds an if statement guard with condition $(\text{type} \mathrel{!}= 3)$ (this condition was synthesized by the Prophet condition synthesis algorithm [12, 13]) before the statement that generates the error message (lines 27-29 in Figure 1). This patch has several features that correspond in the learned model to correct patches. For example, 1) it adds an if condition to guard a call statement and 2) the guard condition checks a supplied argument of the function.
- **Validate Candidate Patches:** Prophet then uses the test suite to attempt to validate the patches in order of highest patch correctness probability. The patch shown at line 26 in Figure 1 is the first patch to validate (i.e., it is the first generated patch that produces correct outputs for all of the test cases in the test suite).

The generated Prophet patch is correct and identical to the developer patch for this defect. Note that the Prophet search space may contain incorrect patches that nevertheless validate (because they produce correct outputs for all test cases in the test suite). In our example, line 20 in Figure 1 presents one such patch. This patch directly returns from the function if $\text{type} \mathrel{!}= 3$. This patch is incorrect because it does not properly set the result data structure (referenced by the result argument) before it returns from the function. Because the negative test case does not check this result data structure, this incorrect patch nevertheless validates. In our example the Prophet learning algorithm successfully ranks such plausible but incorrect patches lower than the correct patch.

3. Design

Prophet first performs an offline training phase to learn a probabilistic model which summarizes important features of successful patches drawn from a large revision database. Given a new defective program p, Prophet generates a search space of candidate patches for p and uses the learned model to recognize and prioritize correct patches. In this way the learned knowledge guides the exploration of the patch search space.

3.1 Probabilistic Model

Given a defective program p and a search space of candidate patches, the Prophet probabilistic model is a parameterized likelihood function which assigns each candidate patch δ a probability $P(\delta \mid p, \theta)$, which indicates how likely δ is a correct patch for p. θ is the model parameter vector which Prophet learns during its offline training phase (see Section 3.2). Once θ is determined, the probability can be interpreted as a normalized score (i.e., $\sum_\delta P(\delta \mid p) = 1$) which prioritizes potentially correct patches among all possible candidate patches.

The Prophet probabilistic model assumes that each candidate patch δ in the search space can be derived from the given defective program p in two steps: 1) Prophet selects a program point $\ell \in L(p)$, where $L(p)$ denotes the set of program points in p that Prophet may attempt to modify; 2) Prophet selects an AST modification operation $m \in M(p, \ell)$ and applies m at ℓ to obtain δ, where $M(p, \ell)$ denotes the set of all possible modification operations that Prophet may attempt to apply at ℓ. Therefore the patch δ is a pair $\langle m, \ell \rangle$.

Based on this assumption, Prophet factors the probability $P(\delta \mid p, \theta)$ as follows:

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

$P(m \mid p, \ell, \theta)$ is a distribution that corresponds to the probability of applying the modification operation m given p and ℓ. Prophet defines $P(m \mid p, \ell, \theta)$ as a parameterized log-linear distribution,

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

where $\phi(p, \ell, m)$ is the feature vector that Prophet extracts from the triple of p, ℓ, and m (see Section 3.3).

$P(\ell \mid p, \theta)$ is a distribution that corresponds to the probability of modifying the program point ℓ given the defective program p. Prophet defines $P(\ell \mid p, \theta)$ as follows (Z is the normalization divisor):

$$P(\ell \mid p, \theta) = \frac{1}{Z} \cdot A \cdot B$$

$$A = (1 - \beta)^{r(p,\ell)}$$

$$B = \frac{\sum_{m' \in M(p,\ell)} \exp\left(\phi(p, m', \ell) \cdot \theta\right)}{\sum_{\ell' \in L(p)} \sum_{m' \in M(p,\ell')} \exp\left(\phi(p, m', \ell') \cdot \theta\right)}$$

The part A is a geometric distribution that encodes the information Prophet obtains from its defect localization algorithm (which identifies target program points to patch). The algorithm performs a dynamic analysis on the execution traces of the program p on the supplied test suite to obtain a ranked list of potential program points to modify (see Section 3.5). $r(p, \ell)$ denotes the rank of ℓ assigned by the defect localization algorithm. If ℓ is not in the Prophet search space (i.e., $\ell \notin L(p)$), then $r(p, \ell) = \infty$. β is the probability of each coin flip trial of the geometric distribution (which Prophet empirically sets to 0.02). The part B is a parameterized log-linear distribution determined by the extracted feature vectors ϕ and the learned parameter vector θ.

$P(\delta \mid p, \theta)$ is the product of $P(\ell \mid p, \theta)$ and $P(m \mid p, \ell, \theta)$:

$$\frac{1}{Z} \cdot (1 - \beta)^{r(p,\ell)} \cdot \frac{\exp\left(\phi(p, m, \ell) \cdot \theta\right)}{\sum_{\ell' \in L(p)} \sum_{m' \in M(p,\ell')} \exp\left(\phi(p, m', \ell') \cdot \theta\right)}$$

Intuitively, this formula assigns the weight $e^{\phi(p,m,\ell)\cdot\theta}$ to each candidate patch $\langle m, \ell \rangle$ based on the extracted feature vector $\phi(p, m, \ell)$ and the learned parameter vector θ. The formula then computes the weight proportion of each patch over the total weight of the entire search space derived from the functions L and M. The formula obtains the final patch probability by multiplying the weight proportion of each patch with a geometric distribution probability, which encodes the defect localization ranking of the patch.

Note that $L(p)$, $r(p, \ell)$, and $M(p, \ell)$ are inputs to the probabilistic model. $M(p, \ell)$ defines the patch search space while $L(p)$ and $r(p, \ell)$ define the defect localization algorithm. The model can work with arbitrary $L(p)$, $r(p, \ell)$, and $M(p, \ell)$, i.e., it is independent of the underlying search space and the defect localization algorithm. It is straightforward to extend the Prophet model to work with patches that modify multiple program points.

3.2 Learning Algorithm

The input to the Prophet training phase is a large revision change database $D = \{\langle p_1, \delta_1 \rangle, \ldots, \langle p_n, \delta_n \rangle\}$, where each element of D is a pair of a defective program p_i and the corresponding successful human patch δ_i. Prophet learns a model parameter θ such that the result probabilistic model assigns a high conditional probability score to δ_i among all possible candidate patches in the search space. Specifically, Prophet learns θ via maximizing the log likelihood of observing the training database D:

$$\theta = \arg\max_\theta \left(\sum_i \log P(\delta_i | p_i, \theta) + \lambda_1 \sum_i |\theta_i| + \lambda_2 \sum_i \theta_i^2 \right)$$

where λ_1 and λ_2 are L1 and L2 regularization factors which Prophet uses to avoid overfitting. Prophet empirically sets both factors to 10^{-3}.

Note that the training database may not contain test suites for each defective program p_i in D. Prophet therefore cannot use its defect localization algorithm (which requires test cases to drive the dynamic analysis) to compute $L(p_i)$ (i.e., the set of candidate program points to modify) or $r(p, \ell)$ (i.e., the rank of each program point ℓ).

The Prophet learning algorithm therefore uses an oracle-like defect localization algorithm to drive the training. For each training pair $\langle p_i, \delta_i \rangle$, the algorithm computes the structural AST difference that the patch δ_i induces to 1) locate the modified program location ℓ_i and 2) identify a set of program points S_i near ℓ_i (i.e., in the same basic block as ℓ_i and within three statements of ℓ_i in this basic block). It then sets $L(p_i) = S_i$ with $r(p_i, \ell_i') = 1$ for all $\ell_i' \in L(p_i)$. We therefore simplify our objective formula by removing the geometric distribution part, which is constant during the training phase:

$$\theta = \arg\max_\theta \left(\sum_i \log g(p, \ell, m, \theta) + \lambda_1 \sum_i |\theta_i| + \lambda_2 \sum_i \theta_i^2 \right)$$

$$g(p, \ell, m, \theta) = \frac{\exp\left(\phi(p, m, \ell) \cdot \theta\right)}{\sum_{\ell' \in L(p)} \sum_{m' \in M(p,\ell')} \exp\left(\phi(p, m', \ell') \cdot \theta\right)}$$

Figure 2 presents the Prophet learning algorithm. Combining standard machine learning techniques, Prophet computes θ via gradient descent as follows:

Input : the training database $D = \{\langle p_1, \delta_1\rangle, \ldots, \langle p_n, \delta_n\rangle\}$, where p_i is the original program and δ_i is the successful human patch for p_i.
Output: the feature weight parameter vector θ.

1 **for** $i = 1$ **to** n **do**
2 $\quad \langle m_i, \ell_i\rangle \longleftarrow \delta_i$
3 $\quad L_i \longleftarrow \text{NearLocations}(p_i, \ell_i)$
4 $n_0 \longleftarrow 0.85 \cdot n$
5 Initialize all elements in θ to 0
6 $\theta^* \longleftarrow \theta$
7 $\alpha \longleftarrow 1$
8 $\gamma^* \longleftarrow 1$
9 $cnt \longleftarrow 0$
10 **while** $cnt < 200$ **do**
11 $\quad$ Assume $g(p, \ell, m, L, \theta) = e^{\phi(p,m,\ell)\cdot\theta} / (\Sigma_{\ell' \in L} \Sigma_{m' \in M(p,\ell')} e^{\phi(p,m',\ell')\cdot\theta})$
12 $\quad$ Assume $f(\theta) = \frac{1}{n_0} \cdot \Sigma_{i=1}^{n_0} \log g(p_i, \ell_i, m_i, L_i, \theta) + \lambda_1 \cdot \Sigma_{i=1}^{k} |\theta_i| + \lambda_2 |\theta|^2$
13 $\quad \theta \longleftarrow \theta + \alpha \cdot \frac{\partial f}{\partial \theta}$
14 $\quad \gamma \longleftarrow 0$
15 $\quad$ **for** $i = n_0 + 1$ **to** n **do**
16 $\quad\quad tot \longleftarrow |\{m \mid m \in M(p_i, \ell),\ \ell \in L_i\}|$
17 $\quad\quad rank \longleftarrow |\{m \mid m \in M(p_i, \ell),\ \ell \in L_i,$ $g(p_i, \ell, m, L_i, \theta) \geq g(p_i, \ell_i, m_i, L_i, \theta)\}|$
18 $\quad\quad \gamma \longleftarrow \gamma + (rank/tot)/(n - n_0)$
19 $\quad$ **if** $\gamma < \gamma^*$ **then**
20 $\quad\quad \theta^* \longleftarrow \theta$
21 $\quad\quad \gamma^* \longleftarrow \gamma$
22 $\quad\quad cnt \longleftarrow 0$
23 $\quad$ **else**
24 $\quad\quad cnt \longleftarrow cnt + 1$
25 $\quad\quad$ **if** $\alpha > 0.01$ **then**
26 $\quad\quad\quad \alpha \longleftarrow 0.9 \cdot \alpha$
27 **return** θ^*

Figure 2. Learning Algorithm

- **AST Structural Difference:** For each pair $\langle p_i, \delta_i\rangle$ in D, Prophet computes the AST structural difference of δ_i to obtain the corresponding modification operation m_i and the modified program point ℓ_i (lines 1-3). The function $\text{NearLocations}(p_i, \ell_i)$ at line 3 returns a set of program points that are close to the known correct modification point ℓ_i. Prophet uses the returned set as $L(p_i)$ to drive the learning.
- **Initialization:** Prophet initializes θ with all zeros. Prophet also initializes the learning rate of the gradient descent (α at line 7) to one. At line 4, Prophet splits the training set and reserves 15% of the training pairs as a validation set. Prophet uses this validation set to measure the performance of the learning process and avoid overfitting. Prophet uses the remaining 85% of the training pairs to perform the gradient descent computation.
- **Update Current θ:** Prophet runs an iterative gradient descent algorithm. Prophet updates θ at lines 11-13 at the start of each iteration.
- **Measure Performance:** For each pair of $\langle p_i, \delta_i\rangle$ in the validation set, Prophet computes the percentage of candidate programs in the search space that have a higher probability score than δ_i (lines 15-18). Prophet uses the average percentage (γ) over all of the validation pairs to measure the performance of the current θ. Lower percentage is better because it indicates that the learned model ranks correct patches higher among all candidate patches.
- **Update Best θ and Termination:** θ^* in Figure 2 corresponds to the best observed θ. At each iteration, Prophet updates θ^* at lines 19-22 if the performance (γ) of the current θ on the validation set is better than the best previously observed performance (γ^*). Prophet decreases the learning rate α at lines 25-26 if θ^* is not updated. If it does not update θ^* for 200 iterations, the Prophet learning algorithm terminates and returns θ^* as the result.

$$
\begin{array}{lll}
c & := & c_1 \texttt{ \&\& } c_2 \mid c_1 \texttt{ || } c_2 \mid v \texttt{!=} const \mid v \texttt{==} const \\
simps & := & v = v_1 \texttt{ op } v_2 \mid v = const \mid \texttt{print } v \\
 & & \mid \texttt{skip} \mid \texttt{break} \\
s & := & \ell\texttt{: } simps \mid \{\ s_1\ s_2\ \ldots\ \} \mid \ell\texttt{: if (}c\texttt{) } s_1\ s_2 \\
 & & \mid \ell\texttt{: while (}c\texttt{) } s_1 \\
p & := & \{\ s_1\ s_2\ \ldots\ \} \\
\multicolumn{3}{l}{v,\ v_1,\ v_2 \in \textbf{Var} \quad const \in \textbf{Int} \quad \ell \in \textbf{Label}} \\
\multicolumn{3}{l}{c,\ c_1,\ c_2 \in \textbf{Cond} \quad s,\ s_1,\ s_2 \in \textbf{Stmt}} \\
\multicolumn{3}{l}{p \in \textbf{Prog} \qquad \textbf{Atom} = \textbf{Var} \cup \textbf{Int}}
\end{array}
$$

Figure 3. The language statement syntax

$$
\begin{array}{ll}
\textbf{Patch} = \textbf{Modification} \times \textbf{Label} & \textbf{Pos} = \{\texttt{C}, \texttt{P}, \texttt{N}\} \\
\textbf{MK} = \{\texttt{InsertControl}, \texttt{InsertGuard}, \texttt{ReplaceCond}, & \\
\quad \texttt{ReplaceStmt}, \texttt{InsertStmt}\} & \\
\textbf{SK} = \{\texttt{Assign}, \texttt{Print}, \texttt{While}, \texttt{Break}, \texttt{Skip}, \texttt{If}\} & \\
\textbf{ModFeature} = \textbf{MK} \cup (\textbf{Pos} \times \textbf{SK} \times \textbf{MK}) & \\
\textbf{ValueFeature} = \textbf{Pos} \times \textbf{AC} \times \textbf{AC} & \\
\text{Stmt} : & \textbf{Prog} \times \textbf{Label} \to \textbf{Stmt} \\
\text{ApplyPatch} : & \textbf{Prog} \times \textbf{Patch} \to \textbf{Prog} \times (\textbf{Cond} \cup \textbf{Stmt}) \\
\text{ModKind} : & \textbf{Modification} \to \textbf{MK} \\
\text{StmtKind} : & \textbf{Stmt} \to \textbf{SK} \\
\psi : & \textbf{Prog} \times \textbf{Atom} \times (\textbf{Cond} \cup \textbf{Stmt}) \to \textbf{AC} \\
\text{FIdx} : & (\textbf{ModFeature} \cup \textbf{ValueFeature}) \to \textbf{Int} \\
 & \forall a, \forall b, (FIdx(a) = FIdx(b)) \iff (a = b)
\end{array}
$$

Figure 4. Definitions and notation. **SK** corresponds to the set of statement kinds. **MK** corresponds to the set of modification kinds. **AC** corresponds to the set of atomic characteristics that the analysis function ψ extracts.

3.3 Feature Selection

Figure 3 presents the syntax of a simple programming language which we use to illustrate the Prophet feature extraction algorithm. See Section 3.5 for the implementation details of extending this algorithm to C programs. Each of the statements (except compound statements) is associated with a unique label ℓ. A program p in the language corresponds to a compound statement. The semantics of the language in Figure 3 is similar to C. We omit the operational semantics details for brevity.

Figure 4 presents the notation we use to present our feature extraction algorithm. Figure 5 presents our feature extraction algorithm. Given a program p, a program point ℓ, and a modification operation m that is applied at ℓ, Prophet extracts features by analyzing both m and the original code near ℓ.

Prophet first partitions the statements near ℓ in the original program p into three sets S_{C}, S_{P}, and S_{N} based on the relative positions of the statements (lines 1-3). S_{C} contains only the statement associated with the modification point ℓ (returned by the utility function Stmt). S_{P} contains the statements that appear at most three state-

```
Input  : the input program p, the modified program point ℓ, and the
         modification operation m
Output: the extracted feature vector φ(p, ℓ, m)
1  Initialize all elements in φ to 0
2  S_C ⟵ {Stmt(p, ℓ)}
3  S_P ⟵ Prev3stmts(p, ℓ))
4  S_N ⟵ Next3stmts(p, ℓ))
5  idx ⟵ FIdx(ModKind(m))
6  φ_idx ⟵ 1
7  for i in {C, P, N} do
8      for s in S_i do
9          idx ⟵ Fid(⟨i, StmtKind(s), ModKind(m)⟩)
10         φ_idx ⟵ 1
11 ⟨p', n⟩ ⟵ ApplyPatch(p, ⟨m, ℓ⟩)
12 for i in {C, P, N} do
13     for a in Atoms(n) do
14         for s in S_i do
15             for ac' in ψ(p', a, n) do
16                 for ac in ψ(p, a, s) do
17                     idx ⟵ FIdx(⟨i, ac, ac'⟩)
18                     φ_idx ⟵ 1
19 return φ
```

Figure 5. Feature Extraction Algorithm

ments before ℓ in the enclosing compound statement (returned by the utility function $\mathrm{Prev3stmts}$). S_{N} contains the statements that appear at most three statements after ℓ in the enclosing compound statement (returned by the utility function $\mathrm{Next3stmts}$).

Prophet then extracts two types of features, modification features (lines 5-10) and program value features (lines 11-18). Modification features capture relationships between the modification m and the surrounding statements, while program value features characterize how the modification works with program values (i.e., variables and constants) in the original and patched code. For each extracted feature, Prophet sets the corresponding bit in θ whose index is identified by the utility function FIdx (lines 5-6, lines 9-10, and lines 17-18). FIdx maps each individual feature to a unique integer value.

Modification Features: There are two possible forms of modification features. The first is simply the kind of modification that m applies. The second are designed to capture relationships between the kinds of statements that appear near the patched statement in the original program and the modification kind of m. So, for example, if successful patches often insert a guard condition before a call statement, a modification feature will enable Prophet to recognize and exploit this fact.

At lines 5-6 in Figure 5, Prophet extracts the modification kind of m as the modification feature. At lines 7-10, Prophet also extracts the triple of the position of an original statement, the kind of the original statement, and the modification kind of m as the modification feature. At line 9, the utility function $\mathrm{StmtKind}(s)$ returns the statement kind of s and the utility function $\mathrm{ModKind}(m)$ returns the modification kind of m.

Prophet currently classifies modification operations into five kinds: `InsertControl` (inserting a potentially guarded control statement before a program point), `AddGuardCond` (adding a guard condition to an existing statement), `ReplaceCond` (replacing a branch condition), `InsertStmt` (inserting a non-control statement before a program point), and `ReplaceStmt` (replacing an existing statement). See Figure 4 for the definition of modification features, statement kinds, and modification kinds.

Program Value Features: Program value features are designed to capture relationships between how variables and constants are used in the original program and how they are used in the patch. For example, if successful patches often insert a check involving a variable that is subsequently passed as a parameter to a subsequent call statement, a program value feature will enable Prophet to recognize and exploit this fact. Program value features relate occurrences of the same variable or constant in the original and patched programs.

To avoid polluting the feature space with surface-level application-specific information, program value features abstract away the specific names of variables and values of constants involved in the relationships that these features model. This abstraction enables Prophet to learn program value features from one application and then apply the learned knowledge to another application.

To extract the features, Prophet first applies the patch to the original program at line 11 in Figure 5. $\mathrm{ApplyPatch}(p, \langle m, \ell \rangle)$ denotes the results of the patch application, which produces a pair $\langle p', n$, where p' is the new patched program and n is the AST node for the new statement or condition that the patch introduces. Prophet then performs a static analysis on both the repaired program and the original program to extract a set of atomic characteristics for each program atom a (i.e., a variable or an integer). In Figure 5, $\psi(p, a, n)$ denotes the set of atomic characteristics extracted for a in n.

At lines 12-18, Prophet extracts each program value feature, which is a triple $\langle i, ac, ac' \rangle$ of the position i of a statement in the original program, an atomic characteristic ac of a program atom in the original statement, and an atomic characteristic ac' of the same program atom in the AST node that the patch introduces. Intuitively, the program value features track co-occurrences of each pair of the atomic characteristic ac in the original code and the atomic characteristic ac' in the modification m. The utility function $\mathrm{Atoms}(n)$ at line 12 returns a set that contains all program atoms (i.e., program variables and constants) in n.

Figure 6 presents the static analysis rules that Prophet uses to extract atomic characteristics $\psi(p, v, n)$. These rules track the roles that v plays in the enclosing statements or conditions and record the operations in which v participates. Note that Prophet can work with any static analysis to extract arbitrary atomic characteristics. It is therefore possible, for example, to combine Prophet with more sophisticated analysis algorithms to obtain a richer set of atomic characteristics.

3.4 Repair Algorithm

Given a program p that contains a defect, the goal of Prophet is to find a correct patch δ that eliminates the defect and correctly preserves the other functionality of p. We use an oracle function Oracle to define patch correctness, specifically $\mathrm{Oracle}(p, \delta) =$ `true` if and only if δ correctly patches the defect in p.

Note that Oracle is hidden. Instead, Prophet assumes that the user provides a test suite which exposes the defect in the original program p. We use the test suite to obtain an approximate oracle T such that $\mathrm{Oracle}(p, \delta)$ implies $T(p, \delta)$. Specifically, $T(p, \delta) =$ `true` if and only if the patched program passes the test suite, i.e., produces correct outputs for all test cases in the test suite.

Repair Algorithm: Figure 7 presents the Prophet repair algorithm. Prophet generates a search space of candidate patches and uses the learned probabilistic model to prioritize potentially correct patches. Specifically, Prophet performs the following steps:

$\psi : \mathbf{Prog} \times \mathbf{Atom} \times (\mathbf{Cond} \cup \mathbf{Stmt}) \to \mathbf{AC}$ $\qquad \psi(p, a, node) = \psi_0(a, node) \cup \psi_1(a, node)$

$\mathbf{AC} = \{\texttt{var}, \texttt{const0}, \texttt{constn0}, \texttt{cond}, \texttt{if}, \texttt{prt}, \texttt{loop}, ==, \texttt{!=}\langle \text{op}, \texttt{L}\rangle, \langle \text{op}, \texttt{R}\rangle, \langle =, \texttt{L}\rangle, \langle =, \texttt{R}\rangle\}$

$$\frac{v \in \mathbf{Var}}{\psi_0(v, node) = \{\texttt{var}\}} \qquad \frac{const = 0}{\psi_0(const, node) = \{\texttt{const0}\}} \qquad \frac{const \in \mathbf{Int} \qquad const \neq 0}{\psi_0(const, node) = \{\texttt{constn0}\}} \qquad \frac{a \notin \text{Atoms}(node)}{\psi_1(a, node) = \emptyset}$$

$$\frac{c = \text{“}v{==}const\text{”}}{\psi_1(v, c) = \{\texttt{cond}, ==\} \qquad \psi_1(const, c) = \{\texttt{cond}, ==\}} \qquad \frac{c = \text{“}v\,\texttt{!=}const\text{”}}{\psi_1(v, c) = \{\texttt{cond}, \texttt{!=}\} \qquad \psi_1(const, c) = \{\texttt{cond}, \texttt{!=}\}}$$

$$\frac{c = \text{“}c_1 \;\; \texttt{\&\&} \;\; c_2\text{”} \text{ or } c = \text{“}c_1 \;\; \texttt{||} \;\; c_2\text{”} \qquad a \in \text{Atoms}(c)}{\psi_1(a, c) = \psi_1(a, c_1) \cup \psi_1(a, c_2)} \qquad \frac{s = \text{“}\ell : v = v_1 \text{ op } v_2\text{”}}{\psi_1(v, s) = \{\langle =, \texttt{L}\rangle\} \qquad \psi_1(v_1, s) = \{\langle \text{op}, \texttt{L}\rangle, \langle =, \texttt{R}\rangle\} \qquad \psi_1(v_2, s) = \{\langle \text{op}, \texttt{R}\rangle, \langle =, \texttt{R}\rangle\}}$$

$$\frac{s = \text{“}\ell : v{=}const\text{”}}{\psi_1(v, s) = \{\langle =, \texttt{L}\rangle\} \qquad \psi_1(const, s) = \{\langle =, \texttt{R}\rangle\}} \qquad \frac{s = \text{“}\ell : \texttt{print} \;\; v\text{”}}{\psi_1(v, s) = \{\texttt{prt}\}} \qquad \frac{s = \text{“}\ell : \texttt{while} \;\; (c) \;\; s_1\text{”} \qquad a \in \text{Atoms}(s)}{\psi_1(a, s) = \psi_1(a, c) \cup \psi_1(a, s_1) \cup \{\texttt{loop}\}}$$

$$\frac{s = \text{“}\{s_1, s_2, \ldots\}\text{”} \qquad a \in Atoms(s)}{\psi_1(a, s) = \psi_1(a, s_1) \cup \psi_1(a, s_2) \cup \cdots} \qquad \frac{s = \text{“}\ell : \texttt{if} \;\; (c) \;\; s_1 \;\; s_2\text{”} \qquad a \in \text{Atoms}(s)}{\psi_1(a, s) = \psi_1(a, c) \cup \psi_1(a, s_1) \cup \psi_1(v, s_2) \cup \{\texttt{if}\}}$$

Figure 6. Atomic Characteristic Extraction Rules for $\psi(p, a, n)$

Input : the original program p, the test suite T and the learned model parameter vector θ
Output: the generated patch δ or NULL is failed

1 $L \longleftarrow \text{DefectLocalizer}(p, T)$
2 $Candidates \longleftarrow \emptyset$
3 **for** $\langle \ell, rank \rangle$ **in** L **do**
4 **for** m **in** $M(p, \ell)$ **do**
5 $prob\_score \longleftarrow (1 - \beta)^{rank-1} \cdot \beta \cdot e^{\phi(p,\ell,m)\cdot\theta}$
6 $Candidates \longleftarrow Candidates \cup \{\langle prob\_score, m, \ell \rangle\}$
7 $SortedCands \longleftarrow \text{SortWithFirstElement}(Candidates)$
8 **for** $\langle \_, m, \ell \rangle$ **in** $SortedCands$ **do**
9 $\delta \longleftarrow \langle m, \ell, \rangle$
10 **if** $T(p, \delta) = true$ **then**
11 **return** δ
12 **return** NULL

Figure 7. Prophet Repair Algorithm

- **Generate Search Space:** At line 1, Prophet runs the defect localization algorithm (DefectLocalizer()) to return a ranked list of candidate program points to modify. At lines 2-6, Prophet then generates a search space that contains candidate patches for all of the candidate program points.
- **Rank Candidate Patch:** At lines 5-6, Prophet uses the learned θ to compute the probability score for each candidate patch. At line 7, Prophet sorts all candidate patches in the search space based on their probability score. Note that the score formula at line 5 omits the constant divisor from the formula of $P(\delta|p)$, because it does not affect the sort results.
- **Validate Candidate Patch:** At lines 8-11, Prophet finally tests all of the candidate patches one by one in the sorted order with the supplied test suite (i.e., T). Prophet outputs the first candidate patch that passes the test suite.

3.5 Implementation

We implement Prophet on top of SPR [12, 13], a previous patch generation system.

Defect Localization: The Prophet defect localizer recompiles the given application with additional instrumentation. It inserts a call back before each statement in the source code to record a positive counter value as the timestamp of the statement execution. Prophet then invokes the recompiled application on all test cases.

For a statement s and a test case i, $t(s, i)$ is the recorded execution timestamp that corresponds to the last timestamp from an execution of the statement s when the application runs with the test case i. If the statement s is not executed at all when the application runs with the test case i, then $t(s, i) = 0$.

We use the notation NegT for the set of negative test cases that expose the defect of the program and PosT for the set of positive test cases that the original program already passes. Prophet computes three scores $a(s)$, $b(s)$, $c(s)$ for each statement s:

$$a(s) = |\ \{i \ \mid \ t(s, i) \neq 0,\ i \in \text{NegT}\}\ |$$
$$b(s) = |\ \{i \ \mid \ t(s, i) = 0,\ i \in \text{PosT}\}\ |$$
$$c(s) = \Sigma_{i \in \text{NegT}} t(s, i)$$

A statement s_1 has higher priority than a statement s_2 if $\text{prior}(s_1, s_2) = 1$, where prior is defined as:

$$prior(s_1, s_2) = \begin{cases} 1 & a(s_1) > a(s_2) \\ 1 & a(s_1) = a(s_2), b(s_1) > b(s_2) \\ 1 & a(s_1) = a(s_2), b(s_1) = b(s_2), \\ & c(s_1) > c(s_2) \\ 0 & \text{otherwise} \end{cases}$$

Intuitively, Prophet prioritizes statements that 1) are executed with more negative test cases, 2) are executed with fewer positive test cases, and 3) are executed later during executions with negative test cases. Prophet considers the first 200 statements as potential statements for modification.

Note that the probabilistic model and the repair algorithm are independent from the defect localization component. Prophet can integrate with any defect localization technique that returns a ranked list of target program points to patch. It is therefore possible to combine Prophet with other (potentially more accurate) defect localization techniques [2, 7, 27].

Search Space: The Prophet probabilistic model can work with any search space of candidate patches. The current implementation of Prophet operates on the same search space as SPR. This search space is derived from a set of parameterized transformation schemas that Prophet applies to target statements identified by the defect localization algorithm [12, 13]. These schemas generate patches that either 1) (Tighten) tighten the condition of a target if statement (by conjoining a condition C to the if condition), 2) (Loosen) loosen the condition of a target if statement (by disjoin-

Commutative Operators	Is an operand of +, *, ==, or !=
Binary Operators	Is a left/right operand of −, /, <, >, <=, >=, . (field access), −> (member access), or [] (index)
Unary Operators	Is an operand of −, ++ (increment), −− (decrement), * (dereference), or & (address-taken)
Enclosing Statements	Occurs in an assign/loop/return/if statement Occurs in a branch condition Is a function call parameter Is the callee of a call statement
Value Traits	Is a local variable, global variable, argument, struct field, constant, non-zero constant, zero, or constant string literal Has an integer, pointer, or struct pointer type Is dereferenced
Patch Related	Is inside an abstract expression Is replaced by the modification operation

Figure 8. Atomic Characteristics of Program Values for C

ing a condition C to the if condition), 3) (Add Guard) add a guard with a condition C to a target statement, 4) (Insert Guarded Control Flow) insert a new guarded control flow statement (if (C) return; if (C) break; or if (C) goto l; where l is an existing label in the program and C is the condition that the guard enforces) before the target statement, 5) (Initialize) insert a memory initialization statement before the target statement, 6) (Replace) replace one value in the target statement with another value, or 7) (Copy and Replace) copy an existing statement before the target statement and replace a value in the copied statement with another value. Prophet uses *condition synthesis* [12, 13] to synthesize an appropriate condition C.

SPR uses a set of hand-coded heuristics to prioritize its search of the generated patch space. These heuristics prioritize patches in the following order: Tighten, Loosen, Insert Guarded Control Flow before the first statement in a compound statement (i.e., a C code block), Add Guard, Initialization, Insert Guarded Control Flow before a statement that is not the first statement in a compound statement, Replace or Copy and Replace before the first statement in a compound statement, and finally all other patches. For each kind of patch, it prioritizes statements to patch in the defect localization order. So SPR first tests all Tighten patches on all target statements that the defect localizer identifies, then all Loosen patches on all identified target statements, and so on. Instead of these heuristics, Prophet uses its learned probabilistic patch correctness model to prioritize its search of the generated patch space.

Feature Extraction for C: Prophet extends the feature extraction algorithm described in Section 3.3 to C programs as follows. Prophet treats call expressions in C as a special statement kind for feature extraction. Prophet extracts atomic characteristics for binary and unary operations in C. For each variable v, Prophet also extracts atomic characteristics that capture the scope of the variable (e.g., global or local) and the type of the variable (e.g., integer, pointer, pointer to structure). The current Prophet implementation tracks over 30 atomic characteristics (see Figure 8 for a list of these atomic characteristics) and works with a total of 3515 features, including 455 modification features and 3060 program value features.

Features for Abstract Expressions: For certain kinds of candidate patches, the SPR staged program repair algorithm generates candidate patch templates with abstract expressions. SPR first validates each patch template. It generates concrete patches from the template only if the patch template validation determines that there may be a concrete patch that passes the supplied test case. This optimization significantly reduces the number of concrete patches the system attempts to validate.

To integrate with the staged program repair technique, Prophet extends its probabilistic model and the learning algorithm to operate on candidate patch templates which may contain an abstract expression. The templates only specify the variables in these abstract expressions but do not determine the concrete forms of these expressions. Prophet treats abstract expressions as a special type of AST node for feature extraction. Prophet also extracts atomic characteristics to indicate whether program variables are inside an abstract expression in a patch template (see Figure 8).

Project	Revisions Used for Training
apr	12
curl	53
httpd	75
libtiff	11
php	187
python	114
subversion	240
wireshark	85
Total	777

Figure 9. Statistics of Collected Developer Patch Database

4. Experimental Results

We evaluate Prophet on 69 real world defects in eight large open source applications: libtiff, lighttpd, the PHP interpreter, gmp, gzip, python, wireshark, and fbc. These defects are from the same benchmark set used to evaluate GenProg, AE, and SPR [11–13, 25]. For each defect, the benchmark set contains a test suite with positive test cases (for which the unpatched program produces correct outputs) and at least one negative test case that exposes the defect (the unpatched program produces incorrect outputs for the negative test cases).

Note that this benchmark set is reported to contain 105 defects [11]. An examination of the revision changes and corresponding check in entries indicates that 36 of these reported defects are not, in fact, defects. They are instead deliberate functionality changes [12, 13]. Because there is no defect to correct, they are therefore outside the scope of Prophet. We nevertheless also report results for Prophet on these functionality changes.

4.1 Methodology

Collect Successful Human Patches: We first collect 777 successful human patches from eight open source project repositories. Figure 9 presents statistics for the collected patch database. These 777 patches include all patches in these eight project repositories that 1) compile in our environment (this is a requirement for applying the Prophet learning algorithm, which operates on abstract syntax trees), 2) are within the Prophet search space, and 3) (as indicated by an automated analysis of the check-in entries) repair defects (as opposed, for example, to adding new functionality).

Train Prophet with Collected Database: We train Prophet with the collected database of successful human patches. Note that our collected code database and our benchmark set share four common applications, specifically libtiff, php, python, and wireshark. For

App	LoC	Tests	Defects/ Changes	Plausible				Correct			
				Prophet	SPR	GenProg	AE	Prophet	SPR	GenProg	AE
libtiff	77k	78	8/16	5/0	5/0	3/0	5/0	2/0	1/0	0/0	0/0
lighttpd	62k	295	7/2	3/1	3/1	4/1	3/1	0/0	0/0	0/0	0/0
php	1046k	8471	31/13	16/2	15/2	5/0	7/0	10/0	9/0	1/0	2/0
gmp	145k	146	2/0	2/0	2/0	1/0	1/0	1/0	1/0	0/0	0/0
gzip	491k	12	4/1	2/0	2/0	1/0	2/0	1/0	0/0	0/0	0/0
python	407k	35	9/2	5/1	5/1	0/1	2/1	0/0	0/0	0/1	0/1
wireshark	2814k	63	6/1	4/0	4/0	1/0	4/0	0/0	0/0	0/0	0/0
fbc	97k	773	2/1	1/0	1/0	1/0	1/0	1/0	0/0	0/0	0/0
Total			69/36	38/4	37/4	16/2	25/2	15/0	11/0	1/1	2/1

Figure 10. Benchmark Applications and Patch Generation Results

each of these four applications, we train Prophet separately and exclude the collected human patches of the same application from the training data. The goal is to ensure that we measure the ability of Prophet to apply the learned model trained with one set of applications to successfully patch defects in other applications.

Reproduce Defects: We reproduce each defect in our experimental environment. 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.

Apply Prophet to Defects: We use the trained Prophet to generate patches for each defect. For comparison, we also run SPR on each defect and obtain the results of GenProg [11] and AE [25] on this benchmark set from previous work [20]. For each defect and each patch generation system, we terminate execution after 12 hours.

To better understand how the probabilistic model, the learning algorithm, and the features affect the result, we also run four variants of Prophet, specifically Random (a naive random search algorithm that prioritizes the generated patches in a random order), Baseline (a baseline algorithm that prioritizes patches in the error localization order, with patches that modify the same statement prioritized in an arbitrary order), MF (an variant of Prophet with only modification features; program value features are disabled), and PF (a variant of Prophet with only program value features; modification features are disabled). All of these variants, Prophet, and SPR differ only in the patch validation order, i.e., they operate with the same patch search space and the same set of optimizations for validating candidate patches.

Note that GenProg and AE require the user to specify the source file name to modify when the user applies GenProg and AE to an application that contains multiple source files (all applications in our benchmark set contain multiple source files) [20]. Prophet and SPR do not have this limitation.

Evaluate Generated Patches: We manually analyze each generated patch to determine whether the generated patch is a correct patch or just a plausible but incorrect patch that produces correct outputs for all of the inputs in the test suite.

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, subsequent developer patches are available for all of the defects in our benchmark set. Our manual code analysis indicates that each of the generated correct patches in our experiments is semantically equivalent to the subsequent developer patch for that defect.

4.2 Patch Generation Result Summary

Figure 10 summarizes the patch generation results for Prophet, SPR, GenProg, and AE. There is a row in the table for each benchmark application. The first column (App) presents the name of the benchmark application, the second column (LoC) presents the number of lines of code in each application, the third column (Tests) presents the number of test cases in the test suite for that application. The fourth column (Defects/Changes) contains entries of the form X/Y, where X is the number of exposed defects in each application and Y is number of exposed functionality changes. The benchmark set contains a total of 69 exposed defects and 36 exposed functionality changes.

The fifth through eighth columns (Plausible) summarize the plausible patches that each system generates. Each entry is of the form X/Y, where X is the number of the 69 defects for which the corresponding system generates at least one plausible patch (that passes the supplied test suite) and Y is the number of the 36 functionality changes for which each system generates at least one plausible patch (that passes the supplied test suite).

The ninth through twelfth columns (Correct) summarize the correct patches that each system generates. Each entry is of the form X/Y, where X is the number of the 69 defects for which the corresponding system generates a correct patch (as the first generated plausible patch) and Y is the number of the 36 functionality changes for which each system generates a correct patch (as the first generated plausible patch).

The results show that Prophet generates correct patches for more defects than SPR, GenProg, and AE (four more defects than SPR, 14 more than GenProg, and 13 more than AE). One potential explanation for the underperformance of GenProg and AE is that the correct Prophet and SPR patches are outside the search spaces of these systems [12, 13, 20].

We attribute the overall success of Prophet to the ability of its learned model to recognize and prioritize correct patches. Our analysis indicates that the Prophet search space contains correct patches for 19 defects in our benchmark applications. With its learned model, Prophet automatically generates correct patches for 15 out of the 19 defects (79%), while SPR, with its hand-coded heuristics, generates correct patches for 11 out of the 19 defects (59%).

4.3 Comparison of Different Systems

Figure 11 presents results from different patch generation systems. The first column (System) presents the name of each system (Ran-

System	Corrected Defects	Mean Rank in Search Space	Time
Prophet	15	Top 11.7%	138.5m
Random	7	Top 41.8%	318.1m
Baseline	8	Top 20.8%	206.6m
MF	10	Top 12.2%	144.9m
PF	13	Top 12.4%	125.3m
SPR	11	Top 17.5%	89.0m
GenProg	1	N/A	N/A
AE	2	N/A	N/A

Figure 11. Comparative Results for Different Systems. For the results of "Mean Rank in Search Space", the lower ones are better.

dom, Baseline, MF, and PF are variants of Prophet with different capabilities disabled, see Section 4.1). The second column (Corrected Defects) presents the number of the 69 defects for which the system produces at least one correct patch. The underlying Prophet search space contains correct patches for 19 of these 69 defects. Prophet, MF, and PF generate plausible patches for all of these 19 defects; Baseline generates plausible patches for 18 of these 19 defects (see Figure 12). Random generates plausible patches for 15 of these 19 defects.

The third column (Mean Rank in Search Space) presents a percentage number, which corresponds to the mean rank, normalized to the size of the search space per defect, of the first correct patch in the patch prioritization 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. "Top X%" in an entry indicates that the corresponding system prioritizes the first correct patch as one of top X% of the candidate patches in the search space on average. Note that we run the random search algorithm with the default random seed to obtain its results in the figure.

Our results show that Prophet delivers the highest average rank (11.7%) for the first correct patch in the search space. The results also highlight the importance of the probabilistic model in enabling Prophet to generate correct patches — the Random and the Baseline systems, which operate without a probabilistic model or heuristics, generate only 7 and 8 correct patches, respectively.

Our results also indicate that the learned model is especially important for Prophet to identify correct patches among the potentially multiple plausible patches that pass a supplied test suite. The Baseline system has a significantly better rank percentage than the Random system, but generates a correct patch for only one more defect than the Random system. Without the learned model, the Baseline algorithm has difficulty recognizing and prioritizing correct patches over plausible but incorrect patches.

The results also highlight how program value features are more important than modification features for distinguishing correct from plausible but incorrect patches. We observed a common scenario that the search space contains multiple plausible patches with different program variables. In these scenarios, the learned model with program value features enables PF (and Prophet) to identify the correct patch among these multiple plausible patches.

The fourth column (Time) in Figure 11 presents the average running time of each system over all defects for which the system generates a correct patch. Our results show that Prophet requires, on average, less than two and half hours to generate each correct patch. Note that it is not possible to directly compare the running times of different systems because each system generates correct patches for a different set of defects.

During our experiments, we observed that for all of the tested systems the patch validation (i.e., update the source code with each candidate patch, recompile the updated application, and run all of the supplied test cases) is by far the most time consuming step — it takes more than 95% of the running time.

4.4 Per-Defect Results

Figure 12 presents detailed results for each of the 19 defects for which the Prophet search space contains correct patches. The figure contains a row for each defect. Each entry in the first column (Defect) is of the form 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 the developer repaired the defect. Each entry of the second column (Search Space) is of the form X(Y), where X is the total number of candidate patches and candidate patch templates (see Section 3.5) that Prophet and SPR consider in the patch search space and Y is the number of correct patches in the search space.

The third through seventh columns (First Correct Patch Rank) present the correct patch generation results for each system. The number in each entry is the rank of the first correct patch in the patch validation order for each system. "✓" in an entry indicates that the corresponding system successfully generates this correct patch as its first plausible patch. "△" in an entry indicates that the algorithm generates a plausible but incorrect patch before it reaches its first correct patch. "X" indicates that the algorithm fails to generate any plausible patch in 12 hours.

The eighth through twelfth columns (Correct/Plausible Patches in Space) present the statistics of plausible patches in the search space. Each entry is of the form X/Y. Y is the total number of plausible patches the corresponding system generates if we run the system on the corresponding defect exhaustively for 12 hours. X is the rank of the first correct patch among these generated plausible patches. "-" indicates that there is no correct patch among these generated plausible patches.

Our results show that for 5 out of the 19 cases (php-307562-307561, php-307846-307853, php-309516-309535, php-310991-310999, and php-307914-307915), all generated plausible patches are correct. The results indicate that for these five cases, the supplied test suite is strong enough to identify correct patches. Therefore any patch generation order is sufficient as long as it allows the patch generation system to find a correct patch within 12 hours. In fact, all five algorithms in Figure 12 generate correct patches for these 5 cases.

Prophet generates correct patches for 10 out of the remaining 14 defects, while SPR only generates correct patches for 6 of these 14 defects. Note that SPR empirically prioritizes candidate patches that change existing branch conditions above all other candidate patches in the search space [12, 13]. This rule conveniently allows SPR to generate correct patches for php-309579-309580, php-309892-309910, and php-311346-311348.

Prophet outperforms SPR on four cases, php-308262-308315, libtiff-d13be-ccadf, gzip-a1d3d4-f17cbd, and fbc-5458-5459. For php-308262-308315, the correct patch inserts an if statement guard for an existing statement. Prophet successfully prioritizes the correct patch as one of top 2% in the patch prioritization order, while the SPR hand-coded heuristics prioritize patches that add a guard statement below patches that change a branch condition. Because of this lower priority, SPR is unable to find the correct patch within 12 hours.

For gzip-a1d3d4-f17cbd, an initialization statement can be inserted at multiple candidate locations to pass the supplied test case,

Defect	Search Space	First Correct Patch Rank					Correct/Plausible Patches in Space				
		Prophet	SPR	MF	PF	Baseline	Prophet	SPR	MF	PF	Baseline
php-307562-307561	29530(1)	3109✓	4925✓	19✓	4444✓	4435✓	1/1	1/1	1/1	1/1	1/1
php-307846-307853	22106(1)	10744✓	3819✓	11116✓	10214✓	5904✓	1/1	1/1	1/1	1/1	1/1
php-308734-308761	14281(2)	5438✓	5721✓	7412✓	4650△	12662✓	1/4	1/4	1/4	3/4	1/4
php-309516-309535	27098(1)	10864✓	4000✓	9303✓	9682✓	8042✓	1/1	1/1	1/1	1/1	1/1
php-309579-309580	51260(1)	829✓	46✓	1977△	499✓	3775△	1/2	1/2	2/2	1/2	2/2
php-309892-309910	36533(4)	496✓	179✓	217△	879✓	1523✓	1/21	1/17	4/21	1/21	1/21
php-310991-310999	87574(2)	888✓	384✓	351✓	1342✓	5061✓	1/1	1/2	1/2	1/1	1/2
php-311346-311348	8730(2)	27✓	312✓	125✓	38✓	977△	1/49	1/50	1/49	1/49	12/50
php-308262-308315	81110(1)	1561✓	7189X	3094✓	2293✓	6784X	1/2	-/0	1/2	1/2	-/0
php-309688-309716	60787(1)	3632△	8338△	2226△	1547△	6340△	38/47	-/17	44/50	29/57	-/25
php-310011-310050	68534(1)	1297△	30647△	5629△	3145△	5581△	6/48	-/22	-/76	8/41	-/33
php-309111-309159	51995(1)	7788△	23666△	7041△	18106△	4308△	9/10	3/10	9/10	10/10	8/10
php-307914-307915	45362(1)	1✓	5748✓	2703✓	1✓	5110✓	1/1	1/1	1/1	1/1	1/1
libtiff-ee2ce-b5691	171340(1)	280✓	13296✓	174✓	44✓	335△	1/328	1/328	1/328	1/328	2/328
libtiff-d13be-ccadf	296165(1)	1179✓	372△	4671△	1333✓	714△	1/1423	3/1723	2/1723	1/1423	2/1723
libtiff-5b021-3dfb3	219464(1)	51186△	56569△	8235△	60913△	133018△	-/242	206/237	178/210	-/202	-/147
gmp-13420-13421	49929(2)	13834✓	14526✓	3214✓	13329✓	41319✓	1/3	1/3	1/3	1/3	1/3
gzip-a1d3d4-f17cbd	47413(1)	1866✓	21885△	16449△	706△	4123△	1/14	4/14	5/14	2/14	3/14
fbc-5458-5459	9788(1)	33✓	454△	741△	27✓	686△	1/37	8/37	5/37	1/37	5/38

Figure 12. Per-Defect Results

but not all of the resulting patches are correct. Prophet successfully identifies the correct patch among multiple plausible patches, while the SPR heuristics prioritize an incorrect patch that inserts the initialization at the start of a basic block.

For libtiff-d13be-ccadf and fbc-5458-5459, there are multiple candidate program variables that can be used to tighten a branch condition to enable the resulting patched program to pass the supplied test suite. The learned program value features enable Prophet (and PF) to successfully identify and prioritize correct patches that manipulate the right variables. The SPR heuristics (and MF, which operates without program value features) incorrectly prioritize patches that manipulate the wrong variables.

5. Related Work

We next survey related work in automatic patch generation. In general, one of the key features that distinguishes Prophet from many of other systems is its ability to generate correct patches for large software projects containing hundreds of thousands of lines of code. Previous patch generation systems that are evaluated on applications in similar scale include SPR [12, 13], CodePhage [24], ClearView [18], GenProg [11], RSRepair [19], AE [25], Kali [20], PAR [8], and NOPOL [3, 6].

SPR: SPR is the current state-of-the-art generate-and-validate (search-based) patch generation system [12] for large applications. SPR applies a set of transformation schemas to generate a search space of candidate patches. It then uses its staged program repair technique to validate the generated patches using a test suite of test cases, at least one of which exposes a defect in the original program. A patch validates if it produces correct outputs for all test cases in the test suite. SPR uses a set of hand-coded heuristics to guide the exploration of the search space.

Prophet works with the same patch search space as SPR but differs in that it learns characteristics of successful patches developed by human developers, then uses these characteristics to guide the exploration of the search space. Our experimental results show that this learned information enables Prophet to more effectively recognize and prioritize correct patches than the hand-coded SPR heuristics.

CodePhage: CodePhage automatically locates correct code in one application, then transfers that code to eliminate defects in another application [24]. CodePhage has been applied to eliminate otherwise fatal integer overflow, buffer overflow, and divide by zero errors. CodePhage relies on the existence of donor applications that already contain the exact program logic required to eliminate the defect. Prophet, in contrast, learns characteristics of successful patches to guide the exploration of an automatically generated search space of newly synthesized candidate patches.

JSNICE: JSNICE [21] is a JavaScript beautification tool that automatically predicts variable names and generates comments to annotate variable types for JavaScript programs. JSNICE first learns, from a "big code" database, a probabilistic model that captures 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 maximizes the probability in the learned model.

Unlike JSNICE, the goal of Prophet is to produce correct patches for defective programs. Prophet therefore aspires to solve deep semantic problems associated with automatically generating correct program logic. To this end, Prophet works with a probabilistic model that combines defect localization information with learned characteristics of successful patches. Prophet extracts a set of powerful features (especially program value features) that abstract away application-specific syntactic elements of the program to summarize useful application-independent characteristics of successful patches.

GenProg, RSRepair, AE, and Kali: GenProg [11, 26] 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 [19] replaces the GenProg genetic search algorithm to instead use random search. AE [25] uses a deterministic patch search algorithm and uses program equivalence relations to prune equivalent patches during testing.

An analysis of the generated patches shows that the overwhelming majority of the patches that these three systems generate are incorrect [20]. Because of errors in the patch validation infrastructure, the majority of the generated patches do not produce correct results even for the test cases in the test suite used to validate the patches [20]. Further analysis of the patches that do produce correct outputs for the test suite reveals that despite the surface complexity of these patches, an overwhelming majority of these patches sim-

ply remove functionality [20]. The Kali patch generation system, which only eliminates functionality, can do as well [20].

Prophet differs in that it works with a richer space of candidate patches, uses learned features of successful patches to guide its search, and generates substantially more correct patches.

Repair with Formal Specifications: Deductive Program Repair formalizes the program repair problem as a program synthesis problem, using the original defective program as a hint [10]. It replaces the expression to repair with a synthesis hole and uses a counterexample-driven synthesis algorithm to find a patch that satisfies the specified pre- and post-conditions. AutoFixE [17] is a program repair tool for Eiffel programming language. AutoFixE leverages the developer-provided formal specifications (e.g., post-condtions, pre-conditions, and invariants) to automatically find and generate repairs for defects. Cost-aware Program Repair [23] abstracts a C program as a boolean constraint, repairs the constraint based on a cost model, and then concretizes the constraint back to a repaired C program. The goal is to find a repaired program that satisfies all assertions in the program with minimal modification cost. The technique was evaluated on small C programs (less than 50 lines of code) and requires human intervention to define the cost model and to help with the concretization.

Prophet differs from these techniques in that it works with large real world applications where formal specifications are typically not available. Note that the Prophet probabilistic model and the Prophet learning algorithm can apply to these specification-based techniques as well, i.e., if there are multiple patches that satisfy the supplied specifications, the learned model can be used to determine which patch is more likely to be correct.

PAR: PAR [8] is another automatic patch generation system. Unlike Prophet, which uses a probabilistic model and machine learning techniques to automatically learn useful characteristics of past successful patches, PAR is based on a set of predefined patch templates that the authors manually summarize from past human patches. 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 [8]. A manual analysis indicates that the PAR search space (with the eight hand-coded templates in the PAR paper [8]) is in fact a subset of the Prophet/SPR search space [12, 13].

Angelic Debugging: Angelic Debugging [2] relaxes the program semantics to support angelic expressions that may take arbitrary values and identifies, in a program, those expressions which, if converted into angelic expressions, enable the program to pass the supplied test cases. Prophet could use the methodology of Angelic Debugging to improve its error localization component.

NOPOL and SemFix: NOPOL [3, 6] applies the angelic debugging technique to locate conditions that, if changed, may enable defective JAVA program to pass the supplied test suite. It then uses an SMT solver to synthesize repairs for such conditions. SemFix [16] replaces a potentially faulty expression in a program with a symbolic value, performs symbolic executions on the supplied test cases to generate symbolic constraints, and uses SMT solvers to find concrete expressions that enable the program to pass the test cases.

Prophet differs from NOPOL and SemFix [16] in that these techniques rely only on the information from the supplied test cases with the goal of finding plausible (but not necessarily correct) patches. Prophet learns features of past successful patches to recognize and prioritize correct patches among multiple plausible patches.

ClearView: ClearView is a generate-and-validate system that observes normal executions to learn invariants that characterize safe behavior [18]. 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.

Repair Model: Martinez and Monperrus manually analyze previous human patches and suggest that if a patch generation system works with a non-uniform probabilistic model, the system would find plausible patches in its search space faster [15]. In contrast, Prophet automatically learns a probabilistic model from past successful patches. Prophet is the first patch generation system to operate with such a learned model. The goal is to automatically identify correct patches among the plausible patches in the search space.

Data Structure Repair: Data structure repair enables applications to recover from data structure corruption errors [5]. 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 [4].

Targeted Recovery Techniques: Failure-oblivious computing [22] 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 accesses, and enables the program to continue execution along its normal execution path.

RCV [14] 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.

Bolt [9] 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.

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.

6. Conclusion

Prophet automatically learns and exploits features of previous successful patches to automatically generate correct patches for defects in different applications. The experimental results show that, in comparison with previous patch generation systems, the learned information significantly improves the ability of Prophet to generate correct patches. Key contributions include a novel parameterized probabilistic model that enables Prophet to learn relevant characteristics of successful patches, application-independent features that generalize across multiple applications, and experimental results that characterize the effectiveness of the different techniques that Prophet implements.

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] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11', pages 121–130, New York, NY, USA, 2011. ACM.

[3] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, CSTVA 2014, pages 30–39, New York, NY, USA, 2014. ACM.

[4] 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.

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

[6] T. Durieux, M. Martinez, M. Monperrus, R. Sommerard, and J. Xuan. Automatic repair of real bugs: An experience report on the defects4j dataset. *CoRR*, abs/1505.07002, 2015.

[7] M. Jose and R. Majumdar. Cause clue clauses: Error localization using maximum satisfiability. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11', pages 437–446, New York, NY, USA, 2011. ACM.

[8] 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.

[9] 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.

[10] E. Kneuss, M. Koukoutos, and V. Kuncak. Deductive program repair. In *Computer-Aided Verification (CAV)*, 2015.

[11] 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.

[12] F. Long and M. Rinard. Staged program repair in SPR. In *Proceedings of ESEC/FSE 2015 (to appear)*, 2015.

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

[14] 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.

[15] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[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*, ICSE '13', pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[17] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. *IEEE Trans. Softw. Eng.*, 40(5):427–449, May 2014.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] R. Samanta, O. Olivo, and E. A. Emerson. Cost-aware automatic program repair. In *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, pages 268–284, 2014.

[24] 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.

[25] 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.

[26] 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.

[27] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, Feb. 2002.