Automatic Patch Generation by Learning Correct Code

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

Abstract

We present Prophet, a novel patch generation system that works with a set of successful human patches obtained from open-source software repositories to learn a probabilistic, application-independent model of correct code. It generates a space of candidate patches, uses the model to rank the candidate patches in order of likely correctness, and validates the ranked patches against a suite of test cases to find correct patches. Experimental results show that, on a benchmark set of 69 real-world defects drawn from eight open-source projects, Prophet significantly outperforms the previous state-of-the-art patch generation system.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis; D.2.5 [*SOFTWARE ENGINEERING*]: Testing and Debugging

Keywords Program repair, Code correctness model, Learning correct code

1. Introduction

We present Prophet, a new generate-and-validate patch generation system for repairing defects in large, real-world applications. To the best of our knowledge, Prophet is the first system to learn a probabilistic model of correct code. Prophet uses this model to automatically generate correct patches that repair defects in incorrect applications:

- **Learned Model of Correct Code:** Working with a set of successful patches obtained from open-source software repositories, Prophet learns a probabilistic model of correct code. It uses this model to rank and identify correct patches within an automatically generated space of candidate patches (only a small fraction of which are correct).
- **Code Interactions:** Each patch inserts new code into the program. But correctness does not depend only on the new code — it also depends on how that new code interacts with the application into which it is inserted. The learned correctness model therefore works with features that capture critical aspects of how the new code interacts with the surrounding code from the patched application.
- **Universal Features:** A fundamental new hypothesis behind the design of Prophet is that, across applications, correct code shares a core set of universal correctness properties. To expose and learn these properties, Prophet works with *universal features* that abstract away shallow syntactic details that tie code to specific applications while leaving the core correctness properties intact at the semantic level. These universal features are critical for enabling Prophet to learn a correctness model from patches for one set of applications, then successfully apply the model to new, previously unseen applications.
- **Large Applications:** Unlike many previous patch generation systems, which work with small programs with hundreds of lines of code [14, 23, 30, 31], Prophet generates correct patches for large, real-world applications with tens of thousands to a million lines of code or more. The previous state-of-the-art system for such applications uses a set of manually derived heuristics to rank candidate patches [18]. The experimental results show that the Prophet code correctness model enables Prophet to significantly outperform this previous state-of-the-art system on the same benchmark set.

Generate-and-validate systems start with a program and a 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 within the (potentially quite large) space of plausible but incorrect patches [18, 25].

Prophet uses its learned model of correct code to rank the patches in its search space, with the goal of obtaining a correct patch as the first (or one of the first few) patches to validate.

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 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 correct patches, Prophet learns the model parameters by 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 correct patches in the training set.

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA
© 2016 ACM. 978-1-4503-3549-2/16/01...$15.00
http://dx.doi.org/10.1145/2837614.2837617

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 ranked list of target program statements to patch (see Section 3.7). Prophet prioritizes statements that are frequently executed on negative inputs (for which the unpatched program produces incorrect results) and infrequently executed on positive inputs (for which the unpatched program produces correct results).
- **Search Space Generation:** Prophet generates a space of candidate patches, each of which modifies one of the statements identified by the defect localization algorithm.
- **Universal Feature Extraction:** For each candidate patch, Prophet extracts features that summarize relevant patch properties. 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 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 an ordered sequence of patches that validate (i.e., produce correct outputs for all test cases in the test suite) as the result of the patch generation process.

Universal Roles and Program Value Features: A key challenge for Prophet is to identify, learn, and exploit universal properties of correct code. Many surface syntactic elements of the correct patches in the Prophet training set (such as variable names and types) tie the patches to their specific applications and prevent the patches from directly generalizing 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 (universal, application-independent) 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 the roles that the same value plays in the patch and the original code that the patch modifies. These relationships capture interactions between the patch and the patched code that correlate with patch correctness and incorrectness. Because the features are derived from universal, application-independent roles, they generalize across different applications.

1.2 Hypothesis

A key hypothesis of this paper is that, across applications, successful human patches share certain characteristics which, if appropriately extracted and integrated with a patch generation system, will enable the system to identify correct patches among the candidate patches in its search space. The experimental are consistent with this hypothesis.

1.3 Experimental Results

We evaluate Prophet on 69 real world defects drawn from eight large open source applications. The results show that, on the same benchmark set, Prophet outperforms previous generate-and-validate patch generation systems, specifically SPR [18], Kali [27], GenProg [15], and AE [35].

The Prophet search space contains correct patches for 19 of the 69 defects. Within its 12 hour time limit, Prophet finds correct patches for 18 of these 19 defects. For 15 of these 19 defects, the first patch to validate is correct. SPR, which uses a set of hand-coded heuristics to prioritize its search of the same space of candidate patches, finds correct patches for 16 of these 19 defects within its 12 hour time limit. For 11 of these 19 defects, the first patch to validate is correct. Kali, GenProg, and AE find correct patches for 2, 1, and 2 defects, respectively.

The results also highlight how program value features are critical for the success of Prophet. Within its 12 hour time limit, a variant of Prophet that disables program value features also finds correct patches for 18 of these 19 defects. But for only 10 of these 19 defects is the first patch to validate correct. 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.

1.4 Contributions

This paper makes the following contributions:

- **Hypothesis:** It presents the hypothesis that, even across applications, correct code shares properties that can be learned and exploited to generate correct patches for incorrect applications.
- **New Approach:** It presents a novel approach for learning correct code. This approach uses a parameterized discriminative probabilistic model to assign a correctness probability to candidate patches. This correctness probability captures not only properties of the new code present in the candidate patches, but also properties that capture how this new code interacts with the surrounding code into which it is inserted. It also presents an algorithm that learns the model parameters via a training set of successful human patches collected from open-source project repositories.
- **Features:** It presents a novel set of universal features for capturing deep semantic code correctness properties across different applications. Because these features abstract away application-specific surface syntactic elements (such as variable names) while preserving important structural characteristics, they significantly improve the ability of Prophet to learn universal properties of correct code.
- **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 system to use a machine learning algorithm to automatically learn and exploit properties of correct code.
- **Experimental Results:** It presents experimental results that evaluate Prophet on 69 real world defects. The Prophet search space contains correct patches for 19 of these 69 defects. For 15 of these 19 defects, Prophet finds a correct patch as the first patch to validate. Working with the same search space, the previous state-of-the-art system finds a correct patch as the first to validate for 11 of these 19 defects [18].

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 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 (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 32-34).

In some situations PHP should suppress these out-of-bounds runtime errors. Consider, for example, a PHP program that calls 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 isset() is to check if a value is properly set or not. Generating an error message when isset() calls the procedure in Figure 1 is invalid because it interferes with the proper operation of 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.

Offline Learning: Prophet works with a training set of successful human patches to obtain a probabilistic model that captures why these patches were successful. We obtain this training set by collecting revision changes from open source repositories. In our example, we train Prophet with patches from seven open source projects (apr, curl, 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 training set, Prophet analyzes a structural diff on the abstract syntax trees of the original and patched code to extract both 1) modification features, which summarize how the patch modifies the program given characteristics of the surrounding code and 2) program value 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 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 set in the probabilistic model.

Apply Prophet: We 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

```
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 //        An unconstrained patch with abstract condition C
23 //        if (C), where C is unconstrained
24 //        An partially instantiated patch
25 //        if (C), where C checks the variable "type"
26
27 //        The guard that the correct Prophet patch inserts
28 //        before the following error generation statement.
29 //        This Prophet patch is identical to the (correct)
30 //        developer patch.
31 //        if (!(type == 3))
32           zend_error(E_NOTICE,
33                      "Uninitialized string offset: %ld",
34                      (*dim).value.lval);
35         Z_STRVAL_P(ptr) = STR_EMPTY_ALLOC();
36         Z_STRLEN_P(ptr) = 0;
37       } else {
38         Z_STRVAL_P(ptr) = (char*)emalloc(2);
39         Z_STRVAL_P(ptr)[0] =
40             Z_STRVAL_P(container)[Z_LVAL_P(dim)];
41         Z_STRVAL_P(ptr)[1] = 0;
42         Z_STRLEN_P(ptr) = 1;
43       }
44       AI_SET_PTR(result, ptr);
45       return;
46     } break;
47     ...
48   }
49 }
```

Figure 1. Simplified Code for PHP bug #53971

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 32-34 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, which uses *transformation schemas*, *staged program repair*, and *condition synthesis* to generate candidate patches [18]. Some (but by no means all) of these candidate patches are generated by a transformation schema (see lines 22-23) that adds an if statement to guard (conditionally execute) the statement at lines 32-34 in Figure 1. This transformation schema contains

an abstract condition that the Prophet condition synthesis algorithm will eventually instantiate with a concrete condition [18]. During search space generation and candidate patch ranking, Prophet does not attempt to fully instantiate the patch. It instead works with *partially instantiated patches* that identify the variable that the final concrete condition will check (but not the final concrete condition itself).
In our example one of the partially instantiated patches is shown as lines 24-25. It 1) adds an if statement guard before the statement at lines 32-34 in Figure 1 (the statement that generates the error message) and 2) has a condition that checks the function parameter type.

- **Rank Candidate Patches:** Prophet computes a feature vector for each candidate (fully or partially instantiated) 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 partially instantiated patch mentioned above (lines 32-34) because it has several features that positively correlate with correct patches in the training set. For example, 1) it adds an if statement to guard a call statement and 2) the guard condition checks a parameter of the enclosing procedure.
- **Validate Candidate Patches:** Prophet then uses the test suite to attempt to validate the candidate patches (including partially instantiated patches) in order of highest patch correctness probability. When the validation algorithm encounters a partially instantiated patch, Prophet invokes the Prophet condition synthesis algorithm to obtain concrete conditions that fully instantiate the patch [18]. In our example, the condition synthesis algorithm comes back with the condition (type != 3) (the resulting patch appears at line 31 in Figure 1). This patch 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 type != 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. The Prophet model ranks this plausible but incorrect patch below the correct patch because the incorrect patch inserts a return statement before a subsequent assignment statement in a code block. This interaction between the patch and the surrounding code incurs a significant penalty in the learned model.

3. Design

Prophet first performs an offline training phase to learn a probabilistic model over features of successful patches drawn from a large set of revisions. 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 model guides the exploration of the search space.

We first discuss how Prophet works with the fully instantiated patches in the Prophet search space. We then extend the treatment to partially instantiated patches (see Section 3.6).

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 3.3). Once θ is determined, the probability can be interpreted as a normalized score (i.e., $\sum_\delta P(\delta \mid p, \theta) = 1$) which prioritizes 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 and 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 $\langle m, \ell \rangle$. We define $P(\delta \mid p, \theta) = P(m, \ell \mid p, \theta)$ for $\ell \in L(p)$ and $m \in M(p, \ell)$ as follows:

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

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

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

Here B is a standard parameterized log-linear distribution determined by the extracted feature vectors ϕ and the learned parameter vector θ. 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 set $L(p)$ of candidate program points to modify (see Section 3.7). $r(p, \ell)$ denotes the rank of $\ell \in L(p)$ assigned by the defect localization algorithm. Here β is the parameter of the geometric distribution (which Prophet empirically sets to 0.02).

We use a geometric distribution for the defect localization information because previous defect localization work reports that statements with higher localization ranks are significantly more likely to be patched than statements with lower localization ranks [11, 37]. The Prophet geometric geometric distribution matches this observation of previous work.

Intuitively, the 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 Defect Localization Approximation for Learning

The input to the Prophet training phase is a large set of revision changes $D = \{\langle p_1, \delta_1 \rangle, \ldots, \langle p_n, \delta_n \rangle\}$, where each element of D

```
Input  : the training set $D = \{\langle p_1, \delta_1\rangle, \ldots, \langle p_n, \delta_n\rangle\}$, where $p_i$ is
         the original program and $\delta_i$ is the successful human patch
         for $p_i$.
Output: the feature weight parameter vector $\theta$.
1  for $i = 1$ to $n$ do
2      $\langle m_i, \ell_i\rangle \longleftarrow \delta_i$
3      $L'_i \longleftarrow \mathrm{NearLocations}(p_i, \ell_i)$
4  $n_0 \longleftarrow 0.85 \cdot n$
5  Initialize all elements in $\theta$ to 0
6  $\theta^* \longleftarrow \theta$
7  $\alpha \longleftarrow 1$
8  $\gamma^* \longleftarrow 1$
9  $cnt \longleftarrow 0$
10 while $cnt < 200$ do
11     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     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     $\theta \longleftarrow \theta + \alpha \cdot \frac{\partial f}{\partial \theta}$
14     $\gamma \longleftarrow 0$
15     for $i = n_0 + 1$ to $n$ do
16         $tot \longleftarrow |\{m \mid m \in M(p_i, \ell),\ \ell \in L'_i\}|$
17         $rank \longleftarrow |\{m \mid m \in M(p_i, \ell),\ \ell \in L'_i,$
               $g(p_i, \ell, m, L'_i, \theta) \geq g(p_i, \ell_i, m_i, L'_i, \theta)\}|$
18         $\gamma \longleftarrow \gamma + (rank/tot)/(n - n_0)$
19     if $\gamma < \gamma^*$ then
20         $\theta^* \longleftarrow \theta$
21         $\gamma^* \longleftarrow \gamma$
22         $cnt \longleftarrow 0$
23     else
24         $cnt \longleftarrow cnt + 1$
25         if $\alpha > 0.01$ then
26             $\alpha \longleftarrow 0.9 \cdot \alpha$
27 return $\theta^*$
```

Figure 2. Learning Algorithm

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 resulting probabilistic model assigns a high conditional probability score to δ_i among all possible candidate patches in the search space.

It is, in theory, possible to learn θ directly over $P(m, \ell \mid p, \theta)$. But obtaining the defect localization information requires 1) a compiled application that runs in the Prophet training environment and 2) a test suite that includes both positive and negative test cases (and not just a standard set of regression test cases for which the unpatched application produces correct output).

The Prophet learning algorithm therefore uses an oracle-like defect localization approximation 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 L'_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 uses maximum likelihood estimation to learn θ over the following formula:

$$\theta = \arg\max_{\theta} \left(\sum_i \log C_i - \lambda_1 \sum_i |\theta_i| - \lambda_2 \sum_i \theta_i^2 \right)$$

$$C_i = \frac{\exp\left(\phi(p_i, m_i, \ell_i) \cdot \theta\right)}{\sum_{\ell' \in L'_i} \sum_{m' \in M(p_i, \ell')} \exp\left(\phi(p_i, m', \ell') \cdot \theta\right)}$$

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}.

Using the defect localization approximation (as opposed to full defect localization) provides at least two advantages. First, it significantly expands the range of applications from which Prophet can draw training patches — it enables Prophet to work with successful human patches from applications even if the application does not fully compile and execute in the Prophet training environment and even if the application does not come with an appropriate test suite. Second, it also improves the running time of the training phase (which takes less than two hours in our experiments, see Section 4), because Prophet does not need to compile and run patches during training.

3.3 Learning Algorithm

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

- **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 $\mathrm{NearLocations}(p_i, \ell_i)$ at line 3 returns a set of program points that are close to the known correct modification point ℓ_i.
- **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 algorithm terminates and returns θ^* as the result.

3.4 Feature Extraction

Figure 3 presents the syntax of a simple programming language which we use to present the Prophet feature extraction algorithm (see the end of this section for a discussion of how we extend the feature extraction algorithm for 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. For brevity, we omit the operational semantics.

Figure 4 presents the notation we use to present the feature extraction algorithm. Figure 5 presents the feature extraction algorithm itself. Given a program p, a program point ℓ, and a modifi-

$$\psi : \mathbf{Prog} \times \mathbf{Atom} \times (\mathbf{Cond} \cup \mathbf{Stmt}) \to \mathbf{AC} \qquad \psi(p, a, node) = \psi_0(a, node) \cup \psi_1(a, node)$$
$$\mathbf{AC} = \{\texttt{var}, \texttt{const0}, \texttt{constn0}, \texttt{cond}, \texttt{if}, \texttt{prt}, \texttt{loop}, \texttt{==}, \texttt{!=}, \langle \text{op}, \texttt{L}\rangle, \langle \text{op}, \texttt{R}\rangle, \langle \texttt{=}, \texttt{L}\rangle, \langle \texttt{=}, \texttt{R}\rangle\}$$

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

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

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

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

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

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

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

Figure 3. The language statement syntax

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

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

```
Input  : the input program $p$, the modified program point $\ell$, and the
         modification operation $m$
Output: the extracted feature vector $\phi(p, \ell, m)$
1  Initialize all elements in $\phi$ to 0
2  $S_{\text{C}} \longleftarrow \{\text{Stmt}(p, \ell)\}$
3  $S_{\text{P}} \longleftarrow \text{Prev3stmts}(p, \ell))$
4  $S_{\text{N}} \longleftarrow \text{Next3stmts}(p, \ell))$
5  $idx \longleftarrow \text{FIdx}(\text{ModKind}(m))$
6  $\phi_{idx} \longleftarrow 1$
7  for $i$ in $\{C, P, N\}$ do
8      for $s$ in $S_i$ do
9          $idx \longleftarrow \text{Fid}(\langle i, \text{StmtKind}(s), \text{ModKind}(m)\rangle)$
10         $\phi_{idx} \longleftarrow 1$
11 $\langle p', n\rangle \longleftarrow \text{ApplyPatch}(p, \langle m, \ell\rangle)$
12 for $i$ in $\{C, P, N\}$ do
13     for $a$ in $\text{Atoms}(n)$ do
14         for $s$ in $S_i$ do
15             for $ac'$ in $\psi(p', a, n)$ do
16                 for $ac$ in $\psi(p, a, s)$ do
17                     $idx \longleftarrow \text{FIdx}(\langle i, ac, ac'\rangle)$
18                     $\phi_{idx} \longleftarrow 1$
19 return $\phi$
```

Figure 5. Feature Extraction Algorithm

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

Prophet then extracts two types of features, modification features (lines 5-10) and program value features (lines 11-18). Modification features capture interactions between the modification m and the surrounding statements, while program value features capture 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 ϕ 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: Prophet implements two classes of modification features. The first class captures the kind of modification that m applies. The second class captures 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 relative to the patched statement ("C" corresponds to the original statement, "P" corresponds to one of the three previous statements in the same block, and "N" corresponds to one of the three next statements in the same block), the kind of the original statement, and the modification kind of m as the modification feature. At line 9, the utility function $\text{StmtKind}(s)$ returns the statement kind of s and the utility function $\text{ModKind}(m)$ 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 nearby call statement, a program value feature will enable Prophet to recognize and exploit this fact. Program value features capture interactions between an occurrence of a variable or constant in the original program and an occurrence of the same variable or constant in the new code in the patch.

To avoid polluting the feature space with application-specific information, program value features abstract away the specific names of variables and values of constants involved in the interactions that these features model. This abstraction enables Prophet to learn properties of correct code as captured by program value features from patches for one set of applications, then apply the learned information to generate correct patches for other applications.

To extract program value features, Prophet first applies the patch to the original program (line 11 in Figure 5). $\text{ApplyPatch}(p, \langle m, \ell\rangle)$ denotes the results of the patch application, which produces a pair $\langle p', n\rangle$, where p' is the new patched program and n is the AST node for the new statement or condition that the patch introduces. Prophet performs a static analysis on both the patched and original programs 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 $\text{Atoms}(n)$ 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. The first three rules in Figure 6 track whether an expression atom is a variable, a zero constant, or a non-zero constant. The fourth through eleventh rules track statement types and operators that are associated with each expression atom. The last three rules recursively compute and propagate atomic characteristics for if statements, statement blocks, and while statements, respectively.

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 the only variable in an abstract expression Is replaced by the modification operation

Figure 7. Atomic Characteristics of Program Values for C

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.

Feature Extraction for C: Prophet extends the feature extraction algorithm described in Section 3.4 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 7 for a list of these atomic characteristics) and works with a total of 3515 features, including 455 modification features and 3060 program value features.

3.5 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 $\text{Oracle}(p, \delta) = \texttt{true}$ if and only if δ correctly patches the defect in p.

Note that the oracle function 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 $\text{Oracle}(p, \delta)$ implies $T(p, \delta)$. Specifically, $T(p, \delta) = \texttt{true}$ 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 8 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:

- **Generate Search Space:** At line 1, Prophet runs the defect localization algorithm ($\text{DefectLocalizer}(p, T)$) 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.

```
Input : the original program $p$, the test suite $T$ and the learned
        model parameter vector $\theta$
Output: emit a list of validated patches
$\langle L, r \rangle \longleftarrow \text{DefectLocalizer}(p, T)$
$Candidates \longleftarrow \emptyset$
for $\ell$ in $L$ do
    for $m$ in $M(p, \ell)$ do
        $prob\_score \longleftarrow (1-\beta)^{r(p,\ell)} \cdot e^{\phi(p,\ell,m)\cdot\theta}$
        $Candidates \longleftarrow Candidates \cup \{\langle prob\_score, m, \ell \rangle\}$
$SortedCands \longleftarrow \text{SortWithFirstElement}(Candidates)$
for $\langle \_, m, \ell \rangle$ in $SortedCands$ do
    $\delta \longleftarrow \langle m, \ell \rangle$
    if $T(p, \delta) = true$ then
        emit $\delta$
```

Figure 8. Prophet Repair Algorithm

- **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 $P(\delta \mid p, \theta)$, 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 a list of validated candidate patches.

3.6 Staged Program Repair and Prophet

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 [18]. This search space is derived from a set of parameterized transformation schemas that Prophet applies to target statements identified by the defect localization algorithm [18]. Some of these schemas contain abstract conditions that the Prophet condition synthesis algorithm will later instantiate to obtain a final patch. Specifically, Prophet implements schemas that 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 disjoining a condition C to the if condition), 3) (Add Guard) add a guard with a condition C to a target statement, and 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. Here C is an abstract condition that the condition synthesis algorithm will later instantiate. This staged approach enables Prophet (like SPR) to efficiently bypass incorrect patches and focus the search on the most promising parts of the search space [18].

Prophet also implements schemas that produce a final fully instantiated patch directly without an intermediate condition synthesis step. These schemas include 1) (Initialize) insert a memory initialization statement before the target statement, 2) (Replace) replace one value in the target statement with another value, and 3) (Copy and Replace) copy an existing statement before the target statement and replace a value in the copied statement with another value.

Partially Instantiated Patches: There are two obvious approaches to implement staged program repair within Prophet. The first is to use staged program repair to generate fully instantiated final patches during the initial generation of the search space, then use the Prophet learned model to rank these patches for validation along with all of the other candidate patches. A downside of this approach is the time required to run the condition synthesis algorithm (which compiles and executes the application potentially multiple times) to generate fully instantiated patches (many of which will have low correctness probabilities).

The second approach is to rank the uninstantiated patch (this patch has an unconstrained abstract condition C), then instantiate the abstract condition C later during patch validation. The downside of this approach is that the correctness of the final instantiated patches will depend heavily on the variables that they access. This information, of course, is not available for patches with unconstrained abstract conditions, which inhibits the ability of Prophet to compute accurate correctness probabilities for the final fully instantiated patches that the condition synthesis algorithm will generate.

Prophet therefore uses an intermediate third approach — it generates and ranks *partially instantiated patches* that specify the variable that the synthesized condition will check, but leave the abstract condition otherwise unconstrained. This approach enables Prophet to work with patches that it can acceptably accurately rank while deferring condition synthesis until patch validation time. Because deferring condition synthesis enables Prophet to move quickly on to start validating highly ranked patches, it can significantly reduce the time Prophet requires to find correct fully instantiated patches.

Learning for Partially Instantiated Patches: We extend the Prophet feature extraction algorithm to handle partially instantiated patches. Specifically, we define atomic characteristics that identify variables that the conditions in partially instantiated patches check (see Figure 7).

The Prophet learning algorithm works with partially instantiated patches as follows. For each patch in the training set that could have been generated by a schema with an abstract condition, it derives the corresponding partially instantiated patch. It then extracts the features for this partially instantiated patch and learns over the partially instantiated patch and its extracted features (instead of learning over the fully instantiated patch).

3.7 Defect Localization

Prophet uses the same defect localization algorithm as SPR [17, 18] (the SPR technical report presents this algorithm [17]). 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 and produces a prioritized list that contains target statements to modify based on the recorded timestamp values. 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.

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 [3, 11, 37].

3.8 Alternative Learning Objective

Prophet uses maximum likelihood estimation to learn the model parameter θ. One alternative learning objective is to minimize the sum of hinge losses as defined by a hinge-loss function $h(p, m, l, \theta)$:

$$h(p, m, l, \theta) = \max_{l' \in L(p), m' \in M(p, m')} \\ ((\phi(p, m', l') \cdot \theta - \phi(p, m, l) \cdot \theta) + \Delta(p, \langle m, l \rangle, \langle m', l' \rangle))$$

Prophet can then learn θ with the following objective function:

$$\theta = \arg\min_{\theta} \left(\sum_i h(p_i, m_i, l_i, \theta) + \lambda_1 \sum_i |\theta_i| + \lambda_2 \sum_i \theta_i^2 \right)$$

Intuitively, for each correct patch in the training set, the hinge-loss function measures the score difference between the correct patch and the incorrect patch with the highest score plus the distance between these two patches. The objective function minimizes the sum of the hinge losses over all correct patches in the training set. λ_1 and λ_2 are regularization parameters, which we empirically set to 10^{-3} (we found that 10^{-3} gives the best results in our experiments). Δ is an arbitrary distance function. In our implementation, we use the euclidean distance between two feature vectors. In the ideal case, the hinge-loss learning algorithm finds a θ such that the score of the correct patch outweighs the incorrect patch with the highest score by a significant margin given by the distance between the two patches.

Although previous work has used the hinge-loss learning algorithm to successfully predict program properties such as variable names and types [28], our experimental results show that, for our set of benchmark defects, maximum likelihood estimation outperforms using the hinge-loss objective function (see Section 4.3).

The reason is that the hinge-loss function considers only the score of the most highly ranked incorrect patch and not the scores of the other incorrect patches. The hinge-loss algorithm therefore (unlike maximum likelihood estimation) does not directly attempt to optimize the rank of the correct patch within the full set of candidate patches. On both the training and benchmark sets, the hinge-loss algorithm is unable to find a θ that consistently ranks the correct patch within the top few patches in the search space. In this situation maximum likelihood estimation, because it considers the scores of all of the patches, produces a θ that ranks the correct patches more highly within the search space than the θ that the hinge-loss algorithm produces. The result is that the correct patches appear earlier in the validation order with maximum likelihood estimation than with the hinge loss algorithm.

4. Experimental Results

We evaluate Prophet on 69 real world defects and 36 functionality changes in eight large open source applications: libtiff, lighttpd, the PHP interpreter, gmp, gzip, python, wireshark, and fbc. This is the same benchmark set used to evaluate SPR [18], Kali [27], GenProg [15], and AE [35].[1] 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 for which the unpatched program produces incorrect output (i.e., the negative test case exposes the defect).

This benchmark set is, to the best of our knowledge, currently the most comprehensive publicly available C data set suitable for evaluating generate-and-validate systems — other benchmark sets have fewer defects, much smaller programs, or do not have the positive and negative test cases and build infrastructures required for generate-and-validate patch generation.

4.1 Methodology

Collect Successful Human Patches: We used the advanced search functionality in GitHub [1] to obtain a list of open source C project repositories that 1) were started before January 1, 2010 and 2) had more than 2000 revisions. We browsed the projects from the list one by one and collected the first eight projects that 1) are command-line applications or libraries running on our experimental environment Ubuntu 14.04, 2) whose repositories contain more than ten revision changes with patches that are within the Prophet search space, and 3) whose compilation flags can be extracted by our scripts for clang to obtain abstract syntax trees for these patches.

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 Successful Human Patches

In this process, we considered but rejected many applications because they do not satisfy the above requirements. For example, we rejected lighttpd because it contained fewer than ten revision changes with patches within the Prophet search space. We rejected git because we were unable to extract its compilation flags using our scripts. We stopped collecting projects when we believed we had obtained a sufficiently large training set of successful patches.

For each of the resulting eight application repositories, we ran a script to analyze the check-in logs to identify and collect all of those patches that repair defects (as opposed to changing or adding functionality) and are within the Prophet search space. From the eight repositories, we collected a total of 777 such patches. Figure 9 presents statistics for these 777 patches.

Train Prophet on Collected Training Set: We train Prophet on the collected set of successful human patches. The collected set of training applications and the benchmark set share four common applications, specifically libtiff, PHP, python, and wireshark. For each of these four applications, we train Prophet separately and exclude the collected human patches of the same application from the training set. The goal is to ensure that we evaluate the ability of Prophet to apply the learned model trained with one set of applications to successfully repair defects in other applications.

The offline training takes less than two hours. Training is significantly faster than repair because the learning algorithm does not compile and run the patches in the training set during training (see Section 3.3). This approach enables Prophet to include patches from applications 1) for which an appropriate test suite may not be immediately available and/or 2) with relevant source code files that may not fully compile on the training platform. These two properties can significantly expand the range of applications that can contribute patches to the training set of successful human patches.

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

[1] This benchmark set is reported to contain 105 defects [15]. 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 [18]. 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.

App	LoC	Tests	Defects/ Changes	Plausible					Correct				
				Prophet	SPR	Kali	GenProg	AE	Prophet	SPR	Kali	GenProg	AE
libtiff	77k	78	8/16	5/0	5/0	5/0	3/0	5/0	2,2/0	1,1/0	0/0	0/0	0/0
lighttpd	62k	295	7/2	3/1	3/1	4/1	4/1	3/1	0,0/0	0,0/0	0/0	0/0	0/0
php	1046k	8471	31/13	17/1	16/1	8/0	5/0	7/0	13,10/0	10,9/0	2/0	1/0	2/0
gmp	145k	146	2/0	2/0	2/0	1/0	1/0	1/0	1,1/0	1,1/0	0/0	0/0	0/0
gzip	491k	12	4/1	2/0	2/0	1/0	1/0	2/0	1,1/0	1,0/0	0/0	0/0	0/0
python	407k	35	9/2	5/1	5/1	1/1	0/1	2/1	0,0/0	0,0/0	0/1	0/1	0/1
wireshark	2814k	63	6/1	4/0	4/0	4/0	1/0	4/0	0,0/0	0,0/0	0/0	0/0	0/0
fbc	97k	773	2/1	1/0	1/0	1/0	1/0	1/0	1,1/0	1,0/0	0/0	0/0	0/0
Total			69/36	39/3	38/3	25/2	16/2	25/2	18,15/0	16,11/0	2/1	1/1	2/1

Figure 10. Benchmark Applications and Patch Generation Results

Apply Prophet to Defects: For each defect, we run the trained Prophet to obtain a sequence of validated plausible patches for that defect. For comparison, we also run SPR on each defect. We obtain the results of Kali [27], GenProg [15], and AE [35] on this benchmark set from previous work [27]. We terminate the execution of Prophet or SPR after 12 hours. The Kali, GenProg, and AE results are from runs that terminate when the first plausible patch validates.

To better understand how the probabilistic model, the learning algorithm, and the features affect the result, we also run five 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 defect 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), PF (a variant of Prophet with only program value features; modification features are disabled), and HL (a variant of Prophet that replaces the maximum likelihood learning with the hinge-loss learning as described in Section 3.8). 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.

We 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 the benchmark set contain multiple source files) [27]. Prophet, SPR, and Kali do not have this limitation.

Although we set 12 hours as the time limit for generating patches, for the 39 defects for which Prophet finds at least one plausible patch, Prophet requires, on average, only 108.9 minutes to find and validate the first plausible patch. For the 15 defects for which the first validated patch is correct, Prophet requires, on average, 138.5 minutes to find and validate the first correct patch.

Evaluate Validated Patches: We manually analyze each validated patch to determine whether the patch is a correct patch or just a plausible but incorrect patch that happens to produce 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 the benchmark set. A manual code analysis indicates that each of the correct patches in the experiments is semantically equivalent to the subsequent developer patch for that defect.

4.2 Patch Generation Result Summary

Figure 10 summarizes the results for Prophet, SPR, Kali, GenProg, and AE. There is a row in the table for each benchmark application. The first column (App) presents the name of the application, the second column (LoC) presents the number of lines of code in each application, and 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 ninth columns (Plausible) summarize the plausible patches that each system finds. Each entry is of the form X/Y, where X is the number of the 69 defects for which the corresponding system finds at least one plausible patch (i.e., a patch that passes the supplied test suite) and Y is the number of the 36 functionality changes for which each system finds at least one plausible patch.

The tenth through fourteenth columns (Correct) summarize the correct patches that each system finds. For Prophet and SPR, each entry is of the form X,Y/Z. Here X is the number of the 69 defects for which Prophet or SPR finds a correct patch before the 12 hour timeout (even if that patch is not the first patch to validate), Y is the number of defects for which Prophet or SPR finds a correct patch as the first patch to validate, and Z is the number of functionality changes for which Prophet or SPR finds a correct patch before the 12 hour timeout (this number is always 0 for Prophet and SPR).

For Kali, GenProg, and AE, each entry is of the form X/Y, where X is the number of the 69 defects for which the corresponding system finds a correct patch and Y is the number of the 36 functionality changes for which the corresponding system finds a correct patch.

The results show that Prophet finds a correct patch as the first to validate for more defects than SPR, Kali, GenProg, and AE (4 more defects than SPR, 14 more than GenProg, and 13 more than Kali and AE). One potential explanation for the underperformance of Kali, GenProg, and AE is that the correct Prophet and SPR patches are outside the search spaces of these systems [18, 27], which suggests that these systems will *never* find correct patches for the remaining defects.

Both Prophet and SPR find a correct patch as the first to validate significantly more often for PHP than for other applications. There are significantly more defects for PHP than for other applications. PHP also has a much stronger test suite (an order of magnitude more test cases) than other applications, which helps both Prophet and SPR find correct patches as the first to validate.

Kali, GenProg, and AE all find a correct patch for one of the functionality changes. This functionality change eliminates support for dates with two digit years. The Kali, GenProg, and AE patches

System	Corrected Defects	Mean Rank in Search Space
Prophet	18,15	Top 11.5%
Random	14,7	Top 41.8%
Baseline	15,8	Top 20.7%
MF	18,10	Top 12.2%
PF	18,13	Top 12.3%
HL	17,13	Top 17.0%
SPR	16,11	Top 17.5%

Figure 11. Comparative Results for Different Systems

all remove the block of code that implements this functionality. This patch is in the Prophet search space, but it is deprioritized in the learned model because few successful human patches only remove code.

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 (Random, 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 finds at least one correct patch. Each entry is of the form X,Y, where X is the number of defects for which the system finds correct patches (whether this correct patch is the first to validate or not) and Y is the number of defects for which the system finds a correct patch as the first patch to validate.

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 for each 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. We compute the size of the search space as the sum of 1) the number of partially instantiated patches generated by transformation schemas with abstract conditions and 2) the number of patches generated by other transformation schemas (these patches do not include abstract conditions). "Top X%" in an entry indicates that the corresponding system prioritizes the first correct patch as one of the top X% of the patches in the search space on average. We run the random search algorithm with the default random seed to obtain the results in the figure. The results are generally consistent with the hypothesis that the more highly a system ranks the first correct patch, the more correct patches it finds and the more correct patches it finds as the first patch to validate.

The results show that Prophet delivers the highest average rank (11.7%) for the first correct patch in the search space. The results also highlight how the Prophet model enables Prophet to successfully prioritize correct patches over plausible but incorrect patches — the Random and Baseline systems, which operate without a probabilistic model or heuristics, find a correct patch as the first to validate only roughly half as often as Prophet.

The results also show that the maximum likelihood learning algorithm in Prophet outperforms the alternative hinge-loss learning algorithm. One potential explanation is that the hinge-loss objective function only considers two patches: the correct patch and the incorrect patch with the highest score. The maximum likelihood objective function, in contrast, considers all of the patches. The result is that the hinge-loss-trained model does not prioritize correct patches as highly in the search space as the maximum likelihood model (also see Section 3.8).

The results also highlight how program value features are more important than modification features for distinguishing correct patches from plausible but incorrect patches. We observed a common scenario that the search space contains multiple plausible patches that operate on 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.

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 size of the search space (i.e., the sum of the number of partially instantiated patches from transformation schemas with abstract conditions and the number of patches from schemas without abstract conditions) and Y is the number of correct patches in the search space.

The third through eighth columns (First Correct Patch Rank) present 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 finds this correct patch as the first plausible patch. "△" in an entry indicates that the algorithm finds a plausible but incorrect patch before it reaches its first correct patch. "X" indicates that the algorithm fails to find any plausible patch in 12 hours.

The ninth through fourteenth columns (Correct/Plausible Patches Validated) present statistics regarding the validated correct and plausible patches that each system finds. Each entry is of the form X/Y. Y is the total number of plausible patches the corresponding system finds if we run the system on the corresponding defect exhaustively for 12 hours. X is the rank of the first correct patch to validate among these validated plausible patches. "-" indicates that the system finds no correct patch among these validated plausible patches.

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

For 10 of the remaining 14 defects, Prophet finds a correct patch as the first to validate. SPR, in contrast, finds a correct patch as the first to validate for 6 of these 14 defects. SPR empirically prioritizes candidate patches that change existing branch conditions above all other candidate patches in the search space [18]. This rule conveniently allows SPR to find a correct patch as the first to validate for php-309579-309580, php-309892-309910, and php-311346-311348.

Prophet outperforms SPR on four defects, 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 ranks the correct patch as one of top 2% in the patch validation order, while the

Defect	Search Space	First Correct Patch Rank						Correct/Plausible Patches Validated					
		Prophet	SPR	MF	PF	HL	Base.	Prophet	SPR	MF	PF	HL	Base.
php-307562-307561	29560(1)	2672✓	4918✓	304✓	4263✓	3341✓	4435✓	1/1	1/1	1/1	1/1	1/1	1/1
php-307846-307853	22131(1)	10742✓	3867✓	11139✓	10123✓	14225✓	5971✓	1/1	1/1	1/1	1/1	1/1	1/1
php-308734-308761	14536(2)	5376✓	5771✓	7525✓	4676△	8770✓	12903✓	1/4	1/4	1/4	3/4	1/4	1/4
php-309516-309535	27098(1)	10954✓	4000✓	9365✓	9442✓	14320✓	8042✓	1/1	1/1	1/1	1/1	1/1	1/1
php-309579-309580	51306(1)	767✓	46✓	1977△	492✓	2347✓	3758△	1/2	1/2	2/2	1/2	1/2	2/2
php-309892-309910	36940(4)	462✓	179✓	217△	863✓	1031✓	1530✓	1/21	1/17	4/21	1/21	1/20	1/21
php-310991-310999	87574(2)	907✓	384✓	351✓	1381✓	866✓	5061✓	1/1	1/2	1/2	1/1	1/1	1/2
php-311346-311348	8730(2)	27✓	312✓	125✓	34✓	107✓	977△	1/49	1/50	1/49	1/49	1/50	12/50
php-308262-308315	81412(1)	1365✓	7191✗	3094✓	2111✓	2836✓	6784✗	1/2	-/0	1/2	1/2	1/2	-/0
php-309688-309716	60936(1)	3465△	8398△	2226△	1396△	474△	6352△	38/47	-/17	44/50	29/57	64/73	-/25
php-310011-310050	68534(1)	1348△	30647△	5629△	3336△	2619△	5581△	6/48	-/22	-/76	8/41	-/58	-/33
php-309111-309159	52908(1)	7701△	24347△	7089△	18919△	6455△	4121△	9/10	3/10	9/10	10/10	10/10	8/10
php-307914-307915	45389(1)	1✓	5748✓	2703✓	1✓	1✓	5110✓	1/1	1/1	1/1	1/1	1/1	1/1
libtiff-ee2ce-b5691	171379(1)	280✓	13296✓	348✓	44✓	12✓	337△	1/328	1/328	1/328	1/328	1/328	2/328
libtiff-d13be-ccadf	296409(1)	1183✓	372△	4671△	1333✓	3220✓	778△	1/1423	3/1723	2/1723	1/1423	1/1723	2/1723
libtiff-5b021-3dfb3	219851(1)	50770△	56644△	8201△	62082△	119838△	132981△	-/242	206/237	178/210	-/202	-/147	-/147
gmp-13420-13421	50672(2)	14102✓	14645✓	3260✓	13085✓	17246✓	41741✓	1/3	1/3	1/3	1/3	1/3	1/3
gzip-a1d3d4-f17cbd	47602(1)	1929✓	21926△	15928△	722△	5262△	4123△	1/14	4/14	5/14	2/14	4/14	3/14
fbc-5458-5459	9788(1)	33✓	454△	741△	28✓	413△	686△	1/37	8/37	5/37	1/37	3/26	5/38

Figure 12. Per-Defect Results

SPR hand-coded heuristics rank patches that add a guard statement below patches that change a branch condition. Because of this lower rank, 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, but not all of the resulting patches are correct. Prophet successfully prioritizes 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.

4.5 Discussion

Hypothesis: The patch generation results are consistent with a key hypothesis of this paper, i.e., that even across applications, correct code shares properties that can be learned and exploited to generate correct patches for incorrect applications. The results show that by learning properties of correct code from successful patches for different applications, Prophet significantly outperforms all previous patch generation systems on a systematically collected benchmark set of defects in large real-world applications.

We note that the applications in the training set share many characteristics with the benchmark applications on which we evaluate Prophet. Specifically, all of these applications are open source Linux applications, written in C, that can be invoked from the command line. It therefore remains an open question whether this hypothesis generalizes across broader classes of programs.

Important Features: The Prophet features capture interactions between the code in the patch and the surrounding code that the patch modifies. The learning algorithm of Prophet then identifies a subset of such interactions that characterize correct patches.

We inspected the feature weights of the learned model. Features with large positive weights capture positively correlated interactions. Examples of such interactions include 1) the patch checks a value that is used as a parameter in a nearby procedure call, 2) the patch checks a pointer that nearby code dereferences, and 3) the patch checks a value that was recently changed by nearby code. These features are positively correlated because the root cause of many defects is a failure to check for a condition involving values that the surrounding code manipulates. Successful patches often insert checks involving these values. Another example of a positively correlated interaction is an inserted function call that replaces one of the parameters with a pointer that nearby code manipulates. Such patches often correct defects in which the programmer forgot to perform an operation on the object that the pointer references.

Features with large negative weights capture negatively correlated interactions. Examples of such interactions include: 1) the patch changes the value of a variable whose value is also changed by nearby code in the same block, 2) the patch checks a global variable, and 3) the patch inserts a call to a function that nearby code in the same block also calls. These features are negatively correlated because they often correspond to redundant, irrelevant, or less organized program logic that rarely occurs in successful patches.

While none of these features is, by itself, able to definitively distinguish correct patches among candidate patches, together they deliver a model that makes Prophet significantly better at finding correct patches than the previous state-of-the-art heuristics that SPR implements.

5. Related Work

We next survey related work in automatic patch generation, attack/error recovery and survival, and program beautification.

5.1 Automatic Patch Generation

SPR: SPR is the previous state-of-the-art generate-and-validate patch generation system for large applications [18]. SPR applies a set of transformation schemas to generate a search space of candidate patches. It then uses staged program repair to validate the generated patches on a test suite of test cases, at least one of which exposes a defect in the original program. 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 uses its learned correctness model to guide the exploration of the search space. Our experimental results show that this learned model 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 [33]. 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. Both CodePhage and Prophet work with correct code from one (or more) applications to obtain correct code for other applications. But Prophet does not transfer existing code. It instead works with a model of correct code learned from one set of applications to guide the exploration of an automatically generated space of candidate patches for different applications.

ClearView: ClearView is a generate-and-validate system that observes normal executions to learn invariants that characterize safe behavior [25]. 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. ClearView operates directly on stripped x86 binaries.

By monitoring the continued execution of patched applications, ClearView recognizes and rejects unsuccessful patches that do not protect the application against attacks or cause the application to fail. It also recognizes and propagates successful patches that protect the application against attacks. In this way ClearView learns whether specific deployed patches are successful in practice.

GenProg, RSRepair, AE, and Kali: GenProg [15, 36] 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 [26] replaces the GenProg genetic search algorithm to instead use random search. AE [35] 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 [27]. 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 [27]. 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 simply remove functionality [27]. The Kali patch generation system, which only eliminates functionality, can do as well [27].

Prophet differs in that it works with a richer space of candidate patches, uses learned properties of successful patches to guide its search, and generates significantly more correct patches, including correct patches that insert new logic to the patched application (as opposed to simply eliminating functionality).

PAR: PAR [12] is another automatic patch generation system. Unlike Prophet, which uses a probabilistic model and machine learning techniques to automatically learn properties of 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 [12].

Monperrus found that PAR fixes the majority of its benchmark defects with only two templates ("Null Pointer Checker" and "Condition Expression Adder/Remover/Replacer") [22]. A manual analysis indicates that the PAR search space (with the eight hand-coded templates in the PAR paper [12]) is in fact a subset of the Prophet/SPR search space [18].

Angelic Debugging: Angelic Debugging [3] 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 defect localization component.

NOPOL and SemFix: NOPOL [4, 8] applies angelic debugging to locate conditions that, if changed, may enable defective Java programs to pass the supplied test suite. It then uses an SMT solver to synthesize repairs for such conditions. SemFix [23] 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 [23] in that these systems rely only on the information from the test cases with the goal of finding plausible (but not necessarily correct) patches. Prophet learns properties of past successful patches to recognize and prioritize correct patches among multiple plausible patches.

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 [21]. In contrast, Prophet automatically learns a probabilistic model of code correctness 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.

PHPQuickFix/PHPRepair: PHPQuickFix and PHPRepair use string constraint-solving techniques to automatically repair PHP programs that generate HTML [31]. By formulating the problem as a string constraint problem, PHPRepair obtains sound, complete, and minimal repairs to ensure the patched php program passes a validation test suite.

FixMeUp: FixMeUp starts with a specification that indicates the conditional statement of a correct access-control check. It then uses program analysis to automatically find and repair missing access-control checks in Web applications [34].

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 [14]. 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 [24] is a program repair tool for the Eiffel programming language. AutoFixE leverages developer-provided formal specifications (e.g., post-condtions, pre-conditions, and invariants) to automatically find and generate repairs for defects. Cost-aware Program Repair [30] 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.

Defect Repair via Q&A Sites: Gao et. al. [10] propose to repair recurring defects by analyzing Q&A sites such as Stack Overflow. The proposed technique locates the relevant Q&A page for a recurring defect via a search engine query, extracts code snippets from the page, and renames variables in the extracted code snippets to generate patches. Prophet is different from this technique, because Prophet directly learns a probabilistic model from successful human patches and does not rely on the existence of exact defect repair logic on Q&A pages.
LeakFix: LeakFix statically analyzes the application source code and automatically inserts deallocation statements to fix memory leak errors in an applicaiton [9]. It guarantees that the new inserted deallocation will not interfere with normal executions.

5.2 Attack/Error Recovery and Survival

Failure-Oblivious Computing, RCV, and RIFL: Failure-oblivious computing [29] 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 application to continue execution along its normal execution path. The results show that this technique enables servers to continue to operate successfully to service legitimate requests even after attacks trigger their memory errors.

RCV [20] 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. The results show that this technique enables applications to continue to execute to provide acceptable output and service to its users on error-triggering inputs.

The reason behind these potentially counterintuitive results is that applications typically process inputs in units such as requests or input file lines. Each unit triggers a corresponding computation that is often relatively loosely coupled with computations for other units. Failure-oblivious computing and RCV succeed because they enable the application to survive attacks and errors without corruption to successfully process subsequent units.

The filtered iterators of RIFL provide explicit support for this pattern — they make the decomposition of the input into separate units explicit and execute the corresponding computations atomically. Input units that trigger errors or vulnerabilities are discarded [32].
Bolt: Bolt [13] 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. The results show that Bolt can enable applications to provide useful results to users even in the face of infinite loops.
DieHard: DieHard [2] 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 [7] 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 [6]. 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 [5].
Input Rectification: Input rectification learns constraints that characterize inputs that applications can process successfully. It processes each input, enforcing any violated constraints to produce a new input that it presents to the application [16]. The results show that this technique effectively converts malicious inputs into inputs that the application can process successfully while typically preserving the desirable data in the input.
SIFT: SIFT uses static program analysis to generate sound input filters that nullify integer overflow errors at security critical program points [19]. It guarantees that any input that passes the generated filters will not be able to trigger the corresponding overflow errors.
JSNICE: JSNICE [28] 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 database of JavaScript programs, a probabilistic model of relationships between either pairs of variable names (for predicting variable names) or pairs of types (for generating comments to annotate variable types). Given a new JavaScript program, JSNICE uses a greedy algorithm to search a space of predicted variable names or types, with the learned model guiding the search.

A key difference between JSNICE and Prophet is that JSNICE does not aspire to change the program semantics — the goal of JSNICE is instead to change variable names and add comments to beautify the program but leave the original semantics intact. The goal of Prophet, in contrast, is to produce correct patches that change the program semantics to eliminate defects. 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 universal properties of correct code.

6. Conclusion

Prophet automatically learns from past successful human patches to obtain a probabilistic, application-independent model of correct code. It uses this model to automatically generate correct patches for defects in real world 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.

Acknowledgements

We thank Tao Lei for valuable discussions about the learning algorithm. We thank the anonymous reviewers for their insightful comments on early draft of the paper. This research was supported by DARPA (Grant FA8650-11-C-7192 and FA8750-14-2-0242).

References

[1] GitHub. `https://github.com/`.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei. Safe memory-leak fixing for c programs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15', pages 459–470, Piscataway, NJ, USA, 2015. IEEE Press.

[10] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei. Fixing recurring crash bugs via analyzing Q&A sites. In *Proc. of ASE*, 2015.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard. Automatic input rectification. ICSE '12, 2012.

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

[18] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 166–178, New York, NY, USA, 2015. ACM.

[19] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard. Sound input filter generation for integer overflow errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14', pages 439–452, New York, NY, USA, 2014. ACM.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] H. Samimi, M. Schäfer, S. Artzi, T. D. Millstein, F. Tip, and L. J. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 277–287, 2012.

[32] J. Shen. RIFL: A Language with Filtered Iterators. Master's thesis, Massachusetts Institute of Technology, 2015.

[33] 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.

[34] S. Son, K. S. McKinley, and V. Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *NDSS*, 2013.

[35] 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.

[36] 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.

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