Anti-Patterns for Automatic Program Repairs

by

Taiyue Liu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2016

© Taiyue Liu 2016

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Automated program repair has been a heated topic in software engineering. In recent years, we have witnessed many successful applications such as *Genprog*, *SPR*, *RSRepair*, etc. Given a bug and its test suite, which includes both passed test cases and failed test cases, these tools aim to automatically generate a patch that fixes the bug without developers' efforts. All these tools adopt a *"Generate-and-Validate"* approach, which assumes a tool-generated patch to be correct as long as it passes all its test cases. However, if test suites are of poor quality that cannot cover all the situations, incorrect tool-generated patches might pass all their test cases and be regarded as correct patches. We call such patches that are incorrect but can pass whose test suites as *overfitted patches*.

In order to investigate the reasons why overfitted patches are generated and to reduce overfitted patches, we perform a deep analysis on the patches composed by developers, and the patches (i.e., the correct and the overfitted patches) that are generated by Genprog and SPR. In this thesis, we propose two orthogonal approaches to filter out overfitted patches: 1) To preserve correct tool-generated patches and filter out only overfitted patches, we propose some patterns, named *anti-patterns*, that can efficiently distinguish correct patches against overfitted patches. We select nine bugs from the Genprog benchmark data set to evaluate the anti-patterns. By embedding the anti-patterns into SPR and filtering out overfitted patches, on average, developers can review 44.7% less tool-generated patches to reach correct patches. Meanwhile, by filtering out overfitted patches at runtime, the anti-patterns speed up SPR's efficiency by 1.34 times on average. 2) We leverage machine learning techniques with meaningful features to predict the correctness of tool-generated patches. Our results show that the machine learning approach cannot preserve correct patches well. In other words, machine learning techniques would mis-classify correct patches as overfitted patches and filter them out. Thus, we believe the machine learning approach requires significant future work, e.g., more representative features and effective classification algorithms, to be useful in practice. These two orthogonal approaches provide automatic program repair tools with valuable guidance on how to avoid generating overfitted patches.

Acknowledgments

Firstly, I want to thank my supervisor–Professor Lin Tan. I can still remember the first time that I saw Lin, when she was introducing the course outline for the fourth-year testing course. Two years has passed since then, I grow up from a novice in research to a master student with my own project, while Lin is always there and offering me with advises and help. Sometimes I might feel disappointed about results, might be stuck with problems, or even hesitant, Lin is always my strong support. Start with basic testing concepts and terminology, she taught me how to manage time, how to make a good presentation, how to write a professional paper, how to conduct research independently, etc. I really appreciate that my research journey is accompanied with Lin.

Then, I want to thank Professor Werner Dietl and Professor Ladan Tahvildari. In the best and busiest season in Canada, they make time to read my thesis and provide me with valuable feedback. Thank you my labmates such as Song, Yuefei, Edmund, Jinqiu, Alex, Chenyang, Ming, and others, I really enjoy the time we spend together. No matter if it is playing Laser Quest, or just having a simple lunch in plaza, they are all my beautiful memories.

Finally, I want to thank for my parents. I know that I bring more worries than happiness to them since I went abroad, or even since I was born, but they always give me the best.

Dedication

This thesis is dedicated to the ones I love and the ones who love me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Repairing software defects is an expensive, tedious, and time-consuming task. To fix a defect, it takes time to investigate the cause of the defect and compose a correct patch, while requires developers to have a deep understanding of the software. In order to save developers' efforts and minimize the time spent fixing a defect, we have seen many successful automatic program repair tools in recent years [9–12, 27, 28, 32, 38, 41, 45, 55–57]. These automatic program repair tools aim to compose defect repairs with little or none developer effort. The difference between these tools falls into using different algorithms to mutate defective software. For example, Genprog [12] proposes to use genetic programming techniques to select mutation operations and compose patches in favor of more passed test cases, while RSRepair [45] chooses mutation operations randomly; SPR [27] receives promising results by tracking variable states in both passed test cases and failed test cases, and leveraging manually encoded templates to mutate buggy programs; Kali [46] creates patches that contain only deletions; DirectFix [32] is an oracle-based repair tool that converts a repair problem into a *Boolean Satisfiability Problem*, which is known as SAT, and tries to generate simple repairs; and there are some other algorithms such as PAR [21], SemFix [38], AE [56], Angelix [34], etc.

Normally, revealing a software defect requires a test suite containing at least one failed test case. In order to fix a defect, these tools first run fault localization algorithms to locate suspicious faulty lines by considering the stack trace of both the passed and the failed test cases. Then, the tools generate a repair search space from the context of suspicious faulty lines. The repair search space contains repair fragments (are also known as repair ingredients), which are synthesized with repair operations such as addition, deletion, and replacement to compose patches. Given a defect, these tools can generate many candidate patches for fixing it. However, not all of these patches can correctly fix the defect.

To eliminate incorrect candidate patches, most automatic program repair tools adopt an approach called *"Generate-and-Validate"*. The assumption behind this approach is, a patch that passes all its test cases is a correct patch. Given a tool-generated patch, the "Generate-and-Validate" approach re-runs the patched program on all the test cases. The patches that can pass all the test cases will be kept and produced by these automatic program repair tools.

However, due to the issue of *"weak test suite"* [28], most tool-generated patches are not correct although they pass all the corresponding test cases. We name the tool-generated patches that are not correct but can pass all the test cases as overfitted patches [49] (also referred as *plausible patches* [46]). The root cause of generating overfitted patches is that, test suites are not thorough enough to cover all situations, which allow some incorrect patches to bypass validation by test suites. This is why tool-generated patches cannot be deployed directly and require developers to review. Tools such as Genprog, SPR, RSRepair, etc. that adopt the "Generate-and-Validate", there exists a lot of overfitted patches in their search space. Manually reviewing tool-generated patches is also expensive or even more expensive than just letting developers compose a patch.

In this thesis, we manually inspect the tool-generated patches from SPR and Genprog and the patches composed by developers. We first propose some anti-patterns and embed these anti-patterns into SPR to filter out overfitted patches at runtime. Also, based on our analysis, we find some meaningful features that might differentiate between correct patches and overfitted patches. We use these features together with different supervised machine learning algorithms to predict the correctness of tool-generated patches.

In this thesis, we make the following contributions:

- Based on our analysis of the tool-generated patches from SPR, we propose anti-patterns that can capture the difference between correct patches and overfitted patches. The tool-generated patches that match any anti-patterns would be regarded as incorrect patches. We embed anti-patterns into SPR in order to filter out overfitted patches at runtime. Through our experiments, on average, developers can review 44.7% fewer tool-generated patches to reach correct patches by embedding the anti-patterns into automatic program repair tools. Meanwhile, with the proposed anti-patterns, we can preserve all the correct patches for the bugs in our selected benchmark data set.

- For the patches that match the anti-patterns, we filter them out immediately and prevent SPR from spending time on validating them. By introducing anti-patterns into SPR, we successfully improve the efficiency of SPR by 1.34 times.

- Besides the patches from SPR, for all the bugs in the benchmark data set, we also analyze their tool-generated patches from Genprog and the patches composed by developers. We predict the correctness of tool-generated patches by using different supervised machine learning algorithms together with Deckard features [15] and meaningful meta features. Our trained models predict overfitted patches with 77.8% precision, 95.5% recall, and 85.7% F-measure. However, six out of the eight correct patches are filtered out by this approach by mistake. Thus, we believe this approach requires significant future work, e.g., more representative features and effective classification algorithms, to be useful in practice.

The rest of this thesis is organized as follows. Chapter 2 introduces the backgrounds of automatic program repair tools, the "Generate-and-Validate" approach, and supervised machine learning. Chapter 3 elaborates the two proposed approaches for filtering out overfitted patches. Chapter 4 describes the setup of experiments to evaluate the proposed approaches. We show and analyze experimental results in Chapter 5. In Chapter 6 and 7, we present the threats in our work and related work respectively. Chapter 8 includes the conclusion and future work to this thesis.

Chapter 2

Background

This section provides the background of generating program repairs automatically, how overfitted patches are produced, and supervised learning.

2.1 Automatic Program Repair Tools

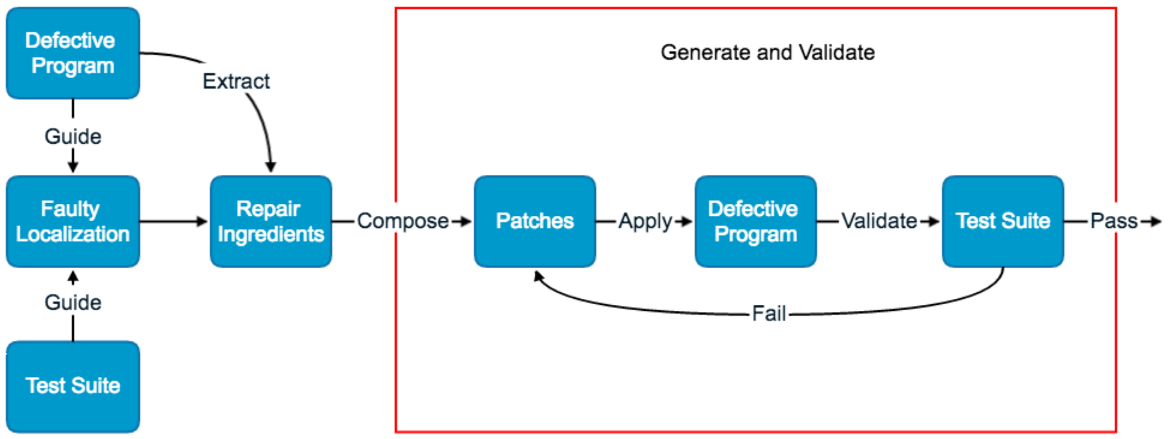

Figure 2.1: Workflow of Automatic Repair Tools

Figure 2.1 demonstrates an overview of the work-flow of search-based automatic repair tools. To fix a bug, the existing automatic program repair tools require the source code of a defective program and the test suite that reveals the bug. For the test suite, there exists at least one failed test case. According to the stack trace of both failed and passed test cases, repair tools run fault localization techniques [7,18,58] to locate the source code lines that are suspected to be buggy. Then, repair tools extract repair ingredients from the context of the suspected lines. Each repair ingredient can be a variable, an expression, a statement, or a chunk of statements. There are a variety of methods for composing patches from repair ingredients. Search-based tools such as SPR apply their templates with abstract placeholders to suspected locations and synthesizes ingredients into their templates to compose patches. Oracle-based tools like Angelix [34], DirectFix [33] and SemFix [38] convert test cases into constraints, add abstract expressions to the suspected locations, and replace the abstract expressions with ingredients by solving a Boolean Satisfiability Problem. These tools adopt the idea of the "Generate-and-Validate" approach to check the correctness of generated patches by using test suites. They will produce patches if and only if the patches can pass the corresponding test suites. As shown in Figure 2.1, the procedures in the red rectangle are for the "Generate-and-Validate" approach. Given a patch, repair tools apply it to the defective program and re-run the test suite for the corresponding bug. If the patched version of the program can pass all the test cases, the patch would be regarded as a correct patch and be produced out. Otherwise, the tools will discard this patch and keep searching for correct patches.

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

  return (1);
bad:
  if (dir)
    _TIFFfree(dir);
  return (0);
}
```

Original Buggy Version

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

Developer Patch Version

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

SPR Patch Version

Figure 2.2: Example of Plausible Patches

2.2 Problem of the "Generate-and-Validate" Approach

Most automatic program repair tools [12, 21, 27, 28, 33, 34, 38, 45, 51, 56] adopt the idea of the approach called "Generate-and-Validate". For a compilable tool-generated patch, this approach provides the tools with a validation step by running the patched program on all test cases. If a patch can pass all test cases, this patch will be produced out by the tools and be regarded as a successful fix to the corresponding bug. The tools would keep exploring search space and generating patches until they finish exploring the search space or reach the time-limit.

Although the "Generate-and-Validate" approach can filter out most incorrect patches, there are still some patches that can bypass the validation of test suites while being incorrect. We call the patches that pass their test suites but are actually incorrect as "Overfitted Patches". The root cause of generating overfitted patches is that test suites are not thorough enough to cover all situations. An overfitted patch can simply delete a functionality of its program and then pass its test suite. Figure 2.2 shows an example of overfitted patches together with its original buggy version and the correct patch from developers. This example is from Libtiff bug d13be72c-ccadf48a. The faulty lines are the if condition from line 591 to line 593. Specifically, the if condition wrongly includes a case when $td \rightarrow td\_nstrips$ is equal to 2. In the corresponding human fix, developers modify the if condition and change it from $td \rightarrow td\_nstrips > 1$ to $td \rightarrow td\_nstrips > 2$, so the case of $td \rightarrow td\_nstrips == 2$ would not be handled by the error handling code. However, the SPR patch disables the whole if block by appending && !(1) to the end of the if condition. Although there is no more warning message for the failed test case running on the patched program, the error handling code between line 678 and line 681 would not function properly anymore.

After going through all candidate repairs in search space or reaching the time-limit, the tools stop and produce all the patches that pass test suites. However, among these patches, there might be only one or two correct patches while the rest are all overfitted patches. Take libtiff bug 5b02179-3dfb33b as an example, running SPR with configurations of fully synthesis and trying first 57000 repair schemes would produce out 239 patches, which include one correct patch and 238 overfitted patches. Here, one scheme is a group of patches that modify the same faulty location with the same template, but with different contents filled in the template. SPR starts with candidate repairs with higher priorities, which depend on the templates that repairs belong to and the ranks of locations returned by fault localization algorithms. But still, for Libtiff bug 5b02179-3dfb33b, the correct patch is at position 210, which means developers need to review the correctness of 210 patches in order to reach the correct one. Sometimes, manually reviewing the correctness of tool-generated patches takes even more time and efforts than just letting developers compose a correct patch. Libtiff bug 5b02179-3dfb33b is not the only case. Among the twenty bugs that have correct patches in SPR search space, nine of them have correct patches that are not as their first patches. It would save reviewing efforts if we can filter out overfitted patches and expose correct patches earlier.

At the same time, the "Generate-and-Validate" approach takes time to apply patches, re-compile the program, and validate patches by test suites. Especially when the size of test suite is very large, validating patches will take a lot of machine time. PHP bugs have the largest test suites in the benchmark set, the test suites corresponding to PHP bugs each contains around seven thousand test cases. Filtering out overfitted patches immediately after they have been generated would save both compilation time and validation time.

2.3 Supervised Learning

Supervised learning is a category of machine learning tasks, which build classification models by learning from training data with class labels (also referred as output or signal) and predict class labels for test data. Training instances for supervised learning each contains a vector of features and a desired class label, while depending on learning algorithms, a supervised learning classification model can be a probabilistic function, a set of hyperplanes, or a decision tree that relates the features of instances to their corresponding class labels. Given a new instance with only features, supervised learning aims to correctly determine the class label for the new instance. We have seen many successful applications of supervised learning in research areas such as defect prediction [16, 17, 20, 39], tag recommendation [50], bug triaging [2, 5].

As the population of overfitted patches is big, we want to filter out overfitted patches so developers can review fewer tool-generated patches. In next chapter, we will explain the two proposed approaches for filtering out overfitted patches. For the anti-pattern approach, we explain it in aspects of how we propose the anti-patterns approach and what the anti-patterns are. For the machine learning approach, we elaborate how we label data, extract features, and build classification models.

Chapter 3

Approaches

We first perform a deep analysis on the patches that are generated by Genprog and SPR. For SPR, we manually inspected all their released patches including both correct and overfitted patches. For Genprog, we inspected more than fifty of its generated patches, which include all the correct patches and some of its overfitted patches. For each of the patches we inspected, we analyze itself and also the developers' patch for the same defect. Here, we regard the developers' patches as the ground-truth and manually check the correctness of tool-generated patches. A tool-generated patch is correct if and only if it is semantically equivalent with the patch composed by developers.

Generating a correct patch is hard for automatic repair tools. Take SPR as an example, to generate a correct patch, SPR needs to: 1) correctly find the faulty location among the hundreds of faulty locations returned by fault localization algorithms. 2) apply appropriate abstract templates to the faulty locations 3) fill in a reasonable expression from the context of the faulty location. In most cases, there exists only one correct patch in repair search space for a bug, while there are tens of thousands of schemes, each of which can be synthesized to multiple patches [29]. Due to the difficulty of generating correct patches, we want to preserve as many correct patches as possible. We are looking for some anti-patterns that are shared among only overfitted patches but not correct patches. These anti-patterns can be good indicators for the correctness of tool-generated patches and can preserve correct patches well. Our proposed anti-patterns are designed for SPR only, as SPR is a more effective tool compared with Genprog. Among the 105 bugs in the benchmark set that is proposed by Genprog, Genprog generates correct patches for only two bugs, while SPR correctly fixes twenty of them.

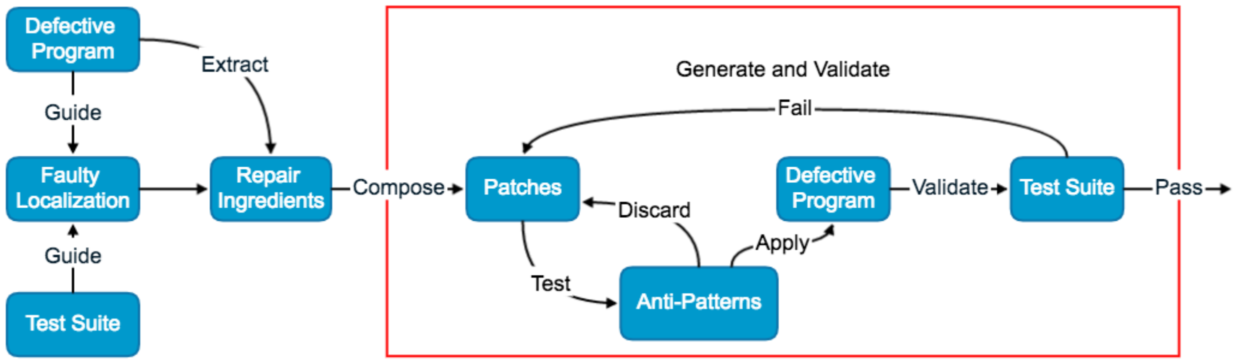

Figure 3.1: Workflow of Automatic Repair Tools with Anti-Patterns

Then, we also make an attempt to predict the correctness of the tool-generated patches by using machine learning techniques with features at both *Abstract Syntax Tree* (*AST* in abbreviation) token level and commit level. The details of the machine learning approach are explained in Section 3.2.

3.1 Anti-Patterns

We propose the anti-patterns approach that particularly aims at filtering overfitted patches out while preserving correct patches. For each overfitted patches we inspect, we brainstorm and list all the patterns that can categorize it. After we go through all the patches, we summarize and select the patterns that occur only in overfitted patches but not correct patches.

Through our manual inspection, we propose four patterns that are good indicators of the overfitted patches. Except for pattern 2 that is implemented by directly modifying SPR source code, the rest three patterns are all implemented by using *Clang* [24], which is a C language frontend.

Figure 3.1 demonstrates the workflow of SPR with the anti-patterns embedded. Besides the steps in the original workflow of automatic program repair tools in Figure 2.1, we add one more step, which is called "Anti-Patterns" in the figure, to check for the anti-patterns. We check for the anti-patterns after patches have been generated and before patches are applied to defective program. If a patch satisfies one of the anti-patterns, we discard that patch and try next patch. Otherwise, we apply the patch to the defective program and validate it by its test suite as in the original workflow.

There are two positions where we can insert the step of checking anti-patterns. Besides the one we are using, we can also insert the step after the patches have been validated by test suites. The reason why we choose the current position is that, by filtering overfitted patches out before applying them to defective programs, we can save time by not re-compiling programs. More important, we save time by not running test cases for the patches that satisfy any of the anti-patterns.

In the following, we will give details about how the patterns are shared among the patches and discuss how the patches with these patterns are composed by SPR.

3.1.1 Anti-Pattern 1: Turning an *if condition* to make it always true or always false

SPR can mutate programs to make an if code block to be always executed or never executed. In the case of always executed, when SPR found that an if code block is executed only in passed test cases while not in failed test cases, it might doubt that not executing the code block is the cause of failure. So SPR would generate a patch to make the condition of the code block always true. Similar reason for the case of never executed, when SPR found that an if code block would be executed only in failed test cases but not passed test cases, it would turn the if condition of the code block to be always false so the code block would never be executed. However, in most cases, disabling a whole if block or making an if block to be always executed is not a correct fix. Figure 2.2 is an example of making an if condition never be executed. Due to the poor quality of the test suite, this kind of overfitted patches can bypass the validation by test suites.

To detect the patches that belong to this pattern, we first locate the if condition modified by SPR. Then, for always true case, we check whether the if condition is in one of the two forms: $if(... \mid\mid (1))$ or $if(...a \mid\mid !a)$. Similar for always false, we check whether the if condition is either $if(...\ \&\&\ !(1))$ or $if(...\ a\ \&\&\ !(a))$. If the if condition is either always true or always false, we filter the patch out before executing test suites.

However, python bug 69783-69784 is a special case. The correct patch generated by SPR for this bug is shown in Figure 3.2 and it turns the if condition into always false. Python bug 69783-69784 is a functionality change instead of a bug, while we do not think this functionality change should be in the benchmark data set. The disabled if condition happens at line 341 and is for handling a input date whose year is in format of two digits.

```
  if (y < 1000) {
      ...

          //SPR generated patch
-          if (acceptval) {
+          if (acceptval && !(1)) {
              if (0 <= y && y < 69)
                  y += 2000;
              else if (69 <= y && y < 100)
                  y += 1900;
              else {
                  PyErr_SetString(PyExc_ValueError,
                                  "year out of range");
                  return 0;
              }
              ...
      }

      ...

  p->tm_year = y - 1900;
  p->tm_mon--;
  p->tm_wday = (p->tm_wday + 1) % 7;
  p->tm_yday--;
  return 1;
}
```

Figure 3.2: A Special Case of Pattern 1

Since developers decide to accept only the dates with four-digit years, they simply delete the whole if block for handling two-digit years. This is the only case in SPR patches that disable a if block while being correct.

3.1.2 Anti-Pattern 2: Adding a *return statement* to skip functionality

Another pattern that is shared among overfitted patches is adding return statements to force programs to exit before reaching buggy lines. Figure 3.3 shows two SPR patches and both of them are for the bug PHP 308262-308315. The first one is an overfitted patch that adds a return statement at line 1264, while the second patch is a correct patch that we want to preserve. As we can see, to avoid producing out the error message at line 1265, the return statement introduced by the overfitted patch simply skips the error message together with all the functionalities after the code chunk. As a result, there is no more error message and the failed test case that checks only standard errors would pass then.

```
if (Z_LVAL_P(dim) < 0 || Z_STRLEN_P(container) <= Z_LVAL_P(dim)) {
        //SPR generated patch
+       if ((type != 0))
+           return;
        zend_error((1 << 3L), "Uninitialized string offset: %ld", (*dim).value.lval);
        Z_STRVAL_P(ptr) = STR_EMPTY_ALLOC();
        Z_STRLEN_P(ptr) = 0;
}
else {
        ...

}
AI_SET_PTR(result, ptr);
return;
```

Overfitted SPR Patch

```
if (Z_LVAL_P(dim) < 0 || Z_STRLEN_P(container) <= Z_LVAL_P(dim)) {
        //SPR generated patch
+       if (!(type == 3))
                zend_error((1 << 3L), "Uninitialized string offset: %ld", (*dim).value
                    .lval);
        Z_STRVAL_P(ptr) = STR_EMPTY_ALLOC();
        Z_STRLEN_P(ptr) = 0;
}
else {
        ...

}
AI_SET_PTR(result, ptr);
return;
```

Correct SPR Patch

Figure 3.3: Example of Pattern 2

We filter out this kind of patches by modifying SPR source code and disabling generating patches in this kind. The reason why this kind of overfitted patches can be generated is, some test cases check only the return value of the program and if the program generates warning or error messages. Adding a return before the error message would skip the code that produce errors or warnings and make the test cases passed. We prevent SPR from generating the patches that satisfy this pattern.

3.1.3 Anti-Pattern 3: Adding a *comparison* with unusual value on the right side

The third pattern is appending a comparison in a form of "variable == value" to an if condition and the value on the right side of the comparison is *unusual*. We define a value

```
//SPR generated patch
- if (((dst == ((void *)0)) || (dst->data == ((void *)0)) || ((((FBSTRING *)dst)->len
    & ~2147483648UL) == 0))) {
+ if (((dst == ((void *)0)) || (dst->data == ((void *)0)) || ((((FBSTRING *)dst)->len
    & ~2147483648UL) == 0)) || (dst->len == 7)) {
    fb_hStrDelTemp_NoLock(src);
    fb_hStrDelTemp_NoLock(dst);
    FB_STRUNLOCK();
    return;
}
...

if( (start > 0) && (start <= dst_len))
  {
  --start;

  if( (len < 1) || (len > src_len) )
    len = src_len;

  if( start + len > dst_len )
    len = (dst_len - start);

  memcpy( dst->data + start, src->data, len );
}
```

Overfitted SPR Patch Version

```
...
//SPR generated patch
- if( (start > 0) && (start <= dst_len))
-   {
+ if (((start > 0) && (start <= dst_len)) && !(len == 0)) {
  --start;
  if ((len < 1) || (len > src_len))
    len = src_len;

  if( start + len > dst_len )
    len = (dst_len - start);

  memcpy( dst->data + start, src->data, len );
}
```

Correct SPR Patch Version

Figure 3.4: Example of Pattern 3

to be "unusual" if it is not global variable, constant, or previously assigned to a variable. Through our inspection, we think the value on the right side of the comparison is added by SPR should be previously defined. Figure 3.4 is an illustration of this pattern from Fbc bug 5458-5459. SPR appends ($dst \rightarrow len\ ==\ 7$) to the if condition, while 7 is never defined in the local context and the header files.

We take the overfitted patch for Fbc bug 5458-5459 as an example to explain the situation of generating such patches. SPR finds that sometimes not executing the if block is the cause of failing test cases. For the executions that the if condition is not true while SPR thinks it is necessary to execute the if block, $dst \rightarrow len$ is happened to be 7. And for the executions that the if condition is not true while SPR also agrees, $dst \rightarrow len$ is never equal to 7. So SPR appends the comparison to the end of the if condition and make the if block executed.

To detect whether a patch belongs to this pattern, we first find the comparison expression that is added by the patch. For SPR-generated comparison in this kind, the right side is always an integer value while the left side is a variable name. For this anti-pattern, we are interested in only the right side of the comparison. We parse all the header files of the patched file including the patched file itself, while recording down all the integer values that are defined. Then, we check if the integer value is defined either in the patched file or the header files. If it is not defined, we filter out the patch.

```
{
  //SPR generated patch
  if ((td->td_nstrips > 1
    && td->td_compression == 1
-    && td->td_stripbytecount[0] != td->td_stripbytecount[1]) {
+    && td->td_stripbytecount[0] != td->td_stripbytecount[1])
+    && !(td->td_subifd == 0)) {
      TIFFWarning(module, "%s: Wrong \"%s\" field, ignoring and calculating from
          imagelength", tif->tif_name, TIFFFieldWithTag(tif, 279)->field_name);
    if (EstimateStripByteCounts(tif, dir, dircount) < 0)
        goto bad;
  }
  ...

  return (1);
bad:
  if (dir)
    _TIFFfree(dir);
  return (0);
}
```

Overfitted SPR Patch

```
{
  //SPR generated patch
  if ((td->td_nstrips > 1
    && td->td_compression == 1
-    && td->td_stripbytecount[0] != td->td_stripbytecount[1]) {
+    && td->td_stripbytecount[0] != td->td_stripbytecount[1])
+    && !(td->td_nstrips == 2)) {
      TIFFWarning(module, "%s: Wrong \"%s\" field, ignoring and calculating from
          imagelength", tif->tif_name, TIFFFieldWithTag(tif, 279)->field_name);
    if (EstimateStripByteCounts(tif, dir, dircount) < 0)
        goto bad;
  }
  ...

  return (1);
bad:
  if (dir)
    _TIFFfree(dir);
  return (0);
}
```

Correct SPR Patch

Figure 3.5: Example of Pattern 4

3.1.4 Anti-Pattern 4: Adding a *variable* that does not appear in the *if condition*

This pattern does not filter out patches, instead it adjusts the ranks of the patches that are within one scheme. One scheme is a group of patches that modify the same faulty location with the same template, but with different contents filled in the template.

For the comparison appended by SPR to an if condition, the left side is always a variable. In this pattern, we favor the variable that has already existed in the if condition. Figure 3.5 is from Libtiff bug d13be72c-ccadf48a, the overfitted patch appends $!(td \rightarrow td\_subifd == 0)$ to the if condition while the correct patch appends $!(td \rightarrow td\_nstrips == 2)$ to the if condition. Since $td \rightarrow td\_nstrips$ already exists in the if condition in the buggy version, we give the correct patch a higher rank than the overfitted patches. To implement this pattern, we save all the original expressions in the if condition that is modified by SPR. Then, given a SPR patch, we check whether the variable appended by the patch has already existed. If it is, we give the patch a higher score than the other patches in the same scheme, so the correct patch can be explored earlier than overfitted patches.

3.2 Machine Learning Method

3.2.1 Data Labeling

Both Genprog and SPR released their generated patches for the bugs in the benchmark set, and for each patch, they clearly state if the patch is overfiited or correct. Meanwhile, for each bug in the benchmark set, there is a correct patch that is composed by developers. Genprog and SPR state a patch to be correct if and only if the patch is semantically equivalent to the patch that is composed by developers. We assgin the label ‘class 0’ to the patches that are correct. The correct patches include all the developers’ patches and tool-generated correct patches. And we label overfitted patches as ‘class 1’. We will explain more on how we label data in Section 4.1.2.

3.2.2 Feature Extraction

Our features are in two categories: 1) *Deckard features* and 2) *meta features*. Deckard [15] is a tree-based code clone detection tool that detects code clones by comparing the ASTs of code snippets. Since the patches generated by both Genprog and SPR are for C programs, we deploy Deckard as AST parser to analyze the patch files. Deckard can parse C source files while generating characteristic vectors of 225 features. Each feature corresponds to one AST node type, such as if condition, for loop, definition of constant, binary operators such as &&, etc., and its value records the number of occurrences of the AST node type in the analyzed file. Given a buggy file and a patch to fix it, we run Deckard on both buggy version and patched version of the file. Then, we get two sets of features, each set of features corresponds to the number of occurrences of AST nodes in one version of the file. We subtract the features of the buggy version from the features of the patched version. After that, the values of the features we get are the numbers of AST nodes modified by tool-generated patches. For example, we assume that the buggy version contains four "if condition" nodes while the patched version contains five "if condition" node. After subtraction, the value of the feature becomes one and we know the patch introduces one "if condition". Deckard features do not contain project-specific information such as method names, variable names, class names, etc.

Our meta features contain lines of addition, lines of deletion, and change in the number of unusual token sequences made by patches. For lines of addition and line of deletion, we call Linux command "diff" between buggy and patched version. We treat lines with plus sign at the beginning as addition and lines with minus sign at the beginning as deletion. We ignore empty lines and lines containing only comments. To count the number of unusual token sequences, we adopt n-gram language model. Existing work [14] shows that software is more repetitive and predictable than natural languages. Besides, Ray et al. [47] demonstrate that n-gram language model is a simple and efficient way to direct inspection efforts. We run 3-gram model on both the buggy version and patched version while empirically set the sequence size to be 5 and probability threshold to be 5×10^{-6}. The value returned from n-gram model is the number of unusual sequences, which have probabilities less than the threshold of 5×10^{-6}. Just like how we handle Deckard features, we subtract the number of unusual sequences of the buggy version from the number of the patched version. Then, the value we get is the number of unusual sequences that are introduced by a patch.

3.2.3 Classification Algorithms

We leverage the features described above together with different supervised learning algorithms to predict the correctness of tool-generated patches. The supervised learning algorithms that we use include *Alternating Decision Tree* (*ADTree* in abbreviation), *Naive Bayes*, *Logistic Regression*, and *Naive Bayes with kernel density estimator*. These learning algorithms are widely used in research [2, 5, 16, 20, 35, 50].

ADTree [43] is a special kind of decision tree. Instead of like normal decision tree that follows only one path from the root to a leaf node, ADTree follows all the paths whose decision nodes are satisfied and generates results based on all the reached leaves. Logistic regression [53] is a kind of generalized linear models but with a slight difference. Logistic regression assumes that the conditional probability of two features follows Bernoulli distribution rather than a normal distribution. Naive Bayes [48] is a simple probabilistic learning algorithm assuming that features are independent to each other and follow normal distribution, while Naive Bayes with kernel density estimator is a variant of Naive Bayes without the assumption that features follow normal distribution. We train the classification models for the selected algorithms by using a data mining software *Weka* [13], which contains state-of-the-art techniques in machine learning.

In this chapter, we propose two approaches for filtering out overfitted patches and saving developers' time on reviewing tool-generated patches. For the next chapter, we will describe how we collect datasets and conduct experiments to evaluate the two approaches.

Chapter 4

Experimental Setup

In this section, we will explain how we conduct experiments to evaluate the proposed anti-patterns and the machine learning approach.

4.1 Evaluated Data Sets and Bugs

4.1.1 Anti-Patterns Data Set

Anti-patterns are for SPR only. To evaluate the performance of our anti-patterns, we choose nine bugs from the Genprog [12] benchmark data set, which contains 69 defects and 36 functionality changes. For the bugs that we select, there are two requirements: 1) there are correct patches existing in SPR search space. 2) the correct patch is not the first patch that passes all the test cases. For the bugs with correct patches as the their first generated patches, they already have optimal results and there is no need to apply anti-patterns on them. Note that, our patterns would not filter out any correct patches for this kind of bugs. For the bugs without correct patches in their search space, our patterns can filter out some overfitted patches. But since there is even no correct patch, our patterns cannot help developers reach the correct patches earlier and so we do not examine our patterns on these bugs. There are nine bugs that satisfy our requirements.

Table 4.1 shows the nine bugs that we try to improve by embedding anti-patterns into SPR. Column 1 lists the names of the projects and Column 2 is the bug IDs for the bugs we try to improve on. The nine bugs are from five different projects with different functionalities such as web programming (*Php*), image processing (*Libtiff*), data compression

Table 4.1: Evaluated Projects and Bugs for Anti-Patterns

Project	Bug ID	# PassingTCs	#FailingTCs	Loc(k)
PHP	308262-308315	6956	1	1046
	309111-309159	6703	1	
	309688-309716	6972	1	
	310011-310050	6989	1	
Libtiff	5b02179-3dfb33b	64	9	77
	d13be7-ccadf4	32	3	
Fbc	5458-5459	466	1	97
Python	69783-69784	306	1	407
Gzip	a1d3d4-f17cbd	2	1	491

(*Gzip*), compilers (*Fbc*, *Python*). Column 3 and 4 are the numbers of passing test cases and failing test cases respectively. The total number of test cases varies among bugs. Php bug 310011-310050 has the largest test suite with 6956 passing test cases and 1 failing test case, while Gzip bug a1d3d4-f17cbd has only 2 passing test cases and 1 failing test cases. The last column shows the lines of code for the projects. The sizes of projects range from 77 thounsand lines of code to 1046 thousand lines of code.

We embed our patterns into SPR source code. The details about how we design anti-patterns are shown in Section 3.1.

4.1.2 Machine Learning Data Set

To evaluate the machine learning approach for filtering out overfitted patches, we adopt the benchmark set released by Genprog. The benchmark set contains 69 defects and 36 functionality changes. Besides the five projects we mentioned in Section 4.1.1, there are also some defects and functionality changes that are from projects *Gmp*, *Lighttpd*, and *Wireshark*, which are math library, web server, and network analysis tool respectively. Both Genprog and SPR released their generated patches. We collect Genprog patches, SPR patches and also developers' patches for the bugs in the benchmark set. Table 4.2 shows details about our data. We collect and use Deckard to parse 423 patches from Genprog, 71 patches from developers and 30 patches from SPR. Among the 423 Genprog patches, 5 of them are correct, while 8 out of 30 SPR patches are correct. We regard all the developers' patches as correct patches.

Table 4.2: Evaluated Patches for Machine Learning Models

Source	Total	Correct
Developer	71	71
Genprog	423	5
SPR	30	8
PHP	158	158

Due to the number of correct patches versus the number of overfitted patches is too imbalanced, we also collect developers' patches from project PHP by a similar approach with the previous studies [22, 23, 36, 40] about bug fixes. More specifically, we first clone PHP source code and call "git log" to receive information of all git commits. Then, we parse commit messages and search for phrases such as "fixed bug #" and "fix bug #". There are also some other phrases that might indicate a bug fix, but due to the consideration of precision, we do not use them. To make sure that the commits we get are indeed bug fixes, we search the bug ids succeeding '#' in the bug tracking system of PHP. We keep only the commits that have bug reports in the tracking system. Meanwhile, we do not consider the bugs have multiple fix commits. For each of the commits, we have a patch and two versions that one is before the patch and one is after the patch. We treat all the collected PHP patches as correct patches. Finally, we label the instances of overfitted patches as 'class 1' and the instances of correct patches as 'class 0'. So in total, we have 440 'class 1' instances and 242 'class 0' instances.

4.2 Evaluation Setup and Metrics

4.2.1 Anti-Patterns Setup and Metrics

To investigate the performance of anti-patterns, we embed anti-patterns into SPR and rerun experiments on the selected nine bugs. We care about only the overfitted patches that are explored earlier than their corresponding correct patches. For the overfitted patches that are explored after the correct patches, their existences do not increase developers' workload for reviewing the correctness of tool-generated patches, since we assume that developers would stop at the first correct patches. Through our experiments, we want to know: 1) how many overfitted patches that happen before the correct patches can be filtered out by anti-patterns. The more overfitted patches are filtered out, the less time and efforts would be spent on reviewing tool-generated patches. 2) whether the correct patches can be preserved and pass the validation from anti-patterns. 3) if the anti-patterns introduce significant runtime overhead to SPR or if the anti-patterns can improve the efficiency of SPR.

4.2.2 Machine Learning Setup and Metrics

We use Genprog patches, Developer patches and PHP as our training set to build machine learning models and use the models to predict the correctness of SPR patches. The four learning algorithms that we use to train machine learning model are explained in Section 3.2.3. Meanwhile, to make training data more balanced, We also perform *SMOTE*, which is a data re-sampling technique that is used by existing work [44] in defect prediction, on our training set to make ratio of overfitted patches versus correct patches become roughly 1:1.

To measure the performance of the machine learning models, we use three metrics, which are *Precision*, *Recall*, and *F-measure*. These three metrics are widely used in defect prediction work [16, 30, 31, 37, 54] to evaluate the performance of prediction models. The following is an brief introduction for the three metrics:

$$Precision = \frac{true\ positive}{true\ positive + false\ positive} \tag{4.1}$$

$$Recall = \frac{true\ positive}{true\ positive + false\ negative} \tag{4.2}$$

$$F - measure = \frac{2 * Precision * Recall}{Precision + Recall} \tag{4.3}$$

To calculate precision, recall, and F-measure, we need three numbers, which are *true positive*, *false positive*, and *false negative*. True positive is the number of overfitted patches that are correctly predicted as overfitted patches, while false positive is the number of correct patches that are wrongly labeled as overfitted patches. False negative measures the number of overfitted patches that are wrongly labeled as correct patches. A higher precision means that we mis-classify fewer correct patches as overfitted patches. A higher recall means that we can save more developer's effort on reviewing the correctness of overfitted patches.

Based on the description above, we conduct experiments to evaluate the two approaches. The experimental results are shown in next chapter together with analysis.

Chapter 5

Experiment Results

Based on the analysis that we perform on developers, Genprog, and SPR patches, and the results of running anti-patterns and machine learning approach, we are willing to answer the following research questions:

- RQ1: Are the proposed anti-patterns effective in filtering out overfitted patches?
- RQ2: What is the runtime overhead of introducing anti-patterns to automatic program repair tools?
- RQ3: Can the machine-learning-based approach filter out overfitted patches effectively?

5.1 RQ1: Are the proposed anti-patterns effective in filtering out overfitted patches?

To evaluate the performance of the proposed anti-patterns, we embed them into SPR source code and re-run experiments on the selected nine bugs. As we mentioned in Section 4.1.1, the selected nine bugs all have correct patches in their search space but there are overfitted patches explored before the correct patches. Table 5.1 are the results from our experiments. The first and second column list project names and bug IDs respectively. Column "Try-at-Least" shows the number of schemes that we try in order to reach the correct patches in search space. Column 4 shows the total number of patches that are generated by trying the

Table 5.1: Performance for Anti-Patterns

Project	Bug ID	Try-at-Least	# Patches	# OrigOverfit	# NewOverfit	FilterRate
PHP	308262-308315	8000	4	3	1	66.7%
	309111-309159	25000	3	2	1	50%
	309688-309716	8500	145	101	97	4.0%
	310011-310050	32000	181	51	27	47.1%
Libtiff	5b02179-3dfb33b	57000	239	209	163	22.0%
	d13be-ccadf	400	309	14	0	100%
Fbc	5458-5459	500	24	13	7	46.2%
Gzip	a1d3d4-f17cbd	8000	4	3	3	0%
Python*	69783-69784	80	6	3	1	66.7%
Total	–	–	915	399	300	44.7%

number of schemes specified in Column 3. Column "# OrigOverfit" shows the number of overfitted patches that happen before the correct patches without the anti-patterns. For a bug, we care about only the overfitted patches that happen before the correct patch, since those overfitted patches are the patches that require developers to review. We assume that developers would stop reviewing when they find the correct patch. Column "# NewOverfit" shows the number of overfitted patches that happen before the correct patches with the anti-patterns have been embedded. From the table, we can see that overfitted patches are the overwhelming majority of tool-generated patches. For the 915 tool-generated patches that SPR generates, there are only nine correct patches while the rest are all overfitted patches. Among the 399 overfitted patches that happen before the correct patches, we filter out 99 of them, in other words, we filter out 24.8% overfitted patches. Except for python 69783-69784, the anti-patterns successfully preserve all the correct patches not only for the selected nine bugs, but also for the bugs that have correct patches being the first generated patch. As we explained in Section 3.1, python 69783-69784 is a special case since it is functionality change instead of a bug. Column "FilterRate" demonstrates how much reviewing effort that the anti-patterns can save for developers. The performance of the anti-patterns varies depending on bugs. For Libtiff bug d13be-ccadf, the anti-patterns successfully expose the correct patches as the first generated patch. However, for the bugs like Gzip a1d3d4-f17cbd, none of its overfitted patches can match our anti-patterns. On average, with the anti-patterns embedded, developers can review 44.7% less tool-generated patches to reach correct patches.

Table 5.2 shows the performance of each individual anti-pattern. The first three columns are identical to Table 5.1, while Column 4, 5, and 6 list the number of overfitted patches

Table 5.2: Performance of Individual Anti-Patterns

Project	Bug ID	Try-at-Least	Pattern1	Pattern2	Pattern3	LostFun
PHP	308262-308315	8000	0	2	0	Less
	309111-309159	25000	0	0	1	Similar
	309688-309716	8500	2	0	2	Similar
	310011-310050	32000	1	3	20	Less
Libtiff	5b02179-3dfb33b	57000	22	15	9	Less
	d13be-ccadf	400	1	0	0	Less
Fbc	5458-5459	500	0	0	6	Similar
Gzip	a1d3d4-f17cbd	8000	0	0	0	Similar
Python*	69783-69784	80	1	0	1	Less
Total	–	–	27	20	39	–

that are filtered out by each anti-pattern correspondingly. Here, we do not show the performance of pattern 4, because pattern 4 just re-ranks the patches in the same scheme instead of filtering patches out. For pattern 4, it ranks the correct patch for Libtiff d13be-ccadf as the first patch while moves 13 overfitted patches, which are in the same scheme with the correct patch, to the back of the correct patch. The last row is the summary for each anti-pattern. As we can see, these three anti-patterns are widely shared among overfitted patches and are not specific to certain bugs. The three patterns filter out 27, 20, and 39 overfitted patches respectively.

Meanwhile, we evaluate the quality of the SPR-generated patches with the anti-patterns embedded. For a bug, we compare its new first patch, which is the first patch that is generated by SPR with the anti-patterns, with its old first patch, which is the first patch that is generated by SPR without the anti-patterns, to see if the new first patch deletes more, less, or a similar extent of functionalities than the old first patch does. For example, if the old first patch is appending && !(1) to an if condition, while the new first patch is inserting a new statement or appending &&!(a == 1) to an if condition. In this case, we think the new patch remove less functionalities than the old patch. Column "LostFun" shows the results of the comparisons. Five out of nine new first patches delete less functionalities than their old first patches do, while the rest delete functionalities to a similar extent as the old first patches do. None of the new first patches deletes more functionalities comparing with the old first patches do. Tool-generated patches that delete less functionalities can narrow down the search scope for the correct fix locations and better guide developers to find a fix.

The proposed anti-patterns are widely shared among overfitted patches but not correct patches. On average, the anti-patterns can reduce 44.7% tool-generated patches for developers to review, while being able to preserve the correct patches for bugs. Meanwhile, the anti-patterns prevent tool-generated patches from removing functionalities.

5.2 RQ2: What is the runtime overhead of introducing anti-patterns into automatic program repair tools?

To answer this question, we conduct another set of experiments on the nine bugs. For each bug, we re-run SPR on it without and with anti-patterns separately. Different from the configurations for the experiments in Section 5.1, here we try one thousand repair schemes for each run. Table 5.3 shows the time needed to finish exploring the first one thousand schemes. Column "PureSPR" is the time needed for the original version of SPR, while Column "PatternSPR" is for the SPR with anti-patterns embedded. To reveal how much time we can save by introducing the patterns, we divide the time in Column 3 by the time in Column 4 and see how many times anti-patterns can speed up SPR. Take Libtiff d13be7-ccadf4 as an example, exploring the first one thousand schemes takes 2 hours, 33 minutes and 1 second for original version of SPR, and 1 hour 5 minutes and 36 seconds for SPR with anti-patterns embedded. The SPR with anti-patterns runs 1.41 times (1583 seconds / 1119 seconds) faster than the original SPR. The reason why anti-patterns can save machine time is, the patches that match anti-patterns would be filtered out immediately, and would not proceed to "Generate-and-Validate" step. In other words, the patterns save time for running test suite on the detected overfitted patches. However, we also see some cases like PHP 309111-309159 and PHP 310011-310050, that the SPR with anti-patterns runs more slowly than the original SPR. This is because most SPR-generated patches do not satisfy anti-patterns and testing anti-patterns on patches introduces extra overhead to SPR. On average, embedding the proposed anti-patterns into SPR improve SPR's efficiency by 1.34 times.

Introducing the proposed anti-patterns to SPR can greatly improve the efficiency of SPR. On average, the anti-patterns speed up SPR by 1.34 times.

Table 5.3: Time Cost for Introducing Anti-Patterns (h:hour, m:minute, s:second)

Project	Bug ID	PureSPR (h:m:s)	PatternSPR (h:m:s)	SpeedUp
PHP	308262-308315	13:02:31	9:08:07	1.43
	309111-309159	1:32:32	1:50:12	0.83
	309688-309716	2:56:46	2:08:19	1.38
	310011-310050	1:51:29	3:59:21	0.47
Libtiff	5b0217-3dfb33	0:27:37	0:18:39	1.41
	d13be7-ccadf4	2:33:01	1:05:36	2.33
Fbc	5458-5459	1:48:04	1:10:37	1.53
Python	69783-69784	1:10:23	0:53:27	1.32
Gzip	a1d3d4-f17cbd	0:09:02	0:05:40	1.36

5.3 RQ3: Can the machine-learning-based approach filter out overfitted patches effectively?

To answer this question, we train machine learning models by using Genprog patches, Developers' patches, and PHP patches and leverage the trained models to predict the correctness of SPR patches. In our training set, we have 418 overfitted patches and 234 correct patches. Note that, we also perform a data re-sampling technique named SMOTE on our training set to make ratio of overfitted patches versus correct patches become roughly 1:1. Table 5.4 shows the performance of machine learning models that are trained with different learning algorithms. The first Column shows the algorithms that we use and the rest three columns are the precision, recall, and F-measure for the corresponding models. The learning algorithms that we use are ADTree, Naive Bayes, Logistic Regression, and Naive Bayes with kernel density estimator (represented as "NaiveK" in the table). As we can see, the results vary with algorithms. The best result happens to the model trained with NaiveK, which predicts overfitted patches with 77.8% precision, 95.5% recall, and 85.7% F-measure. More specifically, 21 out of 22 overfitted patches are predicted as overfitted patches correctly. However, we lose 6 out of 8 correct patches.

As we mentioned in Section 3, generating a correct patch is difficult and costly. From the point of mis-classifying correct patches, using machine learning models to predict the correctness of tool-generated patches is not a practical way.

Table 5.4: Performance for Machine Learning Models

Algorithms	Precision	Recall	F-measure
ADTree	62.5%	22.7%	33.3%
Naive	76.0%	86.4%	80.9%
Logistic	61.1%	50.0%	55.0%
NaiveK	77.8%	95.5%	85.7%

Using machine learning techniques with AST token-level features and meta-features to reduce overfitted patches is impractical due to the mis-classification of correct patches. We believe the machine learning approach requires significant future work, e.g. more representative features and effective classification algorithms, to be useful in practice.

Chapter 6

Threats

6.1 Data Set Selection

For the selection of data set, we choose the data set proposed by Genprog. There are 69 defects and 36 functionality changes in this data set, which is well-maintained and widely used by most existing work in automatic program repair tools [12,27,28,34]. In our experiments, we choose nine bugs from the data set. The nine bugs are from five different projects with different functionalities as we mentioned in Section 4.1.1. Besides these five projects, there are also some bugs from projects gmp, lighttpd, and wireshark. All of these projects have abundant test cases for running "Generate-and-Validate" approach.

At the same time, there exists other data sets. For example, Goues et al. [25] introduces two new data sets in terms of "MANYBUGS" and "INTROCLASS". "MANYBUGS" is an extension to the data set that we select, while "INTROCLASS" consists of 998 defects from the programs that are written by the students from a C introductory programming course.

For fair comparison with existing work such as SPR, we choose Genprog data set as our analyzed target. The performance of our machine learning features may vary depending on the choice of data set. Meanwhile, it is possible that the performance of the anti-patterns fluctuates between projects, although we do not find any clues that the anti-patterns are specific to certain projects through our experiment results.

6.2 Repair Tool Selection

Based on our inspection of the patches from a successful search-based automatic program repair tool SPR, we propose the anti-patterns. The reason why we choose SPR to embed and test the anti-patterns is that, SPR is a more successful tools than Genprog as we mentioned in Chapter3. Due to the variety in methodologies of generating repairs, our proposed patterns might not be all applicable to other repair tools. For example, pattern 2 does not apply to the oracle-based repair tool Angelix [34] since Angelix would not generate a patch to add a return statement, but pattern 1 and pattern 3 are still applicable.

Meanwhile, depending on the repair tools to which anti-patterns are applied, the position where we embed anti-patterns might be different. There exists differences in the workflow of automatic program repair tools, and so anti-pattern might speed up repair tools more or less comparing with they speed up SPR. Take SemFix [38] as an example, SemFix is an oracle-based repair tool that converts test cases into constraints and transforms a program repair problem to a SAT problem. By solving the SAT problem, SemFix can find a program change that makes all test cases passed. We can still check anti-patterns right after patches have been generated, but anti-patterns would not save time by not rerunning test suites. Because the test cases are already converted to constraints at the step of generating patches and there is no more validation step after patches have been generated.

Chapter 7

Related Work

7.1 Automatic Repair Tools

In recent years, we have witnessed many successful automatic program repair tools [1, 6, 8, 12, 21, 27, 33, 34, 38, 42]. All these repair tools aim to generate patches to fix software defects with little or none developers' efforts. To fix defects, repair tools usually require the source code of buggy files and the test suites that can reveal bugs. These repair tools vary in the algorithms of generating patches.

Debroy and Wong [9] propose an intriguing question: if mutating a correct program can result in a realistic fault, is it possible to mutate a faulty program to generate a realistic fix? They deploy mutation technique to generate possible repairs. As generating mutants can be expensive, they combine mutation techniques with fault localization techniques to reduce repairing cost. Their evaluation on the programs from the Siemens suite and Java Ant proves the effectiveness of their strategy.

ClearView [42] is a generate-and-validate system that learns invariants by observing normal executions. When a failure is detected, ClearView finds a set of correlated invariants that characterize normal and erroneous executions, and generates a set of candidate patches to enforce the correlated invariants. ClearView selects the most successful patch based on the continued execution, aiming at minimizing negative effects.

Nopol [10] is dedicated to repair buggy if-conditions and missing preconditions. Given a program and its test suite, Nopol first locates buggy statements and identifies fix oracles. During test suite execution, Nopol collects the context of conditional expressions and their expected values in each test case, then transforms them into a Satisfiability Modulo Theory (SMT) problem. The solutions to the SMT problem are converted to patches.

GenProg [12] uses genetic programming technique [3,4] to to select mutation operations that make more test cases passed. The mutations operations include copy, replacement and deletion. Meanwhile, Genprog introduces the benchmark data set containing 69 defects and 36 functionality changes. This data set is widely adopted by the existing work in automatic program repair.

The authors of PAR [21] manually inspect more than 60,000 human-written patches and propose six fix patterns that are widely adopted (around 30%) in real patches. These patterns are encoded as fix templates to generate repairs.

RSRepair [45] uses the same mutation operations as GenProg's. But instead of using genetic programming to select mutation operation, it deploys random search, which gives equal chance to all the mutations operations. Different from Genprog that keeps adding more operations to complement patches, RSRepair discards invalid patches immediately after they has been generated and so RSRepair requires less test case executions. Their evaluation on 7 programs shows that RSRepair improves Genprog in aspects of both efficiency and effectiveness.

AE [56] defines a model to measure the cost of generating a patch based on size of the fault space, size of the fix space, and test executions. Besides, AE checks the equivalence between patches and so reduces repair search space greatly.

SPR [27] is a state-of-art automatic program repair tool. To fix a defect, SPR first mutates the suspicious faulty lines by using some manually-encoded templates with an abstract variable inserted. Then, SPR assigns a sequence of values to the inserted abstract variable and try to find a sequence of values that can make all the test cases passed. At the same time, SPR records states of the variables in the context and search for a variable whose values can match the found sequence of values. If SPR finds such a variable, SPR would replace the abstract variable with the found variable and generate a patch. The technique proposed by SPR can greatly reduce repair search space and generate more meaningful candidate repairs.

Prophet [28] is an expansion to SPR. Prophet adopts the same defect localization algorithm and the same framework of generating repairs with SPR. Moreover, Prophet has one more step to rank candidate repairs by learning from developer patches in aspects of variable invocations, mutation operations and fix locations. By testing on the same benchmark data set, Prophet successfully exposes correct patches earlier comparing with SPR.

SemFix [38] is an oracle-based repair tool. It converts test cases into repair constraints via symbolic execution and solve the repair constraints using program synthesis. DirectFix [32] believes simple fixes are the best fixes. Unlike SemFix that takes fault localization and repair as separate stages, DirectFix integrates fault localization and repair generation, and tries to generate a patch as simple as possible. Angelix [34] is a more scalable oracle-based repair tool, which can generate a repair that modifies several faulty locations. Relifix [51] is a repair tool designed for fixing regression bugs through manually-encoded fix patterns.

Concurrent to my thesis work, Tan et al. [52] proposes a similar pattern-based approach to address the overfitted patch problem. In their work, they propose six anti-patterns, while four of them are for Genprog and the rest two are for SPR. They do not have our anti-pattern 3 and anti-pattern 4. For the two anti-patterns that are dedicated to SPR, they name them as *Anti-append Trivial* and *Anti-append Early Exit* respectively. For their Anti-append Trivial pattern, it corresponds to our anti-pattern 1. However, it considers only the cases when SPR turns an if condition to $if(...\ ||\ (1))$ or $if(...\ \&\&\ !(1))$, while it does not consider the cases of $if(...\ a\ \&\&\ !(a))$ or $if(...a\ ||\ !a)$. Anti-append Early Exit pattern corresponds to our anti-pattern 2 with a slight difference. Their Anti-append Early Exit pattern filters out not only the patches that introduce return statements, but also the patches that introduce control statements such as *goto* and *break*. Through our inspection on SPR patches, we believe their Anti-append Early Exit pattern would filter out the correct patch of Php bug 308734-308761, which is another bug in the Genprog data set. The correct SPR-generated patch of Php bug 308734-308761 introduces a break statement guarded by an if condition and it is the first patch that passes all the test cases.

7.2 Limitations of "Generate-and-Validate" Approach

Many prior studies [8,12,19,26,32,38,41] indicate that "Generate-and-Validate" approach is feasible and efficient. However, since test cases are limited and imperfect, "Generate-and-Validate" approach is not a perfect validation for the correctness of patches. Existing work [27, 46, 49] has shown that there exists a significant population of tool-generated patches that can pass their test suites but are actually incorrect. Due to the poor quality of test suites, Repair tools like GenProg, RSRepair, AE, etc. generate only zero or one correct patch for each bug, whereas most generated patches are overfitted [46]. Recent work [49] also exposes the limitations of "Generate-and-Validate" approach. In this research, two independent test suites are deployed to produce and validate tool-generated patches separately. One of the two test suites is called training test suite and used for generating patches. If a patch can pass all the test cases in the training test suite, it would be tested by the other test suite. If a patch passes the validation by training test suite but fails to pass the validation by the other test suite, the patch is regarded as overfitted to its training test suite. The results show that patches are always overfitted to their training test suites. Moreover, even if test suites are of high quality and have high coverage, there still exists overfitted patches.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

Automatic program repair tools aim to fix software defects with little or none developers' efforts. Sometimes, the tools require developers to specify the files that are suspected of being buggy, since fault localization algorithms cannot locate the proper buggy files and thus mislead the repair tools to generate irrelevant search space. But still, a successful repair tool is expected to greatly save developers' efforts. However, due to the existence of overfitted patches, which are the patches that can pass the validation by test cases but are actually incorrect, tool-generated patches cannot be deployed directly and require developers to review their correctness before applying. As the population of overfitted patches is big, reviewing tool-generated patches takes even more time and efforts than just letting developers compose a correct patch.

In this thesis, we first perform a deep analysis on the patches generated by SPR, Genprog, developers for the bugs in the benchmark set proposed by Genprog. During our inspection on both overfitted patches and correct patches, we are looking for patterns and features that significantly differentiate between overfitted patches and correct patches. We propose two approaches to address the challenging overfitted patch problem in the automatic program repair area. The two approaches are the anti-patterns approach and the machine learning approach, which are believed to reduce overfitted patches and save developers' efforts on reviewing tool-generated patches.

In aspect of anti-patterns approach, we propose four anti-patterns that are shared among overfitted patches but not correct patches. These anti-patterns can be embedded in automatic program repair tools. A tool-generated patch that matches any anti-patterns would be filtered out immediately after it has been generated. To evaluate the anti-pattern approach, we embed the anti-patterns into a successful repair tool named SPR and select eight bugs and one functionality change that have potential to generate correct patches as their first patches from the widely-adopted benchmark data set proposed by Genprog. Through our experiments, on average, the anti-patterns reduce 44.7% tool-generated patches for developers to review, while preserve the correct patches for all the bugs (except for python 69783-69784, which is the functionality change) in the benchmark set. At the same time, by filtering out overfitted patches at runtime and not running the "Generate-and-Validate" approach on the filtered patches, the anti-patterns approach greatly improves the efficiency of SPR by 1.34 times on average.

For the machine learning method, we adopt 228 features to capture the differences between overfitted patches and correct patches. The 228 features belong to two categories, which are AST token-level features and meta features. We leverage Deckard to extract AST token-level features to capture syntactic changes that are introduced by patches, while use heuristics and n-gram model to extract meta features to capture the abnormal operations in patches. We use the trained machine learning models to predict the correctness of tool-generated patches. Our best results are achieved by using the learning algorithm Naive Bayes with kernel density estimator, with 77.8% precision, 95.5% recall, and 85.7% F-measure. However, among the eight correct patches, the trained model mis-classifies six of them. Due to the mis-classification of the correct patches, we believe that using machine learning method to reduce overfitted patches requires significant future work, e.g., more representative features and effective classification algorithms, to be useful in practice.

8.2 Future Work

8.2.1 Anti-Patterns

Besides the four anti-patterns which we described in Chapter 3, we have other anti-patterns that are also widely shared among overfitted patches. The reason why we do not include them into the anti-pattern approach is that, we want to preserve correct patches but they might wrongly filter out correct ones. This is a trade-off between preserving more correct patches, or filtering out more overfitted patches. For the future work on the anti-patterns approach, we will perform a deeper analysis on the anti-patterns that might filter out correct patches, to see if it is worth to include them.

Meanwhile, based on the insights given by the anti-patterns, we want to revise the repair templates to make them more precise. More precise repair templates could prevent overfitted patches from being generated at root and help repair tools save time by trying fewer repair candidates.

8.2.2 Machine Learning

Currently, our models are trained by the patches from developers, Genprog, and PHP, while Genprog is the only source of overfitted patches in the training set. Since Genprog and SPR use different repair algorithms and repair templates to compose patches, there might be some differences in their generated patches and training models with Genprog patches might not capture the characteristics of SPR patches well. In the future, we will replace the Genprog patches in the training set with the overfitted patches from SPR. In addition, we will collect more human patches from different projects to expand our training set.

Encoding the anti-patterns as features is another direction for our future work. As the proposed anti-patterns are all good indicators of patches' correctness and are proved to be efficient to discriminate correct patches from overfitted patches, we want to encode the anti-patterns as features, add them to machine learning models, and test their efficiency as features in the future.

Bibliography

[1] M. Alkhalaf, A. Aydin, and T. Bultan. Semantic differential repair for input validation and sanitization. In *ISSTA 2014*, pages 225–236, 2014.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE 2006*, pages 361–370, 2006.

[3] A. Arcuri. On the automation of fixing software bugs. In *ICSE 2008*, pages 1003–1006, 2008.

[4] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *CEC 2008*, pages 162–168, 2008.

[5] G. Canfora and L. Cerulo. Supporting change request assignment in open source development. In *SAC 2006*, pages 1767–1772, 2006.

[6] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for web applications. In *FSE 2010*, pages 237–246, 2010.

[7] S. Chandra, E. Torlak, S. Barman, and R. Bodík. Angelic debugging. In *ICSE 2011*, pages 121–130, 2011.

[8] Z. Coker and M. Hafiz. Program transformations to fix C integers. In *ICSE 2013*, pages 792–801, 2013.

[9] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *ICST 2010*, pages 65–74, 2010.

[10] F. Demarco, J. Xuan, D. L. Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with SMT. In *CSTVA 2014*, pages 30–39, 2014.

[11] D. Gopinath, S. Khurshid, D. Saha, and S. Chandra. Data-guided repair of selection statements. In *ICSE 2014*, pages 243–253, 2014.

[12] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, pages 54–72, 2012.

[13] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, pages 10–18, 2009.

[14] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *ICSE 2012*, pages 837–847, 2012.

[15] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE 2007*, pages 96–105, 2007.

[16] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *ASE 2013*, pages 279–289, 2013.

[17] X. Jing, F. Wu, X. Dong, F. Qi, and B. Xu. Heterogeneous cross-company defect prediction by unified metric representation and cca-based transfer learning. In *ESEC/FSE 2015*, pages 496–507, 2015.

[18] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *PLDI 2011*, pages 437–446, 2011.

[19] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search. In *ASE 2015*, pages 295–306, 2015.

[20] T. M. Khoshgoftaar and N. Seliya. Tree-based software quality estimation models for fault prediction. In *METRICS 2002*, pages 203–214, 2002.

[21] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *ICSE 2013*, pages 802–811, 2013.

[22] M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *ICSE 2011*, pages 151–160, 2011.

[23] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, pages 181–196, 2008.

[24] C. Lattner. Llvm and clang: Advancing compiler technology. In *FOSDEM 2011*, 2001.

[25] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. T. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of C programs. *IEEE Trans. Software Eng.*, pages 1236–1256, 2015.

[26] P. Liu, O. Tripp, and C. Zhang. Grail: context-aware fixing of concurrency bugs. In *FSE 2014*, pages 318–329, 2014.

[27] F. Long and M. Rinard. Staged program repair with condition synthesis. In *ESEC/FSE 2015*, pages 166–178, 2015.

[28] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *POPL 2016*, pages 298–312, 2016.

[29] F. Long and M. C. Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *ICSE 2016*, pages 702–713, 2016.

[30] C. D. Manning and H. Schütze. *Foundations of statistical natural language processing.* MIT Press, 2001.

[31] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, pages 308–320, 1976.

[32] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *ICSE 2015*, pages 448–458, 2015.

[33] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *ICSE 2015*, pages 448–458, 2015.

[34] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *ICSE 2016*, pages 691–701, 2016.

[35] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *TSE 2007*, pages 2–13.

[36] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *ICSM 2000*, pages 120–130, 2000.

[37] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang. TBCNN: A tree-based convolutional neural network for programming language processing. *CoRR*, 2014.

[38] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In *ICSE 2013*, pages 772–781, 2013.

[39] T. T. Nguyen, T. N. Nguyen, and T. M. Phuong. Topic-based defect prediction (nier track). In *ICSE 2011*, pages 932–935, 2011.

[40] J. Park, M. Kim, B. Ray, and D. Bae. An empirical study of supplementary bug fixes. In *MSR 2012*, pages 40–49, 2012.

[41] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. *IEEE Trans. Software Eng.*, pages 427–449, 2014.

[42] J. H. Perkins, S. Kim, S. Larsen, S. P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. Wong, Y. Zibin, M. D. Ernst, and M. C. Rinard. Automatically patching errors in deployed software. In *SOSP 2009*, pages 87–102, 2009.

[43] B. Pfahringer, G. Holmes, and R. Kirkby. Optimizing the induction of alternating decision trees. In *PAKDD 2001*, pages 477–487, 2001.

[44] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *FSE 2008*, pages 2–12, 2008.

[45] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *ICSE 2014*, pages 254–265, 2014.

[46] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA 2015*, pages 24–36, 2015.

[47] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. T. Devanbu. On the "naturalness" of buggy code. In *ICSE 2016*, pages 428–439, 2016.

[48] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, 2002.

[49] E. K. Smith, E. T. Barr, C. L. Goues, and Y. Brun. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. In *ESEC/FSE 2015*, pages 532–543, 2015.

[50] Y. Song, Z. Zhuang, H. Li, Q. Zhao, J. Li, W.-C. Lee, and C. L. Giles. Real-time automatic tag recommendation. In *SIGIR 2008*, pages 515–522, 2008.

[51] S. H. Tan and A. Roychoudhury. relifix: Automated repair of software regressions. In *ICSE 2015*, pages 471–482, 2015.

[52] S. H. Tan, H. Yoshida, M. Prasad, and A. Roychoudhury. Anti-patterns in search-based program repair. In *FSE 2016*, 2016.

[53] S. H. Walker and D. B. Duncan. Estimation of the probability of an event as a function of several independent variables. *Biometrika*, pages 167–179, 1967.

[54] S. Watanabe, H. Kaiya, and K. Kaijiri. Adapting a fault prediction model to allow inter languagereuse. In *PROMISE 2008*, pages 19–24, 2008.

[55] W. Weimer. Patches as better bug reports. In *GPCE 2006*, pages 181–190, 2006.

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

[57] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE 2009*, pages 364–374, 2009.

[58] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, pages 183–200, 2002.