PatchID: An Overfitting Patches Recognition Method for Automated Program Repair

**abstract**：自动程序修复（APR）在生成补丁后，需要对补丁的正确性进行验证。通常情况下，APR使用测试套件作为补丁正确性验证的标准。然而，测试套件并不能完全代表程序的规约，这可能导致APR生成大量过拟合补丁，这些补丁不仅无法修复原有错误，还可能引入新的错误。为了帮助开发人员鉴别过拟合补丁，本文提出了一种名为PatchID的过拟合补丁识别方法。该方法的核心思想是通过比较在bug程序和正确补丁程序下的测试用例的程序动态行为来进行判断。如果一个测试用例在bug程序和正确补丁程序下的动态行为一致，那么它被认为是通过的；而如果一个测试用例在bug程序和正确补丁程序下的动态行为不同，那么它被认为是失败的。PatchID首先从bug程序和测试套件中构造导致程序出错的动态行为表达式。然后，它生成新的测试用例来增强原始的测试套件。最后，它从补丁程序中读取相同的动态行为表达式，并根据动态行为表达式的值是否随着补丁而改变来判断补丁是否过拟合。本文在包含155个Defects4J补丁和365个Java+JML补丁的两个数据集上进行了评估。PatchID成功鉴别出第一个数据集中的93个过拟合补丁和9个正确的补丁，以及第二个数据集中的225个补丁。此外，PatchID还将过拟合补丁进一步细分为三类补丁。实验证明，本文提出的方法优于现有的同类方法。

### 1 INTRODUCTION

自动程序修复（APR）在过去的十多年里引起了广泛的研究，涌现了许多修复技术。其中，基于测试套件的修复技术占据了主导地位，特别是自GenProg[1-3]提出以来。基于测试套件的修复工具使用给定的测试套件作为判断依据，如果生成的补丁能够通过测试套件，就被认为是正确的。然而，在实际应用中，测试套件的覆盖率有限，不能完全揭示程序的oracle，因此通过所有测试用例的补丁并不意味着它们完全正确，有时会出现通过所有测试用例但仍存在错误的补丁，即过拟合补丁。这导致了APR技术产生了大量无效的补丁。目前的修复技术还不够成熟，很大程度上只是简单地接受通过测试套件的补丁。然而，这种方法存在局限性，无法准确判断补丁的正确性。因此，需要进一步研究和改进修复技术，以提高补丁的质量和准确性。根据[4]的研究结果显示，GenProg、AE和RSRepair生成的大部分补丁是错误的。近期的研究关注于其他许多修复方法，例如使用人工编写的补丁、修复模板和条件合成、错误修复实例和禁止修改等方法进行修复。然而，这些新方法仍然存在修复性能较低的问题。

由于修复技术的性能不佳，开发人员不得不手动验证大量的过拟合补丁，这耗费了大量资源。因此，解决过拟合补丁的问题成为一个迫切需要研究和解决的问题。过拟合补丁检测方法通常采用以下方案：在APR工具生成补丁后，对这些补丁进行筛选，排除过拟合补丁，从而避免过拟合问题的出现。然而，当前的过拟合补丁检测方法存在准确率低和结果中提供的额外信息有限的问题。这导致开发人员仍旧要手动检测大量的过拟合补丁，因此在修复错误的速度上并没有显著提升。为了解决这些问题，需要进一步研究和改进过拟合补丁检测方法。

目前，能够识别出一个补丁是否过拟合已经是很成功的事了。因为快速识别过拟合补丁能够提高APR技术和开发人员修复错误的成功率，但是如果有技术能够对过拟合补丁进行分类，那么就能够为开发人员提供额外信息。根据文献[5]过拟合补丁可以分为如下三类：

* A-Overfitting Patch：补丁既没有完全修复不正确的行为也没有破坏原来正确的行为。
* B-Overfitting Patch：补丁修复了原来不正确的行为但是破坏了原来正确的行为，这被称为回归错误。
* AB-Overfitting Patch：补丁不但没有修复不正确的行为还破坏了原来正确的行为。

由于测试套件的不完备性，仅仅依靠程序的实际输出是否符合测试输出是远远不够的。当前的自动程序修复（APR）技术对测试套件的利用率相对较低，仅仅利用了测试套件的测试输入和测试输出，没有深入挖掘测试套件中隐藏的信息。因此，人们逐渐开始改变验证策略，并提出了各种过拟合检测方法。其中，Xiong等人[6]提出了一种挖掘测试套件和程序深层行为的技术，通过TEST-SIM和PATCH-SIM的思想，能够成功地排除一部分过拟合补丁，其成功率达到了56%。另外，Yang等人[7]的实验表明，补丁会改变程序的行为。他们试图利用程序的不变量来描述程序的行为，从另一个角度观察程序的预期行为。这些研究启发了我们，针对过拟合问题提出了一种新的技术——PatchID。

PatchID的目标是依靠给定的测试套件来挖掘程序的深层正确行为，以判断补丁的类别。所有的补丁都能够通过测试套件，通过的测试用例反映了程序的正确行为，而失败的测试用例则反映了程序的错误行为。基于这个观点，补丁应该保持程序正确行为的部分，并修改程序错误行为的部分。我们认为在一个正确的程序中，变量之间存在着特定的关系，这些关系通过它们运行时的值得以体现。这些关系是程序正确行为的一种表现形式。所以PatchID是基于以下两个重要的观察来判断一个补丁，它是从程序运行时变量的角度出发：

* PATCH-SIM：应用补丁后，通过的测试用例的程序的动态行为（变量之间的特定关系，本文用五元组来表示）与之前相同，而失败的测试用例的程序的动态行为不同。
* TEST-SIM：当两个测试用例具有相同的动态行为时，则这两个测试用例是属于同一类别的，即它们或者同属于通过的测试用例或者同属于失败的测试用例。

本文基于上述两点，设计并实现了PatchID算法，同时在155个补丁组成的数据集上评估了该方法。这些补丁都是APR技术基于Defects4J生成的，包括HDRepair[17]、jGenprog[8]、ACS[9]、jKail[10]、和Nopol[11]。本文方法成功地从155个补丁中识别出93个过拟合补丁，以及9个正确的补丁。此外，PatchID对93个过拟合补丁进一步将它们分为三类，这是目前其他技术所没有的功能。我们还在365个Java+JML补丁组成的数据集上进行验证，成功判断出225个补丁。

To sum up， the contributions of this paper are:

* 提出了一种用于计算程序执行相似性的程序状态抽象表示方法。
* 设计了一种用于自动补丁生成的过拟合补丁识别与细分算法。
* 实现了过拟合补丁识别的框架并且用实验评估了本文方法，结果表明本文方法是有效的。

本文的其余部分组织如下。第2节首先讨论了相关工作。第3节描述了本文方法涉及的一些必要概念。第4节详细介绍了本文方法。第5节对本文方法进行实验评估。第6节描述了有效性的威胁。最后，在第7节对全文进行总结。

### 2 RELATED WORK

**Automatic Program Repair.** 通常地，程序自动修复方法包括错误定位、补丁生成和补丁验证等三个步骤。它们首先会生成一些候选补丁，然后用现有的测试套件去验证这些候选补丁，通过测试套件的候选补丁将被视为正确补丁。目前已有方法可以大体上分为如下三类：

* 基于启发式搜索的APR：基于启发式搜索的自动修复技术通过人工定义启发式规则，指导修复补丁的生成过程。GenProg、ARJA-e[12-15]、PraPR[16]等属于这一类算法，它们用遗传算法将原程序作为搜索空间寻找补丁，此外还有HistoricalFix[17]、CapGen[18]、ConFix[19]等从历史修复补丁中寻找补丁的修复技术和SCRepair[20]、CRSearcher[21]、ssFix[22]、SimFix[23]、Refactory[24]等利用代码相似度的修复技术。
* 基于人工修复模板的APR：基于人工模板的补丁生成，根据开发者或研究人员的经验预定义一些补丁模板或者补丁生成策略，用于指导修复的过程。PAR[25]、iFixR[26]、SapFix[27]、ErrDoc[28]、BovInspector[29]、LeakFix[30]、AutoFix[31]、GumTree[32]、SketchFix[33]、NPEfix[34]、F1X[35]、HERCULES[36]以及Tan[37] 等人提出的anti-pattern方法等都属于这类算法。
* 基于语义约束的APR：基于语义约束的自动修复方法通过某种手段推断程序的正确规约，作为约束指导补丁的生成过程或对补丁的正确性进行验证。SemFix[38]、DirectFix[39]、Angelix[40]、AllRepair[41]、Nopol、S3[42]、SemGraft[43]、MemFix[44]、FootPatch[45]、SearchRepair[46]、SOSRepair[47]等属于这类算法。

这些修复技术都使用生成&验证的方法来产生补丁。对这些技术来说如果一个补丁能通过测试套件，那么它就是正确的补丁。但是一个程序是无法被完全测试的，现有数据集给出的测试套件都是弱测试套件[48]，所以修复技术在这些数据集上总是会产生大量的过拟合补丁。而这些过拟合补丁需要寻找新的技术来辨别。

**过拟合补丁检测.**

目前，相当多的过拟合补丁检测技术被提出并实现，而这些技术通常可以被分为以下两种。

1. Oracle补全方法

对这个分类进行解释

Xin等人提出DiffTGen[49]，尝试生成新的测试用例，对buggy程序与补丁进行语义差异检查，以此来解决APR的过拟合问题，并且再将生成的测试用例添加到测试集中增强原有的测试集。2018年，Mechtaev[50]等人针对过拟合问题提出了SemGraft。他们引入一个参考程序作为oracle，即功能与buggy程序相同但实现方式不同的参考程序。该方法将检测过拟合补丁的问题转化为语义等价检测问题，使得参考程序的语义符号摘要作为oracle，以此来判断过拟合补丁。

1. 补丁过滤方法

**补丁排序.** 通常情况下，APR可以对同一个bug生成多个不同的候选补丁。而该方法可以对这些候选补丁进行排序，排名靠前的补丁将被认为是正确的补丁，相反地，排名靠后的补丁被认为是过拟合补丁。Fan等人[51]学习人工修复的补丁获得补丁识别模型，利用该模型对候选补丁进行排序，从而排除排名较低的补丁。

**补丁识别.** 该方法直接给出一个候选补丁的类型，并不采用排名的方法对补丁进行过滤。Tan等人[52]从自动生成的APR补丁中提取了反模式(anti-pattern)，利用反模式来排除包含非法修复的补丁。Cashin等人[53]提出PATCHPART，他们试图找到buggy程序中的程序不变量以此来验证补丁是否违背了程序不变量这一准则来排除过拟合补丁。Nilizadeh[54]在小程序上用JML作为oracle来验证APR技术的效果，但是面对类似Defects4J这样的大型项目时，JML无法发挥作用。Xiong首次提出了PATCH-SIM、TEST-SIM，他们的方法不需要oracle，启发式地将新生成的测试用例(测试输入)进行分类，利用程序执行路径的相似度对补丁进行检测，该方法通过路径相似性检测过拟合补丁条件过于苛刻，而且它并没有对过拟合补丁进一步分类。Ghanbari等人[55]又提出了ObjSim, 该方法的原理是在使用补丁前后，比较补丁方法退出点的系统状态，也利用类似PATCH-SIM的思想来判断过拟合补丁。Ye[56]等人又提出了ODS，一种基于深度学习的过拟合识别工具，它可以对每一个补丁提取202个代码特征，然后对代码特征进行编码，最后通过基于决策树的集成学习模型来判断一个补丁。

### 3 PRELIMINARIES

为了识别出一个补丁是否过拟合，PatchID依赖以下的基本概念。

#### A. Program State Abstraction

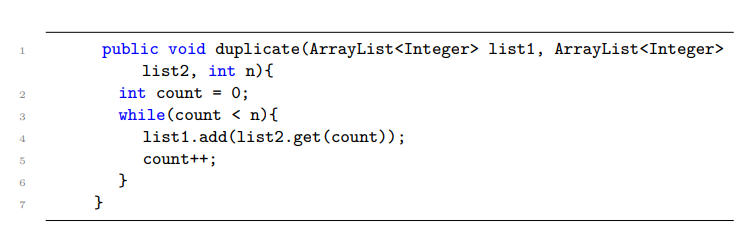
为了描述程序运行时的动态行为，本文引入程序状态抽象[57]的概念。程序状态抽象按照组成的层次依次为程序变量、基本表达式、扩展表达式以及布尔表达式。这些表达式统称为动态行为表达式。

**Variable.**在每一次测试用例执行期间，这些程序变量的值都会被记录到集合。这些程序变量来源于以下两种数据。

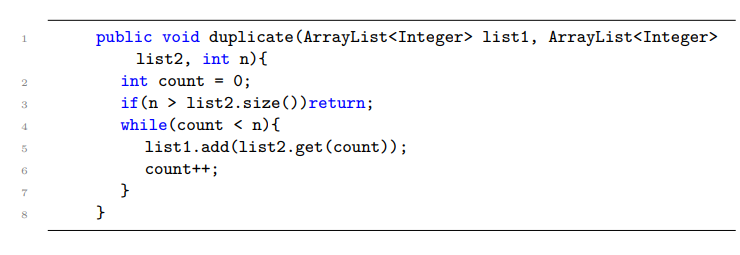
* 数字类型和布尔类型的精确值
* 引用类型表达式的对象标识符

**Expressions**.在一个程序中引用类型或数值类型是可以被监视的，这些类型构成了基本表达式，它们包括1）在Mbug (出现bug的方法)内部声明的局部变量（包括Mbug的参数），并且这些变量在处可见；2）Mbug所属的类的属性；3）在中任何可以计算的表达式，但是自增、自减、赋值和创建等带有副作用的表达式不能被监视。本文用表示在处（表示每一条语句的唯一标识）所有可监视基本表达式的集合。对于在中的每一个引用类型r，如下两种形式构成了扩展表达式，包括：1）r.f()，f()是一个无参函数而且返回的是在处一个可监视的类型。2）当r是this时，那么r.a类型的属性则在处可监视。本文用表示所有可监视类型的扩展表达式的集合。

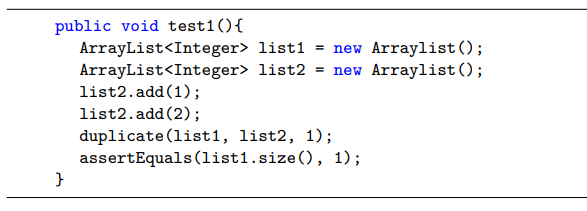
为了具体解释表达式的含义，下面结合Figure 1来加以说明。Figure 1给出了bug程序以及它对应的补丁。bug程序的功能如下：duplicate方法接收两个ArrayList（list1、list2）和一个int n。它的功能就是从list2中复制n个元素到list1。但是这个程序有一个明显的错误，它忽略了n大于list2长度的情况：当n大于list2的元素个数时，程序将抛出IndexOutOfBoundsException。而patch利用if语句避免了这个错误。在bug程序第一行中的count变量、参数中的list1、list2和n，它们都是可监视的基本表达式。而在这一处的扩展表达式就是list1.size()、list2.size()等。



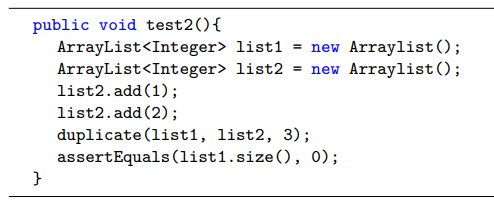
1. bug程序



(b) patch



(c)通过的测试用例test1



（d）失败的测试用例 test2

Figure 1：example

**Boolean expression**. 布尔表达式是由关系表达式和逻辑表达式组成的。本文用表示在处的一个布尔抽象集合。布尔表达式可以表示为如下四种形式：1）for each pair  of expressions of the same type， includes  and ; 2）for each pair  of expressions of integer type， includes ，for ；3）for each expression  of Boolean type， includes  and ；4）for each pair  of expressions of Boolean type， includes  and .

例如在Figure1(a)第一行中，n和list2.size()都是可监视的表达式，并且这两个表达式都是相同的Integer类型，那么可以组成以下六个布尔表达式而程序出现错误的原因就是布尔表达式。如果测试用例满足这个布尔表达式，程序进入while循环时，count的值最终会等于list2.size()，这会导致程序抛出越界错误。所以这导致了test1是通过的测试用例，test2是失败的测试用例，因为test2中list2的个数比3小。

#### B PATCH-SIM and TEST-SIM

如前所述，一个测试套件不像正式规范，它的编码规范是弱的和不完整的。低质量的测试套件是自动修复技术生成过拟合补丁的关键原因。为了充分利用现有的测试套件和增强测试套件来判断过拟合补丁，我们需要以下两个定义：

**TEST-SIM**. 当两个测试用例都有相同的程序行为时，则它们会有相同的测试结果， 即，在同一语句处的布尔表达式以及对应的值相同，那么这两个测试用例应该都是通过的测试用例或者都是失败的测试用例。自动修复技术通常以通过所有测试用例的补丁作为正确的补丁，但是测试套件是无法表达出完整的oracle。为了增强测试套件，我们需要生成新的测试用例，但是PatchID的生成新测试用例的方法并不需要也无法知道测试输出是什么，只需要关注它在语句中的布尔表达式。例如有一个新的测试用例使得list1.size = 0， list2.size = 5， n = 10。则在Figure 1中的bug程序第三行语句里该测试用例的布尔表达式有，与test2在这一处的布尔表达式相同，那么这个测试用例被认为是失败的测试用例。

**PATCH-SIM**. 使用正确补丁后，通过的测试用例与之前的布尔表达式及其取值相同，而失败的测试应该不同。比如有一个bug程序，所有通过的测试用例使一个Boolean abstraction 的值为false，所有失败的测试用例使的值为true。那么判断补丁是否过拟合就不仅仅是观察程序的输出这种单一的方式，而是可以通过比较使用补丁前后在某条语句中的值。通过的测试用例测试补丁时，的值应该和bug程序一致；失败的测试用例测试补丁时，的值应该和bug程序不同。根据PATCH-SIM，我们可以得到前面的例子中，patch是正确的，因为在第四行（对应bug程序的第三行），已经不存在了，所有使得这个布尔表达式成立的测试用例都会转到return语句。

通过这两个观察，我们可以从bug程序和测试套件中挖掘出更多的信息，从而对补丁的判断不需要oracle。利用TEST-SIM对新测试用例启发式地分类，利用PATCH-SIM对补丁进行启发式地分类。

### 4 APPROACH

本部分主要介绍了PatchID的详细流程，包括它的Overview和三个主要的模块，即Snapshot finder、Test generation、Detection。

### 4.1 Overview

下图2给出了本文方法的整体流程。

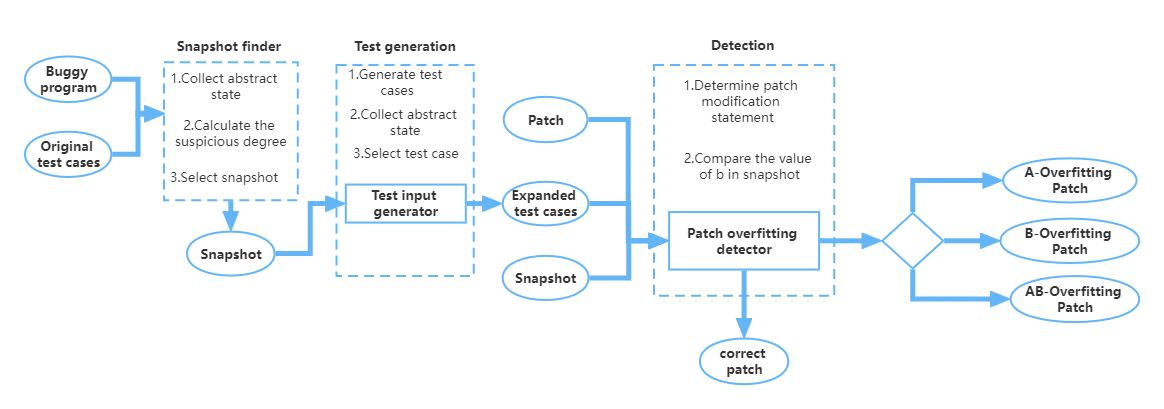


Figure 2：Approach overview

该方法输入一个bug程序和一个原始测试集以及对应的补丁patch。首先通过Snapshot finder找到可疑度最大的snapshot s，再通过s和Test input generator生成新测试用例。然后将新的测试用例添加到测试集里，生成扩展后的测试集。PatchID运行补丁程序patch和扩展的测试集，并保存s中表达式的值。最后通过Patch overfitting detector对比使用补丁前后每个测试用例的表达式值是否变化来判断一个补丁是否过拟合，并对检测为过拟合的补丁进行分类。

### 4.2 Snapshot Finder

Snapshot finder就是为了找到可疑度最高的snapshot 。这一步有两个作用：一是为下一步生成新的测试用例提供标准，从而对新的测试用例进行分类；二是为第三步补丁程序运行测试集后得到的snapshot集合进行对比，从而判断补丁的类型。

**snapshot.** snapshot 。s作为一个五元组，其中是每一条语句的唯一标识，是一个布尔表达式，是的值（true or false）。表示测试套件中每一个测试用例的唯一序号，表示测试用例在Mbug执行过程中的实际值。每一个五元组snapshot都有一个对应的可疑度。Snapshot的可疑度用于衡量bug的可能性。可疑度最大的snapshot最有可能导致程序出现bug。可疑度由以下两种因素决定：1）a syntactic analysis of expression dependence ; 2）a dynamic analysis . 其中随着在前后语句出现的次数增加而增加；在失败的测试用例中取值为的次数越多，在通过的测试用例中取值为的次数越少，那么的值越大。本文采用的可疑度计算公式[57]如下：

因为可疑度计算公式的缺陷，与测试用例实际执行的结果不是完全一致的，所以我们必须注意是用来对之后新测试用例分类的，是用来对补丁进行分类的。Snapshot作为本算法的核心，本文设计如下算法来构建Snapshot

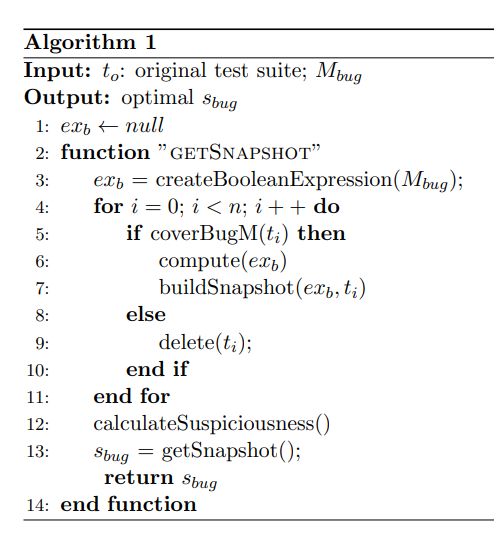


Figure 3 ：Snapshot构建算法

在该算法中，第一步PatchID在Mbug中创建每条语句的Boolean abstraction集合。第二步PatchID使用运行bug程序，如果测试用例覆盖Mbug，那么记录Mbug运行的程序抽象状态，并计算集合中每个Boolean abstraction的值（真实值），生成的snapshot；如果未覆盖，则删除这个测试用例。第三步计算每一个snapshot的可疑度，此时将得到的值（PatchID将通过这个值来生成测试用例）。第四步选择可疑度最高的snapshot :如果有多个snapshot，那么随机选择它们中的一个。为了后续的流程，PatchID会保存一个snapshot集合，该集合的与相同，保存的是每个测试用例对应的真实值。在前面的例子中，表示在bug程序第三行的布尔表达式，由于该表达式的值为true，才导致程序出现错误。

### 4.3 Test Generation

Test generation的目标是利用TEST-SIM准则生成新的测试用例。PatchID不需要去关注这些测试用例的输出是否正确，但是需要这些测试用例生成的snapshot中布尔表达式的值与的相同。也就是说PatchID需要一些新的失败的测试用例，这些测试用例是用来增强测试套件的。

PatchID使用现有的自动测试用例生成工具Evosuite[58]生成一组测试用例。由于Evosuite生成的测试用例是针对一个类的，所以要对这组测试用例进行筛选。类似于Snapshot finder中的步骤，选择覆盖Mbug的测试用例并且保存程序抽象状态和snapshot。如果新测试用例的snapshot()与可疑度最高的()分别对应相同，那么就说明找到了新的失败的测试用例，将它们添加到测试用例集。PatchID不选择正确的新测试用例，因为默认Mbug对于原测试集中通过的测试用例来说，不仅仅输出结果是正确的，程序行为也是正确的。

### 4.4 Identification

这一步的目的是为了判断一个补丁是否是过拟合。它需要之前保存的snapshot集合、扩展测试套件、patch。PatchID将用扩展测试套件来运行patch，保存snapshot集合，然后与对比，最终判断补丁的类型。

#### 4.4.1 Select Statement

对于bug程序来说，补丁一般包含insert、delete、replace和update。那么bug程序的snapshot中的不能直接在patch去监视，因为语句的位置变了，需要重新选择patch中的语句去监听相同的布尔表达式。我们认为无论哪一种修复操作，程序只有在修复操作结束后，才可能有正确的程序行为，所以本文方法选取的位置在修改结束的下一条语句。但是对于一些特殊的情况，就需要用其他的规则去选择。

我们将bug程序和patch第一次不同的语句记为，最后不同的语句记为，规则定义如下：

* 如果是for、while、if等块语句，而在内部，那么在代码块结束的下一条语句。
* 如果与是顺序执行关系，那么在的下一条语句
* 如果是程序或代码块的最后一条语句，则。

#### 4.4.2 Patch identification and Classification

前一步中确定了，PatchID就会使用测试套件运行patch，获取snapshot集合，获取的步骤与4.2相似。接下来的工作就是将两个snapshot集合进行比较判断patch是否过拟合。值的注意的是，过拟合补丁分类也是同时进行的，因为当候选补丁被识别为过拟合补丁时，那么这个补丁将被进一步分类。

为了识别出过拟合补丁，需要两个变量、。对于失败的测试用例，表示两个集合间相同的数量。对于通过的测试用例，表示两个集合间不同的数量。根据变量和取值的不同，补丁的类型定义如下：

我们将两个snapshot集合按照测试用例的唯一标识将它们一一对应，比较每一个测试用例使用补丁前后的值。根据测试用例的类型和来源将分为三类：passing、failing、new。如果，当不同时，；如果，当相同时，。两个集合比较结束后，可以得到和的值。根据上述公式以及得到的和就能判断补丁的类型。

for example，tese1和test2在bug程序中对应的分别是false、true。而patch中的值为false、false。那么得到，则判断patch是正确的。

### 5 EVALUATION

为了实验评估PatchID方法的有效性，我们提出了以下五个研究问题：

* RQ1:PatchID能否识别自动修复工具生成的过拟合补丁？与其他方法是否有优势？
* RQ2:PatchID判断补丁的效率如何？
* RQ3:造成过拟合补丁识别错误的原因是什么？
* RQ4:PatchID对过拟合补丁分类的可靠性如何？

#### 5.1 Dataset

本文分别在两个数据集上进行了实验验证，其中，第一个数据集是文献[8]中收集的数据集，该数据集由6个APR工具在Defects4J上生成的补丁组成的。需要注意的是，为了进行对比实验，我们使用的Defects4j数据集是Xiong文中的数据集一致，并没有运行APR工具收集补丁。该数据集中已经给出了补丁识别结果，这些结果由APR作者给出。第二个数据集Java+JML dataset则是由Nilizadeh等人创建的，其中补丁的结果由原作者给出。我们将PatchID给出的结果与数据集提供的结果进行对比，并以此来评估PatchID的效果。

**Defects4J.** 目前，Just[59]提出的Defects4J是自动程序修复领域使用最广泛的Java程序数据集。Defects4J到目前为止已经有17个项目了，其中包含了835个缺陷。该数据集中的每个程序缺陷包含至少一个可以触发它的测试用例。本文使用该数集中最常被使用的6个项目，即：Chart、Time、Math、Lang、Closure和Mockito，其中，Chart是专门显示图标的项目；Time是用于日期、时间处理的项目；Math是科学计算的项目；Lang是一组用于操作JDK类的额外方法；Closure是Javascript的优化编译器；Mockito是一个用于单元测试的模拟框架。

根据Xiong给出的数据集信息，候选补丁由6个APR工具产生。这6个APR工具分别为jGenProg、Nopol 2015、Nopol 2017、ACS、HDRepair和jKali，其中，jGenProg是GenProg的java版本，它是基于遗传算法的启发式搜索修复工具；Nopol是一个针对Java程序中条件语句错误的修复技术，该技术针对错误语句类型而给出了不同的修复策略：如果定位出错的代码位置是条件语句，则 Nopol 通常生成的修复补丁为修改原始的条件语句;如果定位出错的代码位置是非条件语句，则通过添加一个新的条件跳过当前语句的执行实现修复。本次数据集包括Nopol 2015[67]和Nopol 2017[68]两个版本；ACS是一个高精度的条件语句综合工具，它基于统计分析，提取补丁模板用于修复；HDRepair也是基于统计分析的修复工具；jKali是Kali在Java上的重新实现，它是一个只有删除功能的修复工具。

数据集总共有220个补丁，本文针对这220个补丁进行实验判断其是否为过拟合补丁，一共得到了165个补丁是否为过拟合补丁的运行结果，其余补丁由于超过设定执行时间界限或者无法运行，未能给出最终结果。在这165个补丁中，本文方法除了9个补丁外（因为在原数据集中并未给出9个补丁的分类），剩余的155个补丁均给出了是否为过拟合补丁的判定结果。Table 1给出了155个补丁的具体信息。

Table 1: Defects4j Dataset

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Project  Tools | Chart | Closure | Lang | Math | Mockito | Time | Total |
| Nopol | 12 | 39 | 101 | 25 | 1 | 8 | 95 |
| jKai | 5 | 0 | 0 | 8 | 0 | 1 | 14 |
| jGenprog | 6 | 0 | 0 | 12 | 0 | 2 | 20 |
| HDRepair | 0 | 0 | 2 | 6 | 0 | 1 | 9 |
| ACS | 1 | 0 | 4 | 11 | 0 | 1 | 17 |
| Total | 24 | 39 | 16 | 62 | 1 | 13 | 155 |

**Java+JML dataset.** 由Nilizadeh提出的这个数据集是第一个经过验证的、公开可用的Java程序数据集。它由以下四个部分组成：正确的程序、经过变异的错误程序、测试套件、基于APR的补丁。该数据集的程序具有JML规范，用于实验评估。这个数据集实现了各种经典的算法与数据结构，比如冒泡排序、阶乘、队列等。它们都是用JML编写的正式规范的小型程序，因此，可以被认为是具有oracle的程序。测试套件是使用基于AFL的模糊工具来创建的，测试套件根据生成的测试用例数量规模分为Small和Medium两种。错误程序是通过PITest——一个Java程序突变工具，向每个java程序注入单个错误来创建的。PITest通过更改控制条件、更改赋值表达式、删除方法调用和更改返回值来生成错误。基于APR的修复补丁采用以下的修复工具得到，分别是ARJA-E、Cardumen、jGenProg、jKali、jMutRepair、Kali-A, and Nopol。

我们从Java+JML数据集中选取了基于Medium测试套件的过拟合补丁236个，基于Small测试套件的过拟合补丁336个，这些过拟合补丁都是通过JML specification判断出来的，并且确定这些补丁是过拟合补丁。PatchID算法在总共572个补丁上运行，得到了365个补丁的运行结果，具体结果如表2所示。

Table 2: Java+JML Dataset

| PatchType | Collected | Validated |
| --- | --- | --- |
| Medium | 236 | 144 |
| Small | 336 | 221 |
| Total | 572 | 365 |

#### 5.2 Experiment Setup

我们在JAID框架基础上实现了PatchID。我们运行了两个数据集，其中Defects数据集的运行环境是Ubuntu 18.04、Defects4j 2.0.0、JDK1.8；Java+JML 数据集的运行环境是Windows 10、JDK1.8。

**RQ1.** 为了评估PatchID的效果，我们运行了收集到的补丁集，并且将补丁的判断结果和重要变量的值保存在一个文件中。这些变量分别为监听的语句、snapshot的表达式和值、值不相同的passing test、值相同的failing test、new test。此外，我们与两个过拟合补识别工具进行了对比，分别是PatchSim和ODS。PatchSim是Xiong提出的方法，我们采用和他相同的准则，即PATCH-SIM和TEST-SIM，但是他用程序执行路径的角度来观察，而PatchID采用了程序动态表达式。为了探究基于表达式的PATCH-SIM相对于程序路径执行是否有更好的效果，所以我们选择Xiong的工具。ODS是Ye[56]提出的基于深度学习的过拟合识别工具，我们根据作者给出的使用手册在第一个数据集上收集了ODS的运行结果。我们得到了143个结果，其中有9个补丁是unknown，95个补丁是Ye的实验中给出结果了，我们对Ye的实验结果与143个结果取了并集操作，最终得到了163个补丁识别的结果。我们选择ODS来进行比较是因为ODS是最近的过拟合补丁识别工具，并且它的效果十分出色。我们选择在第一个数据集上进行比较，因为Defects4j更接近真实项目。此外，我们选择206个补丁作为基数（220个补丁里有14个未知类型），因为三个工具支持的补丁交集数量太少。

**RQ2.** 记录每个补丁的运行时间，时间以整分为单位进行计算，多余的秒数按照四舍五入近似。

**RQ3.** 手动分析分类错误的补丁，并分析失败的原因。

**RQ4.** 为了验证PatchID对过拟合补丁分类方法的有效性，我们人工判断了过拟合补丁的具体分类。由于Defects4J的补丁过于复杂，本文只在Java+JML dataset进行了分析。人为判断的依据是从这个数据集的中bug程序的警告文件”esc.txt”和补丁的警告文件” ESC\_Repaired.txt”产生的不同警告。这两个警告文件是该数据集提供的，它们产生的原因是当Java程序运行时的状态不符合JML的预期时，那么就会产生一个警告文件，警告文件记录程序中的哪些行不符合JML的预期，并且记录对应的JML语句。我们对比了每一对警告文件，当ESC\_Repaired的警告数量比esc的少时，我们认为该补丁未完全修复bug，即A-overfitting patch；当ESC\_Repaired出现了新的警告时，我们认为该补丁引入了回归问题，即B-overfitting patch；当上述两种错误都存在时，我们认为该补丁是AB-overfitting patch 。

#### 5.3 Experimental Result

#### 5.3.1 Results of RQ1

**Performance on Defects4J.** Table 3和Table 4分别显示了本文方法在APR工具和不同项目上的运行结果。如表所示，PatchID成功地从155个补丁中过滤出102个补丁，其中过拟合补丁93个，正确的补丁9个。

Table 3: Result By APR Tools

| Tool | All-#C | All-#O | #C | #O | A | B | AB |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Nopol2015 | 4 | 20 | 0 | 10(50%) | 9 | 0 | 1 |
| Nopol2017 | 3 | 68 | 1(33.33%) | 52(76.47%) | 44 | 3 | 5 |
| HDRepair | 4 | 5 | 3(75%) | 1(20%) | 1 | 0 | 0 |
| ACS | 11 | 6 | 4(36.36%) | 6(100%) | 4 | 1 | 1 |
| jKali | 0 | 14 | 0 | 11(78.57%) | 9 | 0 | 2 |
| jGenprog | 5 | 15 | 1(20%) | 13(50%) | 10 | 0 | 3 |
| Total | 27 | 128 | 9(33.33%) | 93(72.65%) | 77 | 4 | 12 |

"All-#C/All-#O"表示所有Correct/Overfitting补丁数量， "#C/#O "表示识别出Correct/Overfitting 补丁数量

A = A-Overfitting Patch，B = B-Overfitting Patch，AB = AB-Overfitting Patch

Table 4: Result By Project

| Project | All-#C | All-#O | #C | #O | A | B | AB |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Lang | 6 | 10 | 2(33.33%) | 8(80%) | 8 | 0 | 0 |
| Math | 13 | 49 | 5(38.46%) | 34(69.39%) | 25 | 2 | 7 |
| Chart | 3 | 21 | 1(33.33%) | 12(57.14%) | 11 | 0 | 1 |
| Time | 2 | 11 | 1(50%) | 10(90.91%) | 10 | 0 | 0 |
| Closure | 2 | 37 | 0 | 29(78.38%) | 23 | 2 | 4 |
| Mockito | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Total | 27 | 128 | 9(33.33%) | 93(72.65%) | 77 | 4 | 12 |

**Overfitting patch.**  从表3中我们可以发现PatchID在Nopol2015、Nopol2017、jKali、jGenprog、ACS这五个修复工具上的效果比较好(均能识别50%以上的过拟合补丁)，但是在HDRepair效果比较差(只能识别20%的过拟合补丁)。

从表4来看，过拟合补丁识别的效果更加明显了。PatchID在Time中表现最好，识别了90.91%的过拟合补丁。在Chart中表现最差，只有57.14%。

**Correct patch.** 在155个补丁中，correct patch有27个，PatchID能够识别其中的9个补丁。其中Math中correct patch的数量最多，patchID共识别出5个。

**Performance on Java+JML dataset.**从表5的数据中可以看出在基于Medium类型的补丁，PatchID能够正确判断93个过拟合补丁，识别了64.58%的过拟合补丁；基于Small类型的补丁，PatchID能够判断132个过拟合补丁，识别了59.73%的过拟合补丁。

从Medium和Small的准确率可以明显看出，随着测试套件中测试用例个数减少，准确率也会下降。这个数据说明了弱测试套件会影响PatchID的判断。

Table 5: Result By PatchType

| PatchType | All-#O | #O | A | B | AB |
| --- | --- | --- | --- | --- | --- |
| Medium | 144 | 93（64.58%） | 90 | 0 | 3 |
| Small | 221 | 132（59.73%） | 120 | 1 | 11 |
| Total | 365 | 225(61.64%) | 210 | 1 | 14 |

**Comparison with other tools.** 我们与其他过拟合补丁识别技术进行了对比，实验结果表明PatchID的整体效果要优于其他技术。表6所示，206个补丁中有170个overfitting patch，36个correct patch。其中PatchID识别过拟合补丁的数量最多，共93个。而ODS识别了72个，PatchSim识别了62个。在识别正确补丁的数量来看，ODS能识别的补丁最多，有26个，PatchID识别了9个，PatchSim未识别任何一个正确补丁。所以总体上来说，PatchID能够正确识别的补丁最多，共102个，而ODS识别98个，PatchSim只有62个。

Table 6: Comparison with PatchSim and ODS

| Dataset | | | PatchSim | | ODS | | PatchID | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Project | #O | #C | #O | #C | #O | #C | #O | #C |
| Chart | 23 | 4 | 13 | 0 | 14 | 3 | 12 | 0 |
| Closure | 54 | 2 | 0 | 0 | 0 | 0 | 29 | 0 |
| Lang | 11 | 6 | 5 | 0 | 10 | 6 | 8 | 2 |
| Math | 68 | 21 | 35 | 0 | 37 | 16 | 34 | 5 |
| Mockito | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| Time | 13 | 2 | 9 | 0 | 11 | 1 | 10 | 1 |
| Total | 170 | 36 | 62 | 0 | 72 | 26 | 93 | 9 |

**Answer to RQ1.** 从实验结果分析，PatchID识别过拟合补丁的效果较好，在两个数据集上分别识别了72.65%和61.64%的过拟合补丁。此外，我们在基于Defects4j的补丁集上与PatchSim、ODS进行对比。结果表明，PatchID在识别过拟合补丁的表现较前两者好。对于PatchSim工具来说，PatchID也同样使用PATCH-SIM和TEST-SIM。但是PatchID识别过拟合补丁和正确补丁的数量上均多于前者，这表明基于程序动态行为表达式的方法比程序执行路径的方法能更好地应用PATCH-SIM和TEST-SIM。值得注意的是，ODS与PatchID都有各自的优势。ODS能够识别更多的正确补丁，PatchID能够识别更多的过拟合补丁。但是观察二者识别的总数时，PatchID占优，比ODS多了4个。总的来说，PatchID能够有效识别过拟合补丁，并且在相同的条件下，能够正确识别过拟合补丁的数量超过了同类型的其他技术。

#### 5.3.2 Results of RQ2

PatchID的目标是帮助开发人员更加快速判断过拟合补丁，节省花费的时间，所以本文关注PatchID判断一个补丁所花费的时间。我们从PatchID的日志上记录了每一个补丁的开始时间与结束时间，最终得到了补丁的运行时间。

**Performance on Defects4J.** 我们记录了155个补丁的运行时间。就如前面所讲，PatchID分三个步骤来判断一个补丁，而根据项目的不同，每个步骤花费的时间占比也不同。如表7所示，大部分(57.42%)的补丁在5分钟内得到结果。在这些补丁运算的过程中，第二步骤，也就是Test Generation，占了大部分时间，而剩余时间都是运行测试用例和计算程序的布尔表达式所花费的时间。运行时间超过五分钟的补丁则需要更多时间来获取程序抽象状态，因为据我们的观察，Evosuite通常在Defects4J上花费12分钟的时间生成新的测试用例。在本文实验过程中，运行时间超过60min的有三个补丁，其中一个花费123min，绝大部分时间用来运行测试用例。判断一个补丁是否为过拟合补丁的平均时间为11.6min，其中在可接受范围t < 10min内的补丁占76.77%，这是令人满意的。

Table 7: Run Time

| Project | t <= 5 | t <= 10 | t <= 30 | t > 30 | Total |
| --- | --- | --- | --- | --- | --- |
| Chart | 22 | 2 | 0 | 0 | 24 |
| Lang | 12 | 2 | 1 | 1 | 16 |
| Math | 35 | 12 | 6 | 10 | 62 |
| Time | 9 | 0 | 3 | 1 | 13 |
| Mockito | 1 | 0 | 0 | 0 | 1 |
| Closure | 10 | 14 | 10 | 5 | 39 |
| Total | 89(57.42%) | 30(19.35%) | 20(12.90%) | 17(10.97%) | 155 |

**Performance on Java+JML dataset.**我们记录了365个补丁的消耗时间的分布情况。从表8中可以看见365个补丁在十分钟内就能得出结果，超过十分钟的补丁只有2个。

Table 8: Run Time

|  |  |  |
| --- | --- | --- |
| PatchType | t <= 10 | t > 10 |
| Small | 220 | 1 |
| Medium | 143 | 1 |
| Total | 363 | 2 |

**Answer to RQ2.** 从两个数据集的结果分析可以得知，PatchID判断一个补丁的时间大部分都低于10分钟。这是一个令人满意的结果，特别是针对第一个数据集，因为Defects4j是一个大型项目。当程序的规模变得更大时，超过十分钟判断一个补丁的情况将变得频繁，但是PatchID仍然能够保证快速判断大多数补丁，甚至其中有不少补丁所花费的时间不超过五分钟。所以PatchID的效率能够提高开发人员过滤过拟合补丁的速度，减少他们花费的时间。

#### 5.3.3 Results of RQ3

我们分析了所有被PatchID识别错误的补丁和对应的日志文件，查看了文件中记录的snapshot，特别是布尔表达式，认为以下两点是产生错误结果的原因。

* Weak test suite
* Unsatisfying snapshot find strategy

这两个原因共同导致了PatchID判断错误。一方面，绝大部分的测试套件都只有12个失败的测试用例，并且覆盖Mbug的测试用例过少，导致PatchID实际能收集到正确的布尔表达式可能性较低。另一方面，虽然PatchID能生成大量的snapshot，但是由于第一个原因，会导致多个snapshot的可疑度相同。产生可疑度相同的原因是可疑度公式的缺陷。此外，由于每个补丁判断过程中只能选一个snapshot作为标准，PatchID只是随机选择其中一个，有相当大的概率会选择无关的snapshot，这也会导致最终判断结果的失败。

此外，由于我们在Test generation这个步骤中只选择了新的失败测试用例，这导致了我们的实验更注重补丁是否修复了原来的错误，而忽视了补丁是否引入了新的错误。

#### 5.3.4 Results of RQ4

为了讨论PatchID在过拟合补丁分类的优点与缺点，本文对AB-overfitting patch分为两类补丁：AB-1、AB-2。其中AB-1补丁代表回归错误发生在错误的方法Mbug外的方法；AB-2表示回归错误仍然出现在Mbug。例如，当方法F调用Mbug，patch导致F出现了错误，那么该补丁被认为AB-1；如果回归错误只出现在Mbug，那么该补丁被认为AB-2。此外，A patch 是A-overfitting patch和AB-overfitting patch 的统称。Table12给出了PatchID在Java+JML dataset上对过拟合补丁分类的效果。我们发现A-Overfitting patch和AB-Overfitting patch的数量相近，这说明在该数据集上，APR补丁实际上会引入回归错误。此外，AB-1的数量远超过AB-2，这还说明回归错误一般会发生在Mbug以外的方法中。从PatchID对过拟合补丁分类的结果来看，在Small和Medium数据集上的正确分类补丁数量分别为54、35，并且这些补丁大多数是A-Overfitting patch，只有一个AB-Overfitting patch。

从表12可以看出，对于大部分AB-Overfitting patch，PatchID都不能正确分类，这是由于PatchID的局限性导致的，因为PatchID只会监听唯一一个方法Mbug的程序抽象状态，而大部分回归错误都发生在Mbug之外的方法，这也是AB-1 patch数量如此之多的原因。

但是，从Table 13来看，PatchID对未修复原来错误的补丁的判断却相当精确，所有未修复原来错误的补丁(包括A、AB-1、AB-2)共有224个，其中Small有131个，Medium有93个，PatchID能全部识别出这些没有修复原有错误的补丁，成功率达到100%。

Table 12 ：Classification Of Overfitting Patches(JML)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Overfitting  PatchType | A | B | AB-1 | AB-2 | Total |
| Small | 55(49) | 1(0) | 70(4) | 6(1) | 132(54) |
| Medium | 34(33) | 0 | 55(2) | 4(0) | 93(35) |

表12中给出了在Java+JM上实际的补丁数量和PatchID分类的各类别补丁数量，括号中的数字代表PatchID正确分类的补丁数量。

Table 13：Classification Of A Patches

|  |  |
| --- | --- |
| Overfitting  PatchType | A |
| Small | 131(131) |
| Medium | 93(93) |

A Patch 指的是所有未修复原有bug的补丁，包括 A-Overfitting patch、AB-Overfitting patch，括号中的数字代表PatchID正确分类的补丁数量

Table 14 ：Classification Of Overfitting Patches(Defects4j)

| Project | All-#O | A | B | AB |
| --- | --- | --- | --- | --- |
| Lang | 10 | 8 | 0 | 0 |
| Math | 49 | 25 | 2 | 7 |
| Chart | 21 | 11 | 0 | 1 |
| Time | 11 | 10 | 0 | 0 |
| Closure | 37 | 23 | 2 | 4 |
| Mockito | 0 | 0 | 0 | 0 |
| Total | 128 | 77 | 4 | 12 |

表14给出了在Defects4j上 93个过拟合补丁的分类结果。PatchID将它们分成三个类别，其中A-Overfitting Patch占比最多，达到77个；其次是B-Overfitting Patch有4个；AB-Overfitting Patch数量为12个。由于该数据集的程序十分复杂，我们无法手动判断实验结果是否正确。但是从PatchID在Java+JML数据集的表现来看，

**Answer to RQ4.** 从上述的实验来看，PatchID对于识别A patch十分准确，准确率达到了100%。但是对于回归错误，PatchID没有识别出来任何一个补丁，其中大部分为AB-1补丁。所以PatchID能够准确分类出A-overfitting patch，但是无法准确分类B-overfitting patch 和AB-overfitting patch。

根据我们的分析，其中的原因在于PatchID只监听Mbug的程序抽象状态，无法监听其他方法。而大部分AB-overfitting patch都是AB-1 patch，换而言之，回归错误发生在Mbug之外，所以PatchID分类出错。而对于A patch，由于原有的错误都存在于Mbug，所以PatchID能够正确分类。而B-overfitting patch在该数据集中只出现了一个，我们无法评估PatchID对该类过拟合补丁的效果。

### 6 THREATS TO VALIDITY

**Threats to Internal Validity.** 我们从两个数据集中丢弃了一些补丁。有些补丁无法被PatchID执行，另一些补丁运行时间超出了我们的预期。因此，会导致评估结果出现偏差。特别是第二个原因，我们运行这些补丁的时间超过了24h，仍然没有得到结果。PatchID不仅仅识别过拟合补丁，还需要帮助开发人员快速排除过拟合补丁来提高修复bug的速度，所以我们只能将超出预期时间的补丁丢弃。

此外，我们在Java+JML dataset数据集中手动判断了过拟合补丁的类型，这可能导致实际的三种过拟合补丁数量与我们判断的数量不一致，因此影响PatchID在过拟合补丁分类上的效果。

**Threats to External Validity.** 我们目前只在两个数据集上评估了PatchID，而PatchID可能在其他数据集上有不同的表现。但是当前的数据集并没有给出过拟合补丁的分类，所以我们不知道PatchID在其他数据集上的过拟合补丁分类方面的效果。

**Threats to Construct Validity.** 我们从动态程序表达式的角度解释了PATCH-SIM，而PATCH-SIM可以从更多的角度评估。从其他角度进行观察可能会有更好的结果。

### 7 CONCLUSION

本文提出了将程序表达式和程序相似行为两者结合的过拟合识别方法，并将过拟合补丁进一步分为三类。我们在JAID框架上实现了本文提出的方法PatchID。正如我们在实验评估中看到的，PatchID方法能够从157个Defects4J数据集的补丁中有效地过滤出78个补丁，380个Java+JML dataset数据集的补丁中过滤出169个补丁。实验结果表明，我们应该深挖测试套件给出的信息，而不再只局限于测试输入和测试输出。未来的工作将着眼于：(1) 强化程序监听的范围，进一步提高过拟合补丁分类的成功率，(2)对识别出的过拟合补丁提供有效的修复方案，使其更接近正确的补丁。

1. Forrest S， Nguyen TV， Weimer W， et al. A genetic programming approach to automated software repair[C]//Proceedings of the 11th Annual conference on Genetic and evolutionary computation. 2009: 947-954.
2. Le Goues C， Dewey-Vogt M， Forrest S， et al. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each[C]//2012 34th International Conference on Software Engineering (ICSE). IEEE， 2012: 3-13.
3. Le Goues C， Nguyen TV， Forrest S， et al. Genprog: A generic method for automatic software repair[J]. Ieee transactions on software engineering， 2011， 38(1): 54-72.
4. Xin Q， Reiss S P. Identifying test-suite-overfittingted patches through test case generation[C]//Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis. 2017: 226-236.
5. Yu Z， Martinez M， Danglot B， et al. Alleviating patch Overfitting with automatic test generation: a study of feasibility and effectiveness for the Nopol repair system[J]. Empirical Software Engineering， 2019， 24(1): 33 -67
6. Xiong Y， Liu X， Zeng M， et al. Identifying patch correctness in test-based program repair[C]//Proceedings of the 40th international conference on software engineering. 2018: 789-799
7. Yang B， Yang J. Exploring the differences between plausible and correct patches at fine-grained level[C]//2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF). IEEE， 2020: 1-8
8. Chen L， Pei Y， Furia C A. Contract-based program repair without the contracts: An extended study[J]. IEEE Transactions on Software Engineering， 2020.
9. Martinez M， Monperrus M. Astor: A program repair library for java[C]//Proceedings of the 25th International Symposium on Software Testing and Analysis. 2016: 441-444.
10. Xiong Y， Wang J， Yan R， et al. Precise condition synthesis for program repair[C]//2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE， 2017: 416-426
11. Rothenberg BC， Grumberg O. Sound and complete mutation-based program repair[C]//International Symposium on Formal Methods. Springer， Cham， 2016: 593-611
12. Yuan Y， Banzhaf W. Arja: Automated repair of java programs via multi-objective genetic programming[J]. IEEE Transactions on software engineering， 2018， 46(10): 1040-1067
13. Qi Y， Mao X， Lei Y， et al. The strength of random search on automated program repair[C]//Proceedings of the 36th International Conference on Software Engineering. 2014: 254-265
14. Yuan Y， Banzhaf W. A hybrid evolutionary system for automatic software repair[C]//Proceedings of the Genetic and Evolutionary Computation Conference. 2019: 1417-1425
15. Yuan Y， Banzhaf W. Toward better evolutionary program repair: An integrated approach[J]. ACM Transactions on Software Engineering and Methodology (TOSEM)， 2020， 29(1): 1-53
16. Kim D， Nam J， Song J， et al. Automatic patch generation learned from human-written patches[C]//2013 35th International Conference on Software Engineering (ICSE). IEEE， 2013: 802-811
17. Le XBD， Lo D， Le Goues C. History driven program repair[C]//2016 IEEE 23rd international conference on software analysis， evolution， and reengineering (SANER). IEEE， 2016， 1: 213-224
18. Wen M， Chen J， Wu R， et al. Context-aware patch generation for better automated program repair[C]//2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE， 2018: 1-11
19. Kim J， Kim S. Automatic patch generation with context-based change application[J]. Empirical Software Engineering， 2019， 24(6): 4071-4106
20. Ji T， Chen L， Mao X， et al. Automated program repair by using similar code containing fix ingredients[C]//2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC). IEEE， 2016， 1: 197- 202
21. Wang Y， Chen Y， Shen B， et al. CRSearcher: Searching code database for repairing bugs[C]//Proceedings of the 9th Asia-Pacific Symposium on Internetware. 2017: 1-6
22. Xin Q， Reiss S P. Leveraging syntax-related code for automated program repair[C]//2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE， 2017: 660-670
23. Jiang J， Xiong Y， Zhang H， et al. Shaping program repair space with existing patches and similar code[C]//Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis. 2018: 298-309
24. Hu Y， Ahmed UZ， Mechtaev S， et al. Re-factoring based program repair applied to programming assignments[C]//2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE， 2019: 388- 398
25. Kim D， Nam J， Song J， et al. Automatic patch generation learned from human-written patches[C]//2013 35th International Conference on Software Engineering (ICSE). IEEE， 2013: 802-811
26. Koyuncu A， Liu K， Bissyandé TF， et al. iFixR: Bug report driven program repair[C]//Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering. 2019: 314-325
27. Marginean A， Bader J， Chandra S， et al. Sapfix: Automated end-to-end repair at scale[C]//2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) . IEEE， 2019: 269-278
28. Tian Y， Ray B. Automatically diagnosing and repairing error handling bugs in c[C]//Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 2017: 752-762
29. Gao F， Wang L， Li X. BovInspector: automatic inspection and repair of buffer overflow vulnerabilities[C]//Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. 2016: 786-791
30. Gao Q， Xiong Y， Mi Y， et al. Safe memory-leak fixing for c programs[C]//2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE， 2015， 1: 459-470
31. Yan H， Sui Y， Chen S， et al. Automated memory leak fixing on value-flow slices for c programs[C]//Proceedings of the 31st Annual ACM Symposium on Applied Computing. 2016: 1386-1393
32. Falleri JR， Morandat F， Blanc X， et al. Fine-grained and accurate source code differencing[C]//Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. 2014: 313-324
33. Hua J， Zhang M， Wang K， et al. Towards practical program repair with on-demand candidate generation[C]//Proceedings of the 40th international conference on software engineering. 2018: 12-23
34. Durieux T， Cornu B， Seinturier L， et al. Dynamic patch generation for null pointer exceptions using metaprogramming[C]//2017 IEEE 24th International Conference on Software Analysis， Evolution and Reengineering (SANER). IEEE， 2017: 349- 358
35. Mechtaev S， Gao X， Tan SH， et al. Test-equivalence analysis for automatic patch generation[J]. ACM Transactions on Software Engineering and Methodology (TOSEM)， 2018， 27(4): 1-37
36. Saha S. Harnessing evolution for multi-hunk program repair[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE， 2019: 13-24
37. Tan SH， Yoshida H， Prasad MR， et al. Anti-patterns in search-based program repair[C]//Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2016: 727-738
38. Nguyen HDT， Qi D， Roychoudhury A， et al. Semfix: Program repair via semantic analysis[C]//2013 35th International Conference on Software Engineering (ICSE). IEEE， 2013: 772-781.
39. Mechtaev S， Yi J， Roychoudhury A. Directfix: Looking for simple program repairs[C]//2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE， 2015， 1: 448-458.
40. Mechtaev S， Yi J， Roychoudhury A. Angelix: Scalable multiline program patch synthesis via symbolic analysis[C]//Proceedings of the 38th international conference on software engineering. 2016: 691-701.
41. Rothenberg BC， Grumberg O. Sound and complete mutation-based program repair[C]//International Symposium on Formal Methods. Springer， Cham， 2016: 593-611.
42. Le XBD， Chu DH， Lo D， et al. S3: syntax-and semantic-guided repair synthesis via programming by examples[C]//Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 2017: 593- 604.
43. Mechtaev S， Nguyen MD， Noller Y， et al. Semantic program repair using a reference implementation[C]//Proceedings of the 40th International Conference on Software Engineering. 2018: 129-139.
44. Lee J， Hong S， Oh H. Memfix: static analysis-based repair of memory deallocation errors for c[C]//Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering . 2018: 95-106.
45. van Tonder R， Goues C L. Static automated program repair for heap properties[C]//Proceedings of the 40th International Conference on Software Engineering. 2018: 151-162.
46. Ke Y， Stolee KT， Le Goues C， et al. Repairing programs with semantic code search (t)[C]//2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE， 2015: 295- 306
47. Afzal A， Motwani M， Stolee KT， et al. SOSRepair: Expressive semantic search for real-world program repair[J]. IEEE Transactions on Software Engineering， 2019， 47(10): 2162-2181
48. Yang J， Zhikhartsev A， Liu Y， et al. Better test cases for better automated program repair[C]//Proceedings of the 2017 11th joint meeting on foundations of software engineering. 2017: 831-841.
49. Xin Q， Reiss S P. Identifying test-suite-overfittingted patches through test case generation[C]//Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis. 2017: 226-236
50. Mechtaev S, Nguyen MD, Noller Y, Grunske L, Roychoudhury A. Semantic program repair using a reference implementation. In: Proc. of the 40th Int’l Conf. on Software Engineering (ICSE 2018). 2018. 11−22.
51. Fan L, Rinard M. Automatic patch generation by learning correct code. In: Proc. of the ACM Sigplan-Sigact Symp. on Principles of Programming Languages. ACM Press, 2016. 298−312.
52. Tan SH， Yoshida H， Prasad MR， et al. Anti-patterns in search-based program repair[C]//Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2016: 727-738
53. Cashin P, Martinez C, Weimer W, et al. Understanding automatically-generated patches through symbolic invariant differences[C]//2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019: 411-414.
54. Nilizadeh A， Leavens GT， Le XBD， et al. Exploring true test Overfitting in dynamic automated program repair using formal methods[C]//2021 14th IEEE Conference on Software Testing， Verification and Validation (ICST). IEEE， 2021 : 229-240
55. Ghanbari A. ObjSim: Lightweight automatic patch prioritization via object similarity[C]//Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2020: 541-544.
56. Ye H, Gu J, Martinez M, et al. Automated classification of overfitting patches with statically extracted code features[J]. IEEE Transactions on Software Engineering, 2021, 48(8): 2920-2938.
57. Chen L， Pei Y， Furia C A. Contract-based program repair without the contracts[C]//2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE， 2017: 637-647
58. Fraser G， Arcuri A. Evosuite: automatic test suite generation for object-oriented software[C]//Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. 2011: 416-419.
59. Just R, Jalali D, Ernst M D. Defects4J: A database of existing faults to enable controlled testing studies for Java programs[C]//Proceedings of the 2014 International Symposium on Software Testing and Analysis. 2014: 437-440