A Learning-to-Rank Based Fault Localization Approach using Likely Invariants

Tien-Duy B. Le[1], David Lo[1], Claire Le Goues[2], and Lars Grunske[3]
[1]School of Information Systems, Singapore Management University, Singapore
[2]School of Computer Science, Carnegie Mellon University, USA
[3]Humboldt University of Berlin, Germany
{btdle.2012,davidlo}@smu.edu.sg, clegoues@cs.cmu.edu, grunske@informatik.hu-berlin.de

ABSTRACT

Debugging is a costly process that consumes much developer time and energy. To help reduce debugging effort, many studies have proposed various fault localization approaches. These approaches take as input a set of test cases (some failing, some passing) and produce a ranked list of program elements that are likely to be the root cause of the failures (i.e., failing test cases). In this work, we propose *Savant*, a new fault localization approach that employs a learning-to-rank strategy, using likely invariant *diffs* and suspiciousness scores as features, to rank methods based on their likelihood of being a root cause of a failure. *Savant* has four steps: method clustering and test case selection, invariant mining, feature extraction, and method ranking. At the end of these four steps, *Savant* produces a short ranked list of potentially buggy methods. We have evaluated *Savant* on 357 real-life bugs from 5 programs from the Defects4J benchmark. We find that, on average, *Savant* can identify the correct buggy method for 63.03, 101.72, and 122 bugs at the top 1, 3, and 5 positions in the produced ranked lists. We have compared *Savant* against several state-of-the-art spectrum-based fault localization baselines. We show that *Savant* can successfully locate 57.73%, 56.69%, and 43.13% more bugs at top 1, top 3, and top 5 positions than the best performing baseline, respectively.

CCS Concepts: Software and its engineering → Software testing and debugging

Keywords: Learning to Rank, Program Invariants, Automated Debugging

1. INTRODUCTION

Software systems are often plagued with bugs that compromise system reliability, usability, and security. One of the main tasks involved in fixing such bugs is identifying the associated buggy program elements. Developers can then study the implicated program elements and their context, and make necessary modifications to resolve the bug. This is a time consuming and expensive process. Many real-world projects receive a large number of bug reports daily [6], and addressing them requires considerable time and effort. Debugging can contribute up to 80% of the total software cost for some projects [50]. Thus, there is a pressing need for automated techniques that help developers debug. This problem has motivated considerable work proposing automated debugging solutions for a variety of scenarios, e.g., [4, 7, 8, 14, 21, 30, 32, 36, 44, 48, 53, 54, 55, 62, 63, 64].

In addition to potentially providing direct developer support, automated debugging approaches are also used by recent work in automated program repair (including [28, 40, 35], and many others). Such tools use automated debugging approaches as a first step to identify likely faulty program elements. These lists guide program repair tools to generate program patches that lead previously-failing tests to pass. The accuracy of automated debugging approaches therefore plays an important role in the effectiveness of program repair tools. Thus, there is a need to improve the effectiveness of automated debugging tools further to support both developers and current program repair techniques.

In this work, we are particularly interested in a family of automated debugging solutions that takes as input a set of failing and passing test cases and then highlights the suspicious program elements that are likely responsible for the failures (failing test cases), e.g., [4, 7, 8, 14, 21, 30, 32, 36, 53, 54, 55, 62, 63]. While these techniques have been shown effective in many contexts, their effectiveness needs to be further improved to localize more bugs more accurately.

We propose a novel technique, *Savant*, for effective automated fault localization. *Savant* uses a learning-to-rank machine learning approach to identify buggy methods from failures by analyzing both classic suspiciousness scores and inferred likely invariants observed on passing and failing test cases. *Savant* is built on three high-level intuitions. First, program elements which follow different invariants when run in failing versus correct executions are suspicious. Second, such program elements are even more suspicious if they are assigned higher suspiciousness scores computed by existing spectrum-based fault localization (SBFL) formulas [5, 21, 57, 58, 60]. Third, some invariant differences are likely to be more suspicious than others. For example, the violation of a "non-zero" invariant (e.g., $x \neq 0$ or $x \neq \texttt{null}$) in the failing execution may indicate a division by zero or null pointer dereference, and is thus likely to be more suspicious than a violation of a "linear binary" invariant (e.g., $x + y + 3 = 0$), due to the prevalence of null pointer dereference errors.

ISSTA'16, July 18–20, 2016, Saarbrücken, Germany
© 2016 ACM. 978-1-4503-4390-9/16/07...$15.00
http://dx.doi.org/10.1145/2931037.2931049

There exists natural variations in existing invariant learning techniques when applied to different executions. The challenge is thus to distinguish between invariant differences that arise from natural execution variation and those that are truly indicative of buggy behavior. In the absence of a clear *a priori* set of rules, we propose to learn the relative importance of different invariant differences and suspiciousness scores from pre-existing fixed bugs, and use that learned information to localize new bugs.

Savant is designed for efficiency and reliability, and employs a number of steps to both reduce runtime and make informed recommendations based on the inferred invariants and computed suspiciousness scores. *Savant* works at the method-level granularity [26, 39, 42, 58] rather than file- [45, 64] or statement-level [57, 60]. Although localizing a bug to the file-level is useful, files can be large, leaving considerable code for developers to filter down to the few lines that contain a bug. For example, the faulty file corresponding to Bug 383 in the Closure Compiler (see Figure 1) is 1,160 lines of code in total. Localizing a bug to the line-level, on the other hand, is ill-suited for multi-line bugs, which are common [37]. Furthermore, developers often lack "perfect bug understanding" [41] and thus looking at a single line of code does not always allow a developer to determine whether it is truly buggy, nor to understand the bug well enough to fix it. A method is not as big as a file, but often contains sufficient context needed to help developers understand a bug. Furthermore, Kochhar et al. reported that 51.81% of their 386 survey respondents chose method-level as the preferred granularity for fault localization [24].

Savant consists of four steps: method clustering & test case selection, invariant learning, feature extraction, and method ranking. Unlike prior work, e.g., [4, 21, 36], *Savant* does not use all available test cases to localize faults. Instead, for scalability, it selects a subset of test cases, including both the failing tests and only those passing tests that cover similar program elements. Next, it uses Daikon [17] to learn likely invariants on the entry and exit of only those methods that are executed by the failing executions. By limiting the number of instrumented methods, Daikon completes its learning process much more quickly than it would by default, especially for larger systems. For efficiency reasons, we choose to *kill* Daikon if it does not complete its processing within one minute. We run Daikon in several settings: first, on traces collected from all executions; then only on correct executions; and finally only on failing executions. By *diff*-ing all pairs of resultant invariant sets, we identify suspicious methods where invariants inferred from one set of do not hold in another. Next, we convert the invariant diffs into a set of features. We also use the suspiciousness scores computed by several SBFL formulae for the suspicious methods as features. All of the extracted features are then provided as input to a learning-to-rank algorithm. The learning-to-rank algorithm learns a ranking model based on a training set of fixed bugs which differentiates invariant differences of faulty from non-faulty methods. This ranking model can then be used to rank suspicious methods for new bugs based on their corresponding invariant differences.

We evaluate *Savant* on 357 bugs from 5 programs in the Defects4J benchmark [22]. We compare *Savant* against several state-of-the-art spectrum-based fault localization solutions, as well as a solution that also uses invariants [43]. Out of the 357 bugs, *Savant* successfully localizes 63.08, 101.72, and 122 buggy methods on average within the top 1 , top 3, and top 5 positions in the ranked lists that it generates, respectively. Compared to the existing state-of-the-art , *Savant* successfully locates 57.73%, 56.69%, and 43.13% more bugs at the top 1, top 3, and top 5 positions, respectively.

Table 1: Raw Statistic Description

Notation	Description
$n_f(e)$	Number of failing test cases that execute program element e
$n_f(\bar{e})$	Number of failing test cases that do not execute program element e
$n_s(e)$	Number of passing test cases that execute program element e
$n_s(\bar{e})$	Number of passing test cases that do not execute program element e
n_f	Total number of failing test cases
n_s	Total number of passing test cases

The contributions of our work are as follows:

1. A novel learning-to-rank solution that uses likely invariants and suspiciousness scores as features to rank suspicious methods in a fault localization context.
2. A novel set of heuristics for method clustering and test case selection for invariant inference on large programs for fault localization. Our heuristics combine k-means clustering and a greedy approach to tackle a known NP-complete problem. Without these heuristics, it is largely infeasible to extract interesting features from the execution traces from thousands of test cases.
3. An evaluation showing that, on 357 bugs from 5 programs in the Defects4J benchmark, *Savant* substantially outperforms many previous fault localization solutions that also analyze test cases to produce a ranked list of suspicious methods.

The remainder of this paper is structured as follows. Section 2, presents background on fault localization, Daikon, and learning-to-rank algorithms. Section 3 demonstrates a motivating example. We elaborate the four steps of *Savant* in Section 4. We describe our experimental setup and results in Section 5. We discuss related work in Section 6, and conclude, mentioning future work, in Section 7.

2. PRELIMINARIES

In this section, we define the spectrum-based fault localization (SBFL) problem and present several state-of-the-art approaches (Section 2.1) We then introduce Daikon, a technique that mines likely invariants from program executions (Section 2.2). Finally, we present learning-to-rank algorithms (Section 2.3).

2.1 Spectrum-Based Fault Localization

The goal of spectrum-based fault localization (SBFL) is to rank potentially faulty program elements based on observations of passing and failing test case execution. A common intuition is that program elements executed more often by failed test cases but never or rarely by passing test cases are more likely to be faulty.

An SBFL algorithm therefore takes as input a faulty program version and a set of failing and passing test cases. Running these test cases on an instrumented version of the faulty program produces program *spectra* corresponding to

a set of program elements executed by each of the test cases. SBFL techniques then compute a set of statistics to characterize each program element e that appears in a spectrum, highlighted in Table 1. These statistics inform the computation of a suspiciousness score per program element; SBFL techniques vary in the underlying formulae [56].

We now describe several of the state-of-the-art SBFL approaches proposed in the previous literature. We focus on those to which we compare *Savant* in Section 5; We discuss other related techniques in Section 6.

Xie *et al.* [56] theoretically analyze the manually-created SBFL formulae used in previous studies to compute suspiciousness scores and demonstrate that $ER1$ and $ER5$ are the two best SBFL families. They are computed as follows:

$$ER1^a(e) = \begin{cases} -1, & \text{if } n_f(e) < n_f \\ n_s - n_s(e), & \text{if } n_f(e) = n_f \end{cases}$$

$$ER1^b(e) = n_f(e) - \frac{n_s(e)}{n_s(e) + n_s(\bar{e}) + 1}$$

$$ER5^a(e) = n_f(e)$$

$$ER5^b(e) = \frac{n_f(e)}{n_f(e) + n_f(\bar{e}) + n_s(e) + n_s(\bar{e})}$$

$$ER5^c(e) = \begin{cases} 0, & \text{if } n_f(e) < n_f \\ 1, & \text{if } n_f(e) = n_f \end{cases}$$

Xie *et al.* [57] further analyze SBFL formulae generated by running an automatic genetic programming (GP) algorithm [60]. The best GP-generated SBFL formulae are:

$$GP02(e) = 2 \times (n_f(e) + \sqrt{n_s}) + \sqrt{n_s(e)}$$

$$GP03(e) = \sqrt{|n_f(e)^2 - \sqrt{n_s(e)}|}$$

$$GP13(e) = n_f(e) \times (1 + \frac{1}{2 \times n_s(e) + n_f(e)})$$

$$GP19(e) = n_f(e) \times \sqrt{|n_s(e) - n_f(e) + n_f - n_s|}$$

In addition to the above state-of-the-art SBFL formulae, we also consider Ochiai [5]:

$$Ochiai(e) = \frac{n_f(e)}{\sqrt{n_f(n_f(e) + n_s(e))}}$$

Xuan and Monperrus propose Multric, a compositional SBFL technique that uses a learning-to-rank algorithm [58] to combine the results of 25 previously-proposed SBFL formulae into a model that produces a single ranked list of potentially-buggy program elements. The 25 formulae are used to compute suspiciousness scores of program elements as usual; these scores are then treated as features input to a learning-to-rank algorithm. Our approach, *Savant*, also uses a learning-to-rank algorithm and includes suspiciousness scores as features, but we include a substantively different set of features extracted from invariant differences. Importantly, *Savant* localizes root causes of bugs in suspicious methods where invariant differences occur, rather than in all methods, as Multric does.

Pytlik *et al.* propose Carrot, a SBFL technique that also leverages likely invariants [43]. Carrot mines a set of likely invariants from executions of successful test cases, and observes how the invariants change when executions of failed test cases are incorporated. These differences indicate potential bug locations. *Savant* also uses likely invariants to identify fault locations. However, we use a much larger set of invariatn types (that is, all invariants produced by Daikon, rather than the six considered by Carrot), employ a method clustering and test case selection heuristic for performance, and use invariants and suspiciousness scores as features to rank program elements.

2.2 Mining Likely Invariants

Daikon [16, 17] is a popular tool for mining likely invariants that hold over a set of program executions at specific program points, typically method entries and exits. It monitors the values of variables at such program locations, and matches them against a set of templates to create candidate invariants. Daikon outputs those candidates that hold on all or most of the executions. Daikon supports a wide variety of invariants, from a large set of 311 templates [15]. Examples, taken from Daikon online manual [1], include:

- **LowerBound:** $x >= c$, where c is a constant and x is a long scalar.
- **LinearBinary:** $ax+by+c = 0$, given two long scalars x and y.
- **NonZero:** $x \neq 0$ or $x \neq$ `null` for long scalar integers or pointers, respectively.
- **EltwiseFloatGreaterEqual:** Given a sequence of `double` values, represents the invariant that adjacent subsequent values are constrained by the $\geq$ relation. Prints as "x[] sorted by >=."

2.3 Learning-to-Rank

Learning-to-rank is a family of *supervised* machine learning techniques solves ranking problems in information retrieval [33]. There are two phases in learning to rank: a learning, and a deployment phase. During the learning phase, learning to rank techniques extract features from training data which contain queries and documents as well as their relevance labels. The output of the learning phase is a ranking model that can predict relevance labels for documents with regard to corresponding queries. The goal of the learning algorithm is to infer an optimal way to combine features that minimizes the loss function; Each learning to rank technique has a specific loss function and learning algorithm to construct this model.

In the deployment phase, the learned ranking model receives new queries and documents, and returns a ranked list of documents sorted by their computed relevance to the input queries. There are three major approaches in learning to rank [33]: pointwise, pairwise, and listwise. Each varies in input, output, and loss functions. In our study, we use rankSVM [29], a pairwise technique, for learning and ranking suspicious program elements.

3. MOTIVATING EXAMPLE

We begin by introducing an example defect to motivate our technique. Consider Bug 383 in the Closure Compiler's bug database[1], summarized in the top part of Figure 1. The bug is assigned a high priority: it causes an issue in Internet Explorer 9 and `jQuery.getScript`. According to the developer patch (bottom of Figure 1), the bug ultimately resides in the `strEscape` method in the `com.google.javascript.jscomp.CodeGenerator` class. To find this method, based on the report, the developer can

[1]https://goo.gl/YtW6Ux

Bug 383 (Priority: high)

`Summary:` `\0` `\x00` and `\u0000` are translated to null character

`Description:`
What steps will reproduce the problem?
1. write script with string constant "\0" or "\x00" or "\u0000"

What is the expected output? What do you see instead?
I expected a string literal with "\0" (or something like that) and instead get a string literal with three null character values.

Please provide any additional information below.
This is causing an issue with IE9 and jQuery.getScript. It causes IE9 to interpret the null character as the end of the file instead of a null character.

```
@@ -963,6 +963,7 @@ class CodeGenerator {
     for (int i = 0; i < s.length(); i++) {
         char c = s.charAt(i);
         switch (c) {
+          case '\{}0': sb.append("\{}\{}0"); break;
           case '\{}n': sb.append("\{}\{}n"); break;
           case '\{}r': sb.append("\{}\{}r"); break;
```

Figure 1: Bug Report (top) and developer patch (bottom) for bug 383 of the Closure Compiler

create test cases to expose the undesired behavior (e.g., returning a null value for "\0"). With these test cases and previously-created passing tests (of which Closure has 6,740), the developer could use existing spectrum-based fault localization formulae to generate a ranked list of methods. The list could then be inspected, in order, until the root cause of the bug is localized.

Generally, the ranked list contains the methods invoked by failing test cases. However, the Closure Compiler is a large project: This bug implicates 6,646 methods! Moreover, none of ten state-of-the-art SBFL formulae (Section 2.1) localize the actual faulty method within the top 10 of the produced list. The two best-performing formulae for this defect ($GP13$ and $GP19$) rank the first faulty method at position #64. Multric assigns the highest suspiciousness score to the faulty method; however, there are more than 1000 methods sharing this score. Existing SBFL approaches provide limited utility for the developer in this case.

On the other hand, *Savant* first uses Daikon to infer invariants from execution traces generated by both passing and failing test cases. Daikon's Invariant Diff utility on the inferred invariant sets implicates 556 suspicious methods with changes in learned invariants between passing versus failing execution types. Assuming any one of these methods is invoked by the failing test cases, we have already reduced the number of implicated methods from more than 6,000 to 556, regardless of SBFL rank. However, this is still an intractably large number. Therefore, we further rank the output using a learning to rank model built on historical changes in invariants of previously fixed bugs. The model assigns a score to each of the identified suspicious methods. We create a ranked list of suspicious methods sorted by the computed scores, and send to the developer for inspection.

Savant localizes the methods implicated in this example within the top-3 program elements (Section 5), significantly outperforming the best SBFL approaches.

4. PROPOSED APPROACH

An overview of *Savant*'s architecture is shown in Figure 2. *Savant* works in two phases. In the training phase, *Savant* learns a statistical model that ranks methods based on the likelihood that they are the root cause of a particular failure, based on features drawn from execution behavior on passing and failing test cases. This model is learned from training data consisting of a set of previously-repaired bugs, consisting of a buggy program version, passing and failing test cases, and ground truth bug locations. The training phase of *Savant* consists of four steps:

- *Method Clustering and Test Case Selection* (Section 4.1): *Savant* first clusters methods executed by failing test cases. For every cluster, *Savant* selects a subset of particularly relevant passing and failing test cases. This step thus outputs a set of method clusters and their corresponding selected test cases. The goal of this step is to limit the memory and runtime cost during invariant mining, while still enabling the inference of useful invariants.
- *Invariant Mining* (Section 4.2): *Savant* uses Daikon to record and infer method invariants for each cluster, based on program behavior on various sets of test cases. This step produces sets of program invariants, inferred from the execution of failing test cases, passing test cases, and their combination.
- *Feature Extraction* (Section 4.3): *Savant* uses the Daikon's Invariant Diff utility to identify differences between method invariants inferred from the execution behavior on different sets of test cases. For instance, if invariant I holds over the set of passing test case executions, but not when the failing test cases are added, the methods where invariant I differs are *suspicious*. For methods whose invariant diffs are *non-empty*, *Savant* runs SBFL formulae to obtain suspiciousness scores. This produces a set of features per method for use as input to a model learning procedure, corresponding to either (1) the frequency of a type of invariant change for the method, or (2) a suspiciousness score computed by one of the SBFL formulae.
- *Model Learning* (Section 4.4): *Savant* takes the features and the ground truth locations to build a ranking *model*. This model is the overall output of the training phase that is passed to the deployment phase.

In the deployment phase (also discussed in Section 4.4), *Savant* takes as input a set of test cases (some failing, some passing), and a buggy program version and then uses the learned model to rank produce a ranked list of methods that are likely responsible for the failing test cases.

4.1 Method Clustering & Test Case Selection

Savant must run the faulty program on passing and failing tests to record execution traces for invariant inference. Both trace collection and invariant mining can be very expensive, and the number of instrumented methods and executed test cases both contribute to the cost. For medium to large programs (e.g., Commons Math, Closure Compiler etc.), the runtime cost of running Daikon to infer invariants for all methods executed by all test cases is very high (i.e., Daikon runs for many hours without producing results, or crashes with an out-of-memory exception). At the same time, we must collect sufficient data to support precise and useful invariant inference.

We resolve this tension via a set of novel heuristics for method clustering and test case selection for invariant inference on large programs for the purpose of fault localization. First, we exclude all methods not executed by any failing

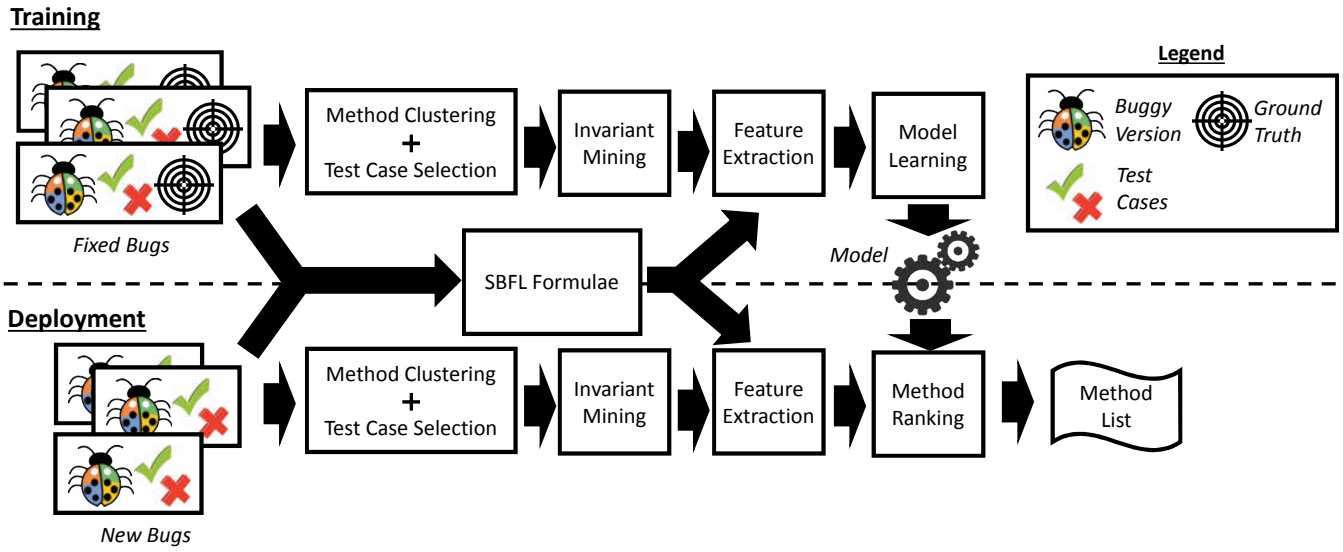

Figure 2: Overview of *Savant*'s Architecture

Algorithm 1 Method Clustering & Test Case Selection

Input: I: all methods executed by unsuccessful test cases
P_i, F_i : passing and failing test cases, respectively
M: maximum cluster size
T: minimum acceptable coverage
Output: Clusters of methods and associated test suites
let $Cs \leftarrow$`reduce_size`$(kmeans(I, \frac{|I|}{M}))$
let $C_r \leftarrow \emptyset$
foreach $c_m \in Cs$ **do**
  let $P_i^s \leftarrow$ `coverage_sort`(P_i, c_m)
  let $P_c \leftarrow \emptyset$
  while $\exists m \in c$ *s.t.* `coverage`$(m, P_c) < M$ **do**
    let $t \leftarrow$ `pop`(P_i^s)
    if *t covers at least one method* $m \in c$ **then**
      $P_c \leftarrow P_c \cup \{t\}$
    end
  end
  $C_r \leftarrow C_r \cup \{c_m, P_c\}$
end
return C_r

test cases. Second, we *cluster* methods by the passing test cases that execute them, and record execution traces for each cluster separately on all of the failing and then a selected subset of the passing test cases. Considering all passing test cases is unnecessarily costly, as many passing test cases are irrelevant to particular sets of methods.

Algorithm 1 describes our clustering and test case selection heuristic. This heuristic represents each method via a *coverage vector* describing all input passing test cases. The value of a dimension is one if the method is covered by the corresponding test case and zero otherwise. *Savant* uses k-means clustering [20] to group similar vectors (i.e., methods). K-means takes as input a set of vectors V and a fixed number of desired clusters k, and produces k clusters, each containing an arbitrary number of methods. We set k to $\|I\|/M$, where I is the number of methods to cluster, and M is the desired maximum cluster size. Because the resulting clusters may contain more than M methods, we heuristically split overlarge clusters into smaller groups no larger than size M. We keep selecting M methods randomly to create new groups until the cluster size is reduced to no more than M.

Savant then selects a subset of passing test cases for each cluster that covers its methods at least T times each. A perfect solution reduces to the NP-complete knapsack problem. We take a computationally simpler greedy approach, which suffices for our purpose. For each cluster, we sort passing test cases in descending order by the number of methods in the cluster that each test covers. We then greedily take test cases from this list until all methods in the cluster are covered at least by at least T test cases. Both M and T can be tuned to fit experimental system capacity. Note that as the number of failing test cases is usually significantly less than the number of passing tests, we are still able to use all failing test cases of the faulty program for each cluster. The output of this step is the set of clusters C_r, eaching containing methods, where for each method cluster there is an associated subset P_c of passing test cases.

4.2 Invariant Mining

For each cluster produced in the preceding step, *Savant* traces execution information for each included method across all failing test cases as well as the selected passing test subset for that cluster. *Savant* collects execution information separately for the methods in the cluster uses the following three sets of test cases: F_i, P_c, and $F_i \cup P_c$ of each method cluster c. For clarity, we refer subsequently to the execution of the failing test cases on a cluster c as F_c, but remind the reader that this involves running all failing test cases on each cluster. This information is then input to Daikon to infer invariants. We refer to the invariants inferred by Daikon as `inv`(F_c), `inv`(P_c), and `inv`$(F_c \cup P_c)$, respectively, and these sets of invariants form the output of this step. *Savant* uses these sets to produce features for the learning-to-rank model construction, discussed next.

4.3 Feature Extraction

Savant extracts two different types of features: *invariant changes* features and *suspiciousness scores* features.

4.3.1 Invariant Change Features

For the first set of features, *Savant* uses Daikon's Invariant Diff tool to describe changes in invariant sets between failing and passing program executions. In a nutshell, Invariant Diff recognizes changes in method invariants inferred from different sets of execution traces. The changes can consist of a transformation of an invariant into another (e.g., *OneOfString* to *SingleString*, or *NonZero* to *EqualZero*), or invariant removal or addition. Our overall insight is that if an invariant I holds over successful test case executions (i.e., $I \in \texttt{inv}(P_c)$), but not when the failing test cases are added (i.e., $I \notin \texttt{inv}(F_c \cup P_c)$), the locations where invariant I differs are suspicious. These suspicious locations are at the entry and exit point of a method, suggesting the bug lies within that method. Thus, *Savant* ultimately analyzes and ranks only these suspicious methods, rather than all methods covered by the failing test cases.

Savant uses Invariant Diff to perform three types of comparisons per cluster: $\texttt{inv}(P_c) \times \texttt{inv}(F_c \cup P_c)$, $\texttt{inv}(F_c) \times \texttt{inv}(P_c)$, and $\texttt{inv}(F_c) \times \texttt{inv}(F_c \cup P_c)$. We refer to the output of Invariant Diff on these pairs as $\texttt{idiff}(P_c, F_c \cup P_c)$, $\texttt{idiff}(F_c, P_c)$, and $\texttt{idiff}(F_c, F_c \cup P_c)$, respectively. We then convert these invariant changes to features. Feature f of method m is a 3-tuple: $f = [I_A, I_B, \texttt{InvDiff}_{AB}]$. I_A is the type of the source invariant inferred from the execution of one set of test cases, I_B is the type of the target invariant to which I_A is transformed, and which holds in the execution of the other set of tests, and $\texttt{InvDiff}_{AB}$ indicates the Invariant Diff's output type from which the change was discovered. We refer to I_A and I_B as the left-hand side (LHS) and right-hand side (RHS) invariant, or the *source* and *target* invariants, interchangeably. The value of feature f is then the frequency of the change between I_A and I_B in an $\texttt{InvDiff}_{AB}$ comparison. Note that the values of I_A and I_B are the invariant types, rather than concrete invariants. Similarly, the value of $\texttt{InvDiff}_{AB}$ in a feature's 3-tuple is a label rather the concrete value of Invariant Diff's output. For example: [**OneOfString, SingleString,** $\texttt{idiff}(F_c, P_c)$] can be read as "A OneOfString invariant learned from execution of F_c becomes a SingleString invariant in the execution of P_c."

Daikon supports many different types of invariants, including abstractions of concrete invariants. For example, UnaryInvariant abstracts the LowerBound and NonZero invariants. These invariant types create an inheritance hierarchy rooted at `daikon.inv.Invariant`[2]. Thus, we enrich the feature set by replacing the LHS and RHS invariants of a feature with their abstract types to form a new feature. For example, [**UnaryInvariant,SingleString,** $\texttt{idiff}(F_c, P_c)$] abstracts [**OneOfString, SingleString,** $\texttt{idiff}(F_c, P_c)$], and [**UnaryInvariant, SingleFloat,** $\texttt{idiff}(F_c, F_c \cup P_c)$] abstracts [**LowerBoundFloat, SingleFloat,** $\texttt{idiff}(F_c, F_c \cup P_c)$].

Each such feature is a pair of invariants that reflects an abstraction of a method's behavioral changes, and collecting many such features improves the chances that *Savant* successfully captures distinctive changes in the behavior of faulty methods. `Invariant` has 311 subclasses, and thus we have $311 \times 311 = 96,721$ potential invariant pairs (and thus overall features) across the three different Invariant Diff runs i.e., $\texttt{idiff}(P_c, F_c \cup P_c)$, $\texttt{idiff}(F_c, P_c)$, and $\texttt{idiff}(F_c, F_c \cup P_c)$.

[2]http://goo.gl/EPwNhV

4.3.2 Suspiciousness Scores Features

It is possible for methods to share similar changes to their invariants between sets of execution traces. These cases are more difficult for ranking models to distinguish faulty and non-faulty methods. *Savant* therefore also includes suspiciousness scores output by spectrum-based fault localization tools as additional features. For each method implicted by changed invariants, *Savant* computes the suspiciousness scores output by 10 state-of-the-art spectrum-based fault localization formulae (Section 2.1): $ER1^a$, $ER1^b$, $ER5^a$, $ER5^b$, $ER5^c$, $GP02$, and $GP03$, $GP13$, $GP19$ and Multric. We also include the suspiciousness scores output by the 25 SBFL formulae (including Ochiai) used by Multric, resulting in 35 features extracted from suspiciousness scores. These features and invariant change features are then forwarded to the model learning and method ranking steps.

4.4 Model Learning and Method Ranking

Feature Normalization. Before learning ranking models, we normalize feature values to a range of $[0, 1]$, as follows:

$$v_i' = \begin{cases} 0 & \text{if } v_i < \min_i \\ \dfrac{v_i - \min_i}{\max_i - \min_i} & \text{if } \min_i \leq v \leq \max_i \\ 1 & \text{if } v_i > \max_i \end{cases} \quad (1)$$

where v_i and v_i' are the original and normalized values of the i^{th} feature of a suspicious method, $\min_i$ and $\max_i$ are the minimum and maximum values of the i^{th} feature inferred from the training data.

Model Learning and Method Ranking. In the model learning step, *Savant* takes as input a set of fixed bugs, their corresponding features, and their ground truth faulty methods. The methods modified by the developers to fix the bugs provide this ground truth. For some bugs in the training set, the difference between the invariants of their faulty methods (for failed and passing test cases) can be empty. We exclude such bugs from the training set. Figure 3 shows the format of input data handled by the model learning step. Given input data in this format, we use rankSVM [29], an off-the-shelf learning-to-rank algorithm, to learn a statistical model that ranks methods based on such features.

In the method ranking step, our approach takes the features generated for a new bug as input to the learned model. Finally, *Savant* outputs a ranked list produced by the learned model to the developer for inspection.

5. EXPERIMENTS

In this section, we empirically evaluate *Savant* on a dataset of real-world Java bugs. We discuss the dataset, experimental settings, and metrics in Section 5.1. Section 5.2 lists our research questions, followed by our findings in Section 5.3. We discuss threats to validity in Section 5.4.

5.1 Experimental Settings

Dataset. Many fault localization approaches [21, 5, 36, 58] are evaluated on artificial bugs (e.g., the SIR benchmark[3],

[3]http://sir.unl.edu/php/previewfiles.php

	Faulty Program	feature$_1$	feature$_2$	feature$_3$	...	label
Method #1	1	$x_1^{(1,1)}$	$x_2^{(1,1)}$	$x_3^{(1,1)}$	...	$y^{(1,1)}$
Method #2	1	$x_1^{(1,2)}$	$x_2^{(1,2)}$	$x_3^{(1,2)}$		$y^{(1,2)}$
			...			
Method #1	2	$x_1^{(2,1)}$	$x_2^{(2,1)}$	$x_3^{(2,1)}$	...	$y^{(2,1)}$
Method #2	2	$x_1^{(2,2)}$	$x_2^{(2,2)}$	$x_3^{(2,2)}$		$y^{(2,2)}$
			...			
			...			

Figure 3: Input Data Format for Model Learning. $x_k^{(i,j)}$ corresponds to the value of feature k for method j in faulty program i. $y^{(i,j)}$ corresponds to the label (i.e., faulty or non-faulty) of method j for faulty program i.

Table 2: Dataset: The bolded components of names denote a shorthand abbreviation. "# Bugs" represents the number of bugs in each project. "Avg. KLOC", "Avg. Tests", and "Avg. Methods" correspond to average size of the program, number of test cases, and number of methods for each buggy version of each program, respectively.

Program	# Bugs	KLOC	Avg. Tests	Methods
JFree**Chart**	26	132.9	1,824.9	7,782.5
Closure Compiler	133	345.6	7,200.1	7,479.5
Commons **Math**	106	111.8	2,905.0	4,792.3
Joda-**Time**	27	110.8	3,924.6	4,083.5
Commons **Lang**	65	52.6	1,859.0	2,151.1

Steimann et al.'s benchmark [49]). However, it is unclear whether such bugs capture true characteristics of real bugs in real programs. Therefore, we evaluate *Savant* on 357 bugs from 5 different software projects in the Defects4J benchmark [22], a database of real, isolated, reproducible software faults from real-world open-source Java projects intended to support controlled studies in software testing. The projects include a large number of test cases, and there exists at least one failing test case per bug. Our choice of evaluation benchmark is inspired by influential previous work in the field [61, 31] that evaluates proposed fault localization approaches on real faulty programs. Table 2 describes the bugs and projects in the evaluation benchmark.

Comparative techniques. We compare *Savant* against the SBFL techniques highlighted in Section 2, as well as Carrot. We set the granularity of localized program entity to method, to match *Savant*. We extend Carrot to use all Daikon invariants and benefit from our test case selection strategy. This is important for scalability: without selection, Carrot takes hours to complete. The extended Carrot approach is referred to as Carrot$^+$. In total, we compare *Savant* against 12 baselines.

Cross Validation. We perform leave-one-out cross validation [19] across each of the five projects. Given a set of n bugs, we divide the set into n groups, of which $n-1$ are used for training and the remaining serves as the test set. We repeat the process n times, using a different group as the test set. We report *total* results across the n iterations. Compared to the standard 10-fold cross validation, leave-one-out cross validation is beneficial for evaluating on smaller datasets, as it provides more training data for each iteration, at the expense of training and evaluation time. *Savant* and Multric are the only two supervised learning techniques we evaluate, and thus we only perform cross validation for these two techniques, to mitigate the risk of overfitting. The other techniques are unsupervised.

Savant's Settings. For method clustering and test case selection, we set the maximum cluster size $M = 10$ and minimum acceptable size $T = 10$. *Savant* uses Daikon (version 5.2.8[4]) to infer invariants, scikit-learn[5] 0.17.0 to perform k-means clustering, and rankSVM with linear kernel (version 1.95[6]) from LIBSVM toolkit [13] for the learning to rank task with default settings. We perform all experiments on an Intel(R) Xeon E5-2667 2.9 GHz system with Linux 2.6.

Metrics. We use three metrics to evaluate fault localization success:

acc@n counts the number of successfully localized bugs within the top-n position of the resultant ranked lists. We use absolute ranks rather than percentages, following findings suggesting that programmers will only inspect the top few positions in a ranked list of potentially buggy statements [41]. We choose $n \in \{1, 3, 5\}$, computing acc@1, acc@3, and acc@5 scores. Note that if two program elements (i.e., methods) share the same suspiciousness score, we *randomly* break the tie. Higher is better for this metric.

wef@n approximates the *wasted effort at n*, or effort wasted by a developer on non-faulty program elements before localizing the root cause a bug. *wef@n* is calculated by the total number of non-faulty program elements in top-n positions of ranked lists before reaching the first faulty program element, or the n$^{\text{th}}$ program element in the ranked lists of all bugs. We again choose $n \in \{1, 3, 5\}$. Smaller is better.

Mean Average Precision (MAP) evaluates ranking methods in information retrieval; we use it to evaluate the ranked list of suspicious elements produced by fault localization techniques. MAP is calculated using the mean of the *average precision* of all bugs, as follows:

$$AP = \sum_{i=1}^{M} \frac{P(i) \times pos(i)}{number\ of\ faulty\ methods}$$

where i is a rank of the method at the i^{th} position in the ranked list, M is total number of methods in the ranked list, and $pos(i)$ is a boolean function indicating whether the i^{th} method is faulty. $P(i)$ is the precision at i^{th} position starting from the beginning, defined as:

$$P(i) = \frac{\#faulty\ methods\ in\ the\ top\ i}{i}.$$

In cross-validation, we compute the Mean Average Precision (MAP) across all average precisions output across n iterations. Higher is better. Note that MAP is a very strict evaluation metric and its score is typically low (< 0.5) [26,

[4]http://plse.cs.washington.edu/daikon/download/
[5]http://scikit-learn.org/
[6]https://goo.gl/pHku7x

Table 3: *Savant*'s effectiveness in terms of average acc@n ($n \in \{1,3,5\}$), wef@n ($n \in \{1,3,5\}$) and MAP.

P	Total Bugs	Avg. acc @1	Avg. acc @3	Avg. acc @5	Avg. wef @1	Avg. wef @3	Avg. wef @5	Avg. MAP
Chart	26	5.00	8.00	9.00	20.00	56.00	86.00	0.201
Closure	133	2.00	9.00	13.00	131.00	384.00	627.00	0.041
Math	106	22.03	36.72	47.00	82.97	226.14	348.14	0.261
Time	27	5.00	12.00	12.00	22.00	55.00	85.00	0.247
Lang	65	29.00	36.00	41.00	33.00	90.00	131.00	0.535
Overall	357	63.03	101.72	122.00	288.97	811.14	1277.14	0.221

45, 64]. For bugs appearing in single a method, even if all faulty methods appear in the top-3 position, the MAP score is only 0.33.

In spectrum-based fault localization, program elements are often assigned the same suspiciousness score. Thus, we repeat all metric calculations 100 times, using 100 different seeds to randomly break ties.

5.2 Research Questions

We investigate the following research questions:

RQ1: How effective is *Savant*? In this research question, we evaluate how effectively *Savant* identifies buggy methods for the 357 bugs, computing average acc@n, wef@n, and MAP scores ($n \in \{1,3,5\}$).

RQ2: How does *Savant* compare to previous approaches? In this research question, we compare *Savant* to 12 previous techniques across all evaluation metrics.

RQ3: What is the impact of the different feature sets on performance? By default, we use all features from both Invariant Diff and the suspiciousness scores. In this research question, we compare the two types of features to evaluate their individual contribution to *Savant*'s effectiveness by training and evaluating models on each set of features independently.

RQ4: How much training data does *Savant* need to work effectively? In the default setting, *Savant* uses $n-1$ out of n bugs as training data. We investigate the effectiveness of *Savant* with reduced amount of training data. We do this by evaluating *Savant* in a k-fold cross validation setting, for k ranging from 2–10.

RQ5: How efficient is *Savant*? In this research question, we measure the average running time needed for *Savant* to output a ranked list of methods for a given bug.

5.3 Findings

RQ1: *Savant*'s Effectiveness. Table 3 shows the effectiveness of *Savant* on bugs from the five Defects4J projects. *Savant* successfully localizes 63.03, 101.72, and 122.00 out of 357 bugs in terms of average acc@1, acc@3, and acc@5 score, respectively. The wasted effort across all projects is 288.97, 811.14, and 1277.14 (wef@1, wef@3, and wef@5, respectively). The overall average MAP score is 0.221. Among the five projects, *Savant* is most effective on Commons Lang, achieving the highest *acc@k* (29, 36, and 41, respectively), and a MAP score of 0.535. Over these experiments, *Savant* terminated Daikon twice on a bug from the Math project due to the time limit. In total, we invoked Daikon 129,798 times for the 357 faulty versions.

RQ2: *Savant* vs. Previous work.

Table 4 shows the effectiveness of the 12 baseline approaches on our dataset. Among the baselines, the top four performers are ER1^b, *GP*13, *GP*19 and Multric, and they achieve more or less the same score for many metrics. The absolute best performers are ER1^b and *GP*13.

Savant outperforms these baselines in all metrics. It outperforms ER1^b and *GP*13 by 57.73%, 56.69%, and 43.13% in terms of average acc@1, acc@3, and acc@5 scores. The wasted effort of *Savant* is lower than those of ER1^b and *GP*13 by 8.85%, 10.94%, and 12.78% (wef@1, wef@3, and wef@5 respectively). In terms of average MAP score, our approach is more effective than ER1^b and *GP*13 by 51.37%.

Table 4: Effectiveness of Baseline Approaches; We use the shorthand names for programs for brevity. "B" stands for baseline approaches, "P" is the project name, "MUL" stands for Multric , and "OA" is the overall results.

B	P	Average acc @1	Average acc @3	Average acc @5	Average wef @1	Average wef @3	Average wef @5	MAP
ER1^a	Chart	0.93	2.42	3.50	25.07	72.83	118.27	0.08
	Closure	2.61	6.15	9.26	130.39	385.35	634.22	0.04
	Math	8.31	19.70	28.80	97.69	275.74	434.00	0.16
	Time	1.00	4.48	7.00	26.00	72.52	114.00	0.07
	Lang	2.46	17.86	25.54	62.54	164.62	246.95	0.18
	OA	15.31	50.61	74.10	341.69	971.06	1547.44	0.11
ER1^b	Chart	4.00	4.00	6.00	22.00	66.00	108.00	0.15
	Closure	2.61	6.15	9.26	130.39	385.35	634.22	0.04
	Math	8.35	19.77	28.89	97.65	275.57	433.66	0.16
	Time	3.00	3.00	3.00	24.00	72.00	120.00	0.05
	Lang	22.00	32.00	38.09	43.00	112.50	168.38	0.37
	OA	39.96	64.92	85.24	317.04	911.42	1464.26	0.15
ER5^a *ER5*^{b}	Chart	4.00	4.49	5.56	22.00	65.44	107.00	0.15
	Closure	0.31	0.69	1.17	132.69	397.48	661.40	0.01
	Math	2.41	7.32	11.68	103.59	303.63	494.48	0.06
	Time	3.00	3.00	3.00	24.00	72.00	120.00	0.05
	Lang	23.20	31.83	38.32	41.80	113.22	169.24	0.38
	OA	32.92	47.33	59.73	324.08	951.77	1552.12	0.10
ER5^c	Chart	0.97	2.95	4.64	25.03	72.04	115.56	0.07
	Closure	0.31	0.69	1.17	132.69	397.48	661.40	0.01
	Math	2.36	7.21	11.54	103.64	303.87	494.99	0.06
	Time	0.25	0.85	1.37	26.75	79.40	130.92	0.02
	Lang	5.37	15.60	24.27	59.63	163.13	248.92	0.17
	OA	9.26	27.30	42.99	347.74	1015.92	1651.79	0.06
*GP*02	Chart	0.00	0.00	0.00	26.00	78.00	130.00	0.01
	Closure	0.10	0.30	0.50	132.90	398.36	663.49	0.00
	Math	0.15	0.47	0.83	105.85	317.01	527.50	0.01
	Time	0.00	0.00	0.00	27.00	81.00	135.00	0.00
	Lang	8.00	11.00	12.12	57.00	167.00	273.78	0.12
	OA	8.25	11.77	13.45	348.75	1041.37	1729.77	0.03
*GP*03	Chart	0.00	1.00	2.00	26.00	76.00	125.00	0.02
	Closure	0.65	1.01	1.02	132.35	396.37	660.34	0.01
	Math	0.31	0.90	1.37	105.69	316.14	525.60	0.01
	Time	0.00	0.00	0.00	27.00	81.00	135.00	0.01
	Lang	8.00	13.00	16.00	57.00	163.00	261.00	0.14
	OA	8.96	15.91	20.39	348.04	1032.51	1706.94	0.03
*GP*13	Chart	4.00	4.00	6.00	22.00	66.00	108.00	0.15
	Closure	2.61	6.15	9.26	130.39	385.35	634.22	0.04
	Math	8.35	19.77	28.89	97.65	275.57	433.66	0.16
	Time	3.00	3.00	3.00	24.00	72.00	120.00	0.05
	Lang	22.00	32.00	38.09	43.00	112.50	168.38	0.37
	OA	39.96	64.92	85.24	317.04	911.42	1464.26	0.15
*GP*19	Chart	4.00	4.00	6.00	22.00	66.00	108.00	0.15
	Closure	2.60	6.11	9.22	130.40	385.42	634.37	0.04
	Math	8.35	19.77	28.89	97.65	275.57	433.66	0.16
	Time	3.00	3.00	3.00	24.00	72.00	120.00	0.05
	Lang	22.00	32.00	38.09	43.00	112.50	168.38	0.37
	OA	39.95	64.88	85.20	317.05	911.49	1464.41	0.15
Ochiai	Chart	2.00	4.00	6.00	24.00	69.00	109.00	0.12
	Closure	1.95	4.70	8.77	131.05	388.77	639.29	0.04
	Math	8.27	19.71	28.82	97.73	275.71	433.99	0.16
	Time	5.50	7.00	9.00	21.50	62.50	99.50	0.12
	Lang	18.84	28.50	35.09	46.16	125.47	187.35	0.34
	OA	36.56	63.91	87.68	320.44	921.45	1469.13	0.14
MUL	Chart	4.35	6.44	8.18	21.65	62.14	98.35	0.15
	Closure	1.83	5.14	7.84	131.17	388.24	639.73	0.03
	Math	6.33	16.88	25.36	99.67	283.34	448.52	0.14
	Time	3.71	5.47	6.52	23.29	67.33	108.65	0.10
	Lang	22.80	30.04	34.06	42.20	113.80	177.57	0.36
	OA	39.02	63.97	81.96	317.98	914.85	1472.82	0.14
Carot$^+$	Chart	2.05	3.86	5.16	22.95	65.95	104.24	0.11
	Closure	0.35	1.01	1.88	132.65	396.99	659.67	0.01
	Math	5.81	13.84	20.48	99.19	284.99	457.32	0.10
	Time	0.90	1.86	2.36	26.10	76.69	126.25	0.06
	Lang	20.48	33.84	39.58	41.52	102.21	146.49	0.41
	OA	29.59	54.41	69.46	322.41	926.83	1493.97	0.12

Savant's improvement

Legend 10% ≤ improvement <20% | 20% ≤ improvement <50% | 50% ≤ improvement <100% | 100% ≤ improvement

Table 5: *Savant*'s effectiveness using different features.

Feature Set	Avg. acc @1	Avg. acc @3	Avg. acc @5	Avg. wef @1	Avg. wef @3	Avg. wef @5	Avg. MAP
Inv. Changes	55.23	83.50	105.00	296.77	848.64	1346.14	0.179
Susp. Scores	53.81	86.56	105.09	298.19	845.11	1340.82	0.214
Default	63.03	101.72	122.00	288.97	811.14	1277.14	0.221

Overall, *Savant* outperforms all popular and state-of-the-art baseline approaches across all measured metrics.

Note that although Multric includes Ochiai as a feature, it does not considerably outperforms Ochiai. Multric outperforms Ochiai by 6.73% and 0.09% in terms of average acc@1 and acc@3 scores, and the wasted effort of Multric is only lower than that of Ochiai by 0.77% and 0.72% in terms of average wef@1 and wef@3 score. For the other metrics (average acc@10, wef@10 and MAP scores), Ochiai outperforms Multric. This result is in contrast with that of *Savant* which outperforms all existing techniques by a much larger margin (e.g., 57.73% versus 6.73% for acc@1).

RQ3: Different Sets of Features. Table 5 shows the effectiveness of *Savant* using invariant change features, suspiciousness scores features, and their combination (the default). *Savant* is less effective if only one type of features is used to construct ranking models, across all metrics. *Savant* using only suspiciousness scores is more accurate than using only invariant changes according to most metrics, though notably *not* acc@1, nor wef@1. Regardless of the features used in the model, and unlike the baselines, *Savant* only ranks methods that have changes in invariants in the two set of execution traces, instead of all methods in faulty programs. This explains why *Savant* built using only suspiciousness score features outperforms Multric. Overall, the combination of the two feature types significantly improves the effectiveness of *Savant*.

Table 6: Varying training data size: average acc@n ($n \in \{1,3,5\}$), wef@n ($n \in \{1,3,5\}$) and Mean Average Precision (MAP). "K" represents the number of folds in cross-validation setting.

K	Avg. acc @1	Avg. acc @3	Avg. acc @5	Avg. wef @1	Avg. wef @3	Avg. wef @5	Avg. MAP
10	61.53	94.72	118.50	290.47	825.64	1298.64	0.215
9	53.03	90.72	110.50	298.97	844.14	1335.14	0.204
8	63.85	94.98	109.69	288.15	815.86	1302.45	0.219
7	68.35	102.19	111.50	283.65	797.83	1276.33	0.220
6	59.35	96.48	117.69	292.65	818.36	1294.45	0.222
5	68.85	94.50	112.00	283.15	809.02	1292.52	0.223
4	63.53	92.22	110.05	288.47	817.14	1307.59	0.219
3	62.50	93.00	114.46	289.50	822.50	1307.54	0.216
2	59.68	90.07	110.00	292.32	825.27	1314.61	0.211
Default	63.03	101.72	122.00	288.97	811.14	1277.14	0.221

RQ4: Varying Training Data Size. Table 6 shows the effectiveness of *Savant* in various cross validation settings. Although the effectiveness of *Savant* varies from setting to setting, the range of effectiveness is fairly small. Among the settings, 5-fold cross validation ($k = 5$) achieves the best performance in most metrics compared to others, but there is no setting that outperforms the others across all metrics. We conclude that the amount of training data has little impact on *Savant*'s accuracy.

RQ5: Efficiency. Table 7 shows average running time for *Savant* on our dataset, including both learning and ranking. The average time to output a ranked list of methods for a given bug from any of the five projects is 13.894 seconds,

Table 7: Running time of *Savant* (in seconds)

Project	Mean	Standard Deviation
JFreeChart	1.530	0.285
Closure Compiler	31.845	3.971
Commons Math	4.101	0.609
Joda-Time	2.378	0.459
Commons Lang	1.970	0.341
Overall	13.894	14.232

with a standard deviation of 14.232 seconds. This average is dominated by the running time on a single project (the Closure Compiler); the median running time is 2.378 seconds (Joda-Time). Among the projects, *Savant* has lowest average execution time on JFreeChart bugs (1.53 seconds). Closure Compiler bugs take the longest to localize. The Closure Compiler is considerably larger than the other projects, leading to a longer running time in constructing ranking models. We observe, however, that this running time is conservative, since it includes the training phase, which could be amortized across different bug localization efforts, and overall is reasonable in practice.

5.4 Threats to Validity

Threats to internal validity relate to experimental errors and biases. We mitigate this risk by using an externally-created dataset (Defects4J [22]). Since this dataset is created by others, it reduces experimenter bias. To reduce the likelihood of experimental errors, we have carefully checked our implementation to the best of our abilities.

Threats to external validity relate to the generalizability of our proposed approach. We have evaluated our approach on 357 real bugs from 5 Java programs. Many previous fault localization approaches are evaluated solely on synthetic bugs [5, 21, 36, 38, 43, 58] or on substantially fewer real bugs [31, 46, 61]. In the future, we plan to further mitigate this threat by evaluating *Savant* on more bugs from more programs written in various programming languages with diverse numbers of test cases.

Threats to construct validity relate to the suitability of our evaluation metrics. We make use of acc@n, wef@n, and MAP ($n \in \{1,3,5\}$). *acc@n* is based on findings by Parnin and Orso [41] which recommend the use of absolute ranks rather than percentages of program inspected. *wef@n* and MAP are widely-used metrics in fault localization [3, 18, 58] and various ranking techniques [26, 45, 52, 59, 64], respectively.

6. RELATED WORK

Spectrum-Based Fault Localization. There have been a wide variety of formulae proposed for computing the suspiciousness scores of program elements based on passing and failing test cases. Tarantula is an early, foundational SBFL technique that computes suspiciousness scores [21] based on the intuition that program elements often executed by passing test cases but never or rarely executed by failing test cases are less likely to be buggy. Subsequent researchers have proposed and evaluated alternative, more accurate formulae, such as Ochiai, proposed by Abreu et al. [4].

We describe and compare to a number of formulae that have been recently demonstrated as either theoretically optimal or high-performing. Xie et al. [56] theoretically compared many different suspiciousness scores, finding that for-

mulae belonging to two families outperform the others, including Naish's formula [38]. They also find that four of the scoring functions proposed by Yoo [60] derived using genetic programming (GP), are optimal. Xuan and Monperrus [58] proposed Multric, which combines multiple fault localization techniques together using a learning-to-rank algorithm. Lucia et al. use 40 association measures in data mining and statistics for fault localization [36]. However, Lucia et al.'s association measures only slightly improve over Ochiai [5]; Multric's improvement is by a much larger margin. We have compared our approach against Multric [58], formulae belonging to the best families identified by Xie et al. [56], and the best performing formulae produced by genetic programming [60]. Our experiments demonstrate that *Savant* outperforms these state-of-the-art approaches by a substantial margin.

Pytlik et al. [43] were the first to propose the use of likely invariants for fault localization, as we discussed in Section 2. Our approach uses a larger set of invariants, and uses them as features in a ranking algorithm. Our comparison to an extended version of Pytlik et al.'s approach showed that our approach achieves better performance on our dataset.

Sahoo et al. [46] extend Pytlik et al.'s approach by adding test case generation and backward slicing to reduce the number of program elements to inspect, and modify parts of failing test cases to create new successful tests. Otherwise, it requires specifications that describe the test cases. Sahoo et al. evaluated their approach by localizing 6 real faults in MySQL, 1 real fault in Squid, and 1 real fault in Apache 2.2. Our work differs from theirs in several respects. First, instead of *filtering* invariants, *Savant* uses them as features to rank program elements. It is possible to combine the filtering with the ranking approach in future work. Second, *Savant* avoids several restrictions and limitations of the previous work, in that it is not limited to programs with test oracles, programs where character-level rewriting of test cases makes sense, nor programs with test case specifications. Finally, our evaluation is substantially larger. We unfortunately cannot evaluate Sahoo et al.'s approach on our dataset since we do lack test oracles (unless we manually create them for all failing test cases), character-level deletion of test cases do not make sense for most of the programs (except for some bugs from Closure Compiler), and the test cases do not come with specifications. Moreover, Sahoo et al.'s approach is demonstrated on C rather than Java programs.

Le et al. [26] recently proposed to combine spectrum-based fault localization with information retrieval based fault localization. Information retrieval based fault localization, e.g., [64, 59, 23], takes as input a description of a bug and returns a set of program elements with similar contents as the words that appear in the bug description. Our work is complementary with theirs: While their approach focuses on combining two families of techniques, our approach focuses on improving one of the families. It is possible to replace the spectrum-based fault localization component of this approach with *Savant*, an option we leave to future work.

Mining Likely Invariants and Its Usages. Many approaches have been proposed to mine likely invariants in various formats [11, 12, 16, 25, 27, 34, 51]. We use Daikon [16], the most popular invariant mining tool. We describe it in Section 2.

We highlight a selection of the many approaches that use mined invariants, often inferred using Daikon, for various purposes. Baliga et al. [9] infer data structure invariants to prevent kernel-level rootkits from maliciously modifying key data structures. Schuler et al. [47] use likely invariants to assess the impact of mutants, demonstrating that mutants that violate likely invariants are less likely to be equivalent mutants. Abreu et al. use two kinds of program invariants to create a test oracle which can be used to label test cases as passed or failed [2]. This approach – as well as existing approaches on test oracle generation (c.f., [10]) – can be used in conjunction with an automated test generation technique to create new labelled test cases to enrich the input of an SBFL technique, including ours.

7. CONCLUSION AND FUTURE WORK

Debugging is a time consuming process. To help developers debug, many spectrum-based fault localization (SBFL) techniques have been proposed. The accuracy of these techniques is not yet optimal, and thus additional research is needed to advance the field. In this work, we extend the frontiers of research in SBFL, by proposing *Savant*, an approach that uses Daikon invariants to construct a rich set of features to localize buggy program elements using a learning-to-rank algorithm. We have evaluated our solution on a set of 357 bugs from 5 programs in the Defects4J benchmark. Our evaluation demonstrates that *Savant* can successfully localize 63.03, 101.72, 122 bugs on average within the top 1, top 3, and top 5 listed methods, respectively. We have compared *Savant* against 10 SBFL techniques that have been proven to outperform many other SBFL techniques, a hybrid SBFL technique that also uses learning-to-rank (Multric), and an extended version of an SBFL technique that also uses likely invariants (Carrot$^+$). *Savant* can locate 57.73%, 56.69%, and 43.13% more bugs at top 1, top 3, and top 5 methods as compared to the best performing baseline techniques.

In a future work, we plan to improve *Savant* further by selectively including a subset of invariants specialized for a target buggy program version and its spectra. We also plan to include a refinement process which incrementally adds or removes invariants to produce a better ranked list of methods. Furthermore, we plan to extend our evaluation to include more bugs beyond those in the Defects4J benchmark and compare *Savant* against other fault localization approaches. We also plan to investigate the impact of number of passed and failed test cases as well as other factors on the effectiveness of *Savant*.

Dataset and Tool Release. Savant's dataset and implementation are publicly available at https://goo.gl/PBCqFX.

8. ACKNOWLEDGMENTS

The authors gratefully acknowledge the partial support of AFRL, under AFRL Contract No. FA8750-15-2-0075 as well as the National Research Foundation, Prime Minister's Office, Singapore under its International Research Centres in Singapore Funding Initiative. Any views, opinions, findings or recommendations are those of the author(s) and do not necessarily reflect the views of the sponsoring agencies.

9. REFERENCES

[1] http://plse.cs.washington.edu/daikon/download/doc/daikon.html#Invariant-list.

[2] R. Abreu, A. González, P. Zoeteweij, and A. J. C. van Gemund. Automatic software fault localization using generic program invariants. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2008.

[3] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009.

[4] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 2009.

[5] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the Accuracy of Spectrum-based Fault Localization. In *TAICPART-MUTATION*, 2007.

[6] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse Technology eXchange, ETX 2005*, pages 35–39, 2005.

[7] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *ISSTA*, 2010.

[8] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical fault localization for dynamic web applications. In *Proceedings of the 32nd ACM/IEEE Int. Conference on Software Engineering*, ICSE '10, 2010.

[9] A. Baliga, V. Ganapathy, and L. Iftode. Detecting kernel-level rootkits using data structure invariants. *IEEE Trans. Dependable Sec. Comput.*, 8(5):670–684, 2011.

[10] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Trans. Software Eng.*, 41(5):507–525, 2015.

[11] I. Beschastnikh, J. Abrahamson, Y. Brun, and M. D. Ernst. Synoptic: studying logged behavior with inferred models. In *19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and 13rd European Software Engineering Conference (ESEC-13),2011*, pages 448–451, 2011.

[12] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Trans. Software Eng.*, 41(4):408–428, 2015.

[13] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[14] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, 2005.

[15] D. Developers. Daikon invariant list. plse.cs.washington.edu/daikon/download/doc/daikon.html#Invariant-list. Accessed: 2015-08-20.

[16] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99*, pages 213–224, 1999.

[17] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[18] A. González-Sanchez, R. Abreu, H. Gross, and A. J. C. van Gemund. Prioritizing tests for fault localization through ambiguity group reduction. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011.

[19] J. Han and M. Kamber. *Data Mining: Concepts and Techniques (2nd ed.)*. Morgan Kaufmann, 2006.

[20] D. J. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining*. MIT Press, 2001.

[21] J. Jones and M. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *20th IEEE/ACM international Conference on Automated software engineering*, 2005.

[22] R. Just, D. Jalali, and M. D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Int. Symposium on Software Testing and Analysis, ISSTA '14*, pages 437–440, 2014.

[23] D. Kim, Y. Tao, S. Kim, and A. Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE Transactions on Software Engineering*, 39(11):1597–1610, 2013.

[24] P. S. Kochhar, X. Xia, D. Lo, and S. Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 2016 International Symposium on Software Testing and Analysis*. ACM, 2016.

[25] I. Krka, Y. Brun, and N. Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *Proc. of the 22nd ACM SIGSOFT Int. Symposium on Foundations of Software Engineering, (FSE-22),*, pages 178–189, 2014.

[26] T. B. Le, R. J. Oentaryo, and D. Lo. Information retrieval and spectrum based bug localization: better together. In *Proc. of the 10th Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, pages 579–590, 2015.

[27] T. C. Le, S. Qin, and W. Chin. Termination and non-termination specification inference. In *Proc. of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015*, pages 489–498, 2015.

[28] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[29] C.-P. Lee and C.-b. Lin. Large-scale linear ranksvm. *Neural computation*, 26(4):781–817, 2014.

[30] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[31] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *ACM SIGPLAN Notices*, volume 40, pages 15–26. ACM, 2005.

[32] C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff. Sober: Statistical model-based bug localization. In *ESEC/FSE*, 2005.

[33] T.-Y. Liu. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009.

[34] D. Lo and S. Khoo. Smartic: towards building an accurate, robust and scalable specification miner. In *Proc. of the 14th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering, FSE 2006*, pages 265–275, 2006.

[35] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, 2016.

[36] Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process*, 26(2):172–219, 2014.

[37] Lucia, F. Thung, D. Lo, and L. Jiang. Are faults localizable? In *MSR*, pages 74–77, 2012.

[38] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):11, 2011.

[39] A. T. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *26th IEEE/ACM Int. Conference on Automated Software Engineering*, 2011.

[40] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE '13*, pages 772–781, 2013.

[41] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proc. of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011*, pages 199–209, 2011.

[42] D. Poshyvanyk, Y. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Software Eng.*, 33(6):420–432, 2007.

[43] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. *Proceedings of Workshop on Automated and Algorithmic Debugging*, 2003.

[44] S. Rao and A. C. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *MSR*, 2011.

[45] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *28th IEEE/ACM Int. Conference on Automated Software Engineering*, 2013.

[46] S. K. Sahoo, J. Criswell, C. Geigle, and V. S. Adve. Using likely invariants for automated software fault localization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2013.

[47] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA*, 2009.

[48] B. Sisman and A. C. Kak. Incorporating version histories in information retrieval based bug localization. In *MSR*, pages 50–59, 2012.

[49] F. Steimann, M. Frenkel, and R. Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 314–324. ACM, 2013.

[50] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology. Planning Report 02-3.2002*, 2002.

[51] M. Trinh, Q. L. Le, C. David, and W. Chin. Bi-abduction with pure properties for specification inference. In *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013*, pages 107–123, 2013.

[52] Q. Wang, C. Parnin, and A. Orso. Evaluating the usefulness of IR-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA*, pages 1–11, 2015.

[53] W. E. Wong, V. Debroy, and B. Choi. A family of code coverage-based heuristics for effective fault localization. *J. Syst. Softw.*, 83(2):188–208, Feb 2010.

[54] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. M. Thuraisingham. Effective software fault localization using an RBF neural network. *IEEE Transactions on Reliability*, 61(1):149–169, 2012.

[55] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai. Effective fault localization using code coverage. In *COMPSAC*, pages 449–456, 2007.

[56] X. Xie, T. Y. Chen, F. Kuo, and B. Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Trans. Softw. Eng. Methodol.*, 22(4):31, 2013.

[57] X. Xie, F. Kuo, T. Y. Chen, S. Yoo, and M. Harman. Provably optimal and human-competitive results in SBSE for spectrum based fault localisation. In *5th Int. Symposium on Search Based Software Engineering - SSBSE 2013, Proceedings*, pages 224–238, 2013.

[58] J. Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *30th IEEE Int. Conference on Software Maintenance and Evolution, 2014*, pages 191–200, 2014.

[59] X. Ye, R. C. Bunescu, and C. Liu. Learning to rank relevant files for bug reports using domain knowledge. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE*, 2014.

[60] S. Yoo. Evolving human competitive spectra-based fault localisation techniques. In *4th International Symposium Search Based Software Engineering SSBSE*, 2012.

[61] A. Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE*, 1999.

[62] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE*, pages 1–10, 2002.

[63] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transaction on Software Engineering*, 2002.

[64] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *ICSE*, pages 14–24, 2012.