

Mining Control Flow Abnormality for Logic Error Isolation*

Chao Liu Xifeng Yan Jiawei Han

Department of Computer Science
University of Illinois at Urbana-Champaign
{chaoliu, xyan, hanj}@cs.uiuc.edu

Abstract

Analyzing the executions of a buggy program is essentially a data mining process: Tracing the data generated during program executions may disclose important patterns and outliers that could eventually reveal the location of software errors. In this paper, we investigate program *logic errors*, which rarely incur memory access violations but generate incorrect outputs. We show that through mining program control flow abnormality, we could isolate many logic errors without knowing the program semantics.

In order to detect the control abnormality, we propose a hypothesis testing-like approach that statistically contrasts the evaluation probability of condition statements between correct and incorrect executions. Based on this contrast, we develop two algorithms that effectively rank functions with respect to their likelihood of containing the hidden error. We evaluated these two algorithms on a set of standard test programs, and the result clearly indicates their effectiveness.

Keywords. software errors, abnormality, ranking

1 Introduction

Recent years have witnessed a wide spread of data mining techniques into software engineering researches, such as in software specification extraction [2, 22, 19] and in software testings [5, 7, 23]. For example, frequent itemset mining algorithms are used by Livshits et al. to find, from software revision histories, programming patterns that programmers are expected to conform to [22]. A closed frequent itemset mining algorithm, CloSpan [27], is employed by Li et al. in boosting the performance of storage systems [17] and in isolating copy-paste program errors [18]. Besides frequent pattern-based approaches,

existing machine learning techniques, such as Support Vector Machines, decision trees, and logistic regression are also widely adopted for various software engineering tasks [5, 6, 7, 23, 21].

In this paper, we develop a new data mining algorithm that can assist programmers' manual debugging. Although programming language designs and software testings have greatly advanced in the past decade, software is still far from error (or bug, fault) free [8, 16]. As a rough estimation, there are usually 1-10 errors per thousand lines of code in *deployed* softwares [11]. Because debugging is notoriously time-consuming and laborious, we wonder whether data mining techniques can help speed up the process. As the initial exploration, this paper demonstrates the possibility of isolating logic errors through statistical inference.

The isolation task is challenging in that (1) statically, a program, even of only hundreds of lines of code, is a complex system because of the inherent intertwined data and control dependencies; (2) dynamically, the execution paths can vary greatly with different inputs, which further complicates the analysis; and (3) most importantly, since logic errors are usually tricky, the difference between incorrect and correct executions is not apparent at all. Therefore, it is almost like looking for a needle in a haystack to isolate logic errors. Due to these characteristics, isolating program errors could be a very important playground for data mining research.

From a data mining point of view, isolating logic errors is to discover suspicious buggy regions through screening bulky program execution traces. For example, our method first monitors the program runtime behaviors, and then tries to discover the region where the behavior of incorrect executions (or runs) diverges from that of correct ones. Different from conventional software engineering methods, this data mining approach assumes no prior knowledge of the program semantics, thus providing a higher level of automation.

Before detailing the concepts of our method, let us first take a brief overview of program errors. Based

*This work was supported in part by the U.S. National Science Foundation NSF ITR-03-25603, IIS-02-09199, and IIS-03-08215. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies.

on their faulty symptoms, we can roughly classify program errors into two categories: memory errors and logic errors. Memory errors usually manifest themselves through memory abnormalities, like segmentation faults and/or memory leaks. Some tools, like Purify, Valgrind and CCured, are designed for tracking down memory errors. Logic errors, on the other hand, refer to the program logic incorrectness. Instead of causing programs to abort due to memory violations, programs with logic errors usually exit silently without segmentation faults. The only faulty symptom is its malfunction, like generating none or incorrect outputs. Because logic errors do not behave abnormally at the memory access level, off-the-shelf memory monitoring tools, like Purify, have little chance to isolate them. Considering the plethora of tools for memory errors, we plan to focus on isolating logic errors in this paper. To illustrate the challenges, let us first examine an example.

Program 1 Buggy Code - Version 9 of replace

```

01 void dodash (char delim, char *src, int *i,
02             char *dest, int *j, int maxset)
03 {
04     while (...) {
05         ...
06         if ( isalnum(src[*i-1]) && isalnum(src[*i+1])
07             /* && (src[*i-1] <= src[*i+1]) */)
08             {
09                 for (k = src[*i-1]+1; k<=src[*i+1]; k++)
10                     junk = addstr(k, dest, j, maxset);
11                 *i = *i + 1;
12             }
13         ...
14         (*i) = (*i) + 1;
15     }
16 }

```

EXAMPLE 1. *Program 1 shows a code segment of replace, a program that performs regular expression matching and substitutions. The replace program has 512 lines of non-blank C code, and consists of 20 functions. In the dodash function, one subclause (commented by /* and */) that should have been included in the if condition is missed by the developer. This is a typical “incomplete logic” error. We tested this buggy program through 5542 test cases and found that 130 out of them failed to give the correct outputs. Moreover, no segmentation faults happened during executions.*

For logic errors as the above, without any memory violations, the developer generally has no clues for debugging. Probably, he will resort to conventional debuggers, like GDB, for a step-by-step tracing and

verify observed values against his expectations in mind. To make things worse, if the developer is not the original author of the program, which is very likely in reality, such as maintaining legacy codes, he may not even be able to trace the execution until he at least roughly understands the program. However, understanding other programmers’ code is notoriously troublesome because of un-intuitive identifiers, personalized coding styles and, most importantly, the complex control and data dependencies formulated by the original author.

Knowing the difficulty of debugging logic errors, we are interested in finding an automated method that can prioritize the code examination for debugging, e.g., to guide programmers to examine the buggy function first. Although it is unlikely to totally unload the debugging burden from human beings, we find it possible to minimize programmers’ workloads through machine intelligence. For instance, by correctly suggesting the possible buggy regions, one can narrow down the search, reduce the debugging cost, and speed up the development process. We follow this moderate thought in this study.

In order to facilitate the analysis of program executions, we need to first encode each execution in such a way that discriminates incorrect from correct executions. This is equivalent to a proper extraction of error-sensitive features from program executions. Because the faulty symptom of logic error is the unexpected execution path, it is tempting to register the *exact* path for *each* execution. However, this scheme turns out to be hard to manipulate, as well as expensive to register. Noticing that executions are mainly directed by condition statements (e.g., **if**, **while**, **for**), we finally decide to summarize the execution by the evaluation frequencies of conditionals. In other words, for each condition statement, we record how many times it is evaluated as **true** and **false** respectively in each execution. Although this coding scheme loses much information about the exact execution, it turns out to be easy to manipulate and effective.

Based on the above analysis, we treat each conditional in the buggy program as one distinct feature and try to isolate the logic error through contrasting the behaviors of correct and incorrect executions in terms of these conditionals. Specifically, we regard that *the more divergent the evaluation of one conditional in incorrect runs is from that of correct ones, the more likely the conditional is bug-relevant*. In order to see why this choice can be effective, let us go back to EXAMPLE 1.

For convenience, we declare two boolean variables A and B as follows,

```

A = isalnum(src[*i-1]) && isalnum(src[*i+1]);
B = src[*i-1]<= src[*i+1];

```

If the program were written correctly, $A \wedge B$ should have been the guarding condition for the `if` block, *i.e.*, the control flow goes into the `if` block if and only if $A \wedge B$ is true. However, in the buggy program where B is missing, any control flow that satisfies A will fall into the block. As one may notice and we will explain in Section 2, an execution is logically correct until $(A \wedge \neg B)$ is evaluated as `true` when the control flow reaches Line 6. Because the `true` evaluation of $(A \wedge \neg B)$ only happens in incorrect runs, and it contributes to the `true` evaluation of A , the evaluation distribution of the `if` statement should be different between correct and incorrect executions. This suggests that if we monitor the program conditionals, like the A here, their evaluations will shed light on the hidden error and can be exploited for error isolation. While this heuristic can be effective, one should not expect it to work with any logic errors. After all, even experienced programmers may feel incapable to debug certain logic errors. Given that few effective ways exist for isolating logic errors, we believe this heuristic worths a try.

The above example motivates us to infer about the potential buggy region from the divergent branching probability. Heuristically, the simplest approach is to first calculate the average `true` evaluation probabilities for both correct and incorrect runs, and treat the numeric difference as the measure of divergence. However, simple heuristic methods like the above generally suffer from weak performance across various buggy programs. In this paper, we develop a novel and more sophisticated approach to quantifying the divergence of branching probabilities, which features a similar rationale to hypothesis testing [15]. Based on this quantification, we further derive two algorithms that effectively rank functions according to their likelihood of containing the error. For example, both algorithms recognize the `codash` function as the most suspicious in EXAMPLE 1.

In summary, we make the following contributions.

1. We introduce the problem of isolating logic errors, a problem not yet well solved in software engineering community, into data mining research. We believe that recent advances in data mining would help tackle this tough problem and contribute to software engineering in general.
2. We propose an error isolation technique that is based on mining control flow abnormality in incorrect runs against correct ones. Especially, we choose to monitor only conditionals to bear low overhead while maintaining the isolation quality.
3. We develop a principled statistical approach to quantify the bug relevance of each condition state-

ment and further derive two algorithms to locate the possible buggy functions. As evaluated on a set of standard test programs, both of them achieve an encouraging success.

The remaining of the paper is organized as follows. Section 2 first provides several examples that illustrate how our method is motivated. The statistical model and the two derived ranking algorithms are developed in Section 3. We present the analysis of experiment results in Section 4. After the discussion about related work in Section 5, Section 6 concludes this study.

2 Motivating Examples

In this section, we re-visit EXAMPLE 1 and explain in detail the implication of control flow abnormality to logic errors. For clarity in what follows, we denote the program with the subclause `(src[*i-1] <= src[*i+1])` commented out as the incorrect (or buggy) program \mathcal{P} , and the one without comments is the correct one, denoted as $\hat{\mathcal{P}}$. Because $\hat{\mathcal{P}}$ is certainly not available when debugging \mathcal{P} , $\hat{\mathcal{P}}$ is used here only for illustration purposes: It helps illustrate how our method is motivated. As one will see in Section 3, our method collects statistics only from the buggy program \mathcal{P} and performs all the analysis.

Given the two boolean variables A and B as presented in Section 1, let us consider their evaluation combinations and corresponding branching actions (either `enter` or `skip` the block) in both \mathcal{P} and $\hat{\mathcal{P}}$. Figure 1 explicitly lists the actions in \mathcal{P} (left) and $\hat{\mathcal{P}}$ (right), respectively. Clearly, the left panel shows the actual actions taken in the buggy program \mathcal{P} , and the right one lists the expected actions if \mathcal{P} had no errors.

	A	$\neg A$		A	$\neg A$
B	<i>enter</i>	<i>skip</i>	B	<i>enter</i>	<i>skip</i>
$\neg B$	<i>enter</i>	<i>skip</i>	$\neg B$	<i>skip</i>	<i>skip</i>

Figure 1: Branching Actions in \mathcal{P} and $\hat{\mathcal{P}}$

Differences between the above two tables reveal that in the buggy program \mathcal{P} , unexpected actions take place if and only if $A \wedge \neg B$ evaluates to `true`. Explicitly, when $A \wedge \neg B$ is `true`, the control flow actually enters the block, while it is expected to skip. This incorrect control flow will eventually lead to incorrect outputs. Therefore, for the buggy program \mathcal{P} , one run is incorrect if and only if there exist `true` evaluations of $A \wedge \neg B$ at Line 6; otherwise, the execution is correct although the program contains a bug.

While the boolean expression $\mathcal{B}: (A \wedge \neg B) = \text{true}$ exactly characterizes the scenario under which incorrect

executions take place, there is little chance for an automated tool to spot B as bug relevant. The obvious reason is that while we are debugging the program \mathcal{P} , we have no idea of what B is, let alone its combination with A . Because A is observable in \mathcal{P} , we are therefore interested in whether the evaluation of A will give away the error. If the evaluation of A in incorrect executions significantly diverge from that in correct ones, the `if` statement at line 6 may be regarded as bug-relevant, which points to the exact error location.

	A	$\neg A$
B	n_{AB}	$n_{\bar{A}B}$
$\neg B$	$n_{A\bar{B}} = 0$	$n_{\bar{A}\bar{B}}$

	A	$\neg A$
B	n'_{AB}	$n'_{\bar{A}B}$
$\neg B$	$n'_{A\bar{B}} \geq 1$	$n'_{\bar{A}\bar{B}}$

Figure 2: A Correct and Incorrect Run in \mathcal{P}

We therefore contrast how A is evaluated differently between correct and incorrect executions of \mathcal{P} . Figure 2 shows the number of `true` evaluations for the four combinations of A and B in one correct (left) and incorrect (right) run. The major difference is that in the correct run, $A \wedge \neg B$ never evaluates `true` ($n_{A\bar{B}} = 0$) while $n'_{A\bar{B}}$ must be nonzero for one execution to be incorrect. Since the `true` evaluation of $A \wedge \neg B$ implies $A = \text{true}$, we therefore expect that the probability for A to be `true` is different between correct and incorrect executions. As we tested through 5,542 test cases, the `true` evaluation probability is 0.727 in a correct and 0.896 in an incorrect execution on average. This divergence suggests that the error location (i.e., Line 6) does exhibit detectable abnormal behaviors in incorrect executions. Our method, as developed in Section 3, nicely captures this divergence and ranks the `dotdash` function as the most suspicious function, isolating this logic error.

The above discussion illustrates a simple, but motivating example where we can use the branching statistics to capture the control flow abnormality. Sometimes when the error does not occur in a condition statement, control flows may still disclose the error trace. In Program 2, the programmer mistakenly assigns the variable `result` with `i+1` instead of `i`, which is a typical “off-by-one” logic error.

The error in Program 2 literally has no relation with any condition statements. However, the error is triggered only when the variable `junk` is nonzero, which is eventually associated with the `if` statement at Line 5. In this case, although the branching statement is not directly involved in this off-by-one error, the abnormal branching probability in incorrect runs can still reveal the error trace. For example, the `makepat` function is identified as the most suspicious function by our

Program 2 Buggy Code - Version 15 of `replace`

```

01 void makepat (char *arg, int start, char delim,
02               char *pat)
03 {
04     ...
05     if(!junk)
06         result = 0;
07     else
08         result = i + 1; /* off-by-one error */
09         /* should be result = i */
10     return result;
11 }

```

algorithms.

Inspired by the above two examples, we recognize that given that the dependency structure of a program is hard to untangle, there do exist simple statistics that capture the abnormality caused by hidden errors. Discovering these abnormalities by mining the program executions can provide useful information for error isolation. In the next section, we elaborate on the statistical ranking model that identifies potential buggy regions.

3 Ranking Model

Let $T = \{t_1, t_2, \dots, t_n\}$ be a test suite for a program \mathcal{P} . Each test case $t_i = (d_i, o_i)$ ($1 \leq i \leq n$) has an input d_i and the desired output o_i . We execute \mathcal{P} on each t_i , and obtain the output $o'_i = \mathcal{P}(d_i)$. We say \mathcal{P} *passes* the test case t_i (i.e., “a passing case”) if and only if o'_i is identical to o_i ; otherwise, \mathcal{P} *fails* on t_i (i.e., “a failing case”). We thus partition the test suite T into two disjoint subsets T_p and T_f , corresponding to the passing and failing cases respectively,

$$T_p = \{t_i | o'_i = \mathcal{P}(d_i) \text{ matches } o_i\},$$

$$T_f = \{t_i | o'_i = \mathcal{P}(d_i) \text{ does not match } o_i\}.$$

Since program \mathcal{P} passes test case t_i if and only if \mathcal{P} executes correctly, we use “correct” and “passing”, “incorrect” and “failing” interchangeably in the following discussion.

Given a buggy program \mathcal{P} together with a test suite $T = T_p \cup T_f$, our task is to *isolate the suspicious error region by contrasting \mathcal{P} 's runtime behaviors on T_p and T_f* .

3.1 Feature Preparation

Motivated by previous examples, we decide to instrument each condition statement in program \mathcal{P} to collect the evaluation frequencies at runtime. Specifically, we take the entire boolean expression in each condition

statement as one distinct *boolean feature*. For example, `isalnum(src[*i-1]) && isalnum(src[*i+1])` in Program 1 is treated as *one* feature. Since a feature can be evaluated zero to multiple times as either `true` or `false` in each execution, we define *boolean bias* to summarize its exact evaluations.

DEFINITION 1. (BOOLEAN BIAS) *Let n_t be the number of times that a boolean feature \mathcal{B} evaluates `true`, and n_f the number of times it evaluates `false` in one execution. $\pi(\mathcal{B}) = \frac{n_t - n_f}{n_t + n_f}$ is the boolean bias of \mathcal{B} in this execution.*

$\pi(\mathcal{B})$ varies in the range of $[-1, 1]$. It encodes the distribution of \mathcal{B} 's value: $\pi(\mathcal{B})$ is equal to 1 if \mathcal{B} always assumes `true` and -1 as it sticks to `false`, and in between for all other mixtures. If a conditional is never touched during an execution, $\pi(\mathcal{B})$ is defined to be 0 because of no evidence favoring either `true` or `false`.

3.2 Methodology Overview

Before detailed discussions about our method, we first lay out the main idea in this subsection. Following the convention of statistics, we use uppercase letters for random variables and lowercases for their realizations. Moreover, $f(X|\theta)$ is the probability density function (pdf) of a distribution (or population) family indexed by the parameter θ .

Let the entire test case space be \mathcal{T} , which conceptually contains all possible input and output pairs. According to the correctness of \mathcal{P} on cases from \mathcal{T} , \mathcal{T} can be partitioned into two disjoint sets \mathcal{T}_p and \mathcal{T}_f for passing and failing cases. Therefore, \mathcal{T} , \mathcal{T}_p , and \mathcal{T}_f can be thought as random samples from \mathcal{T} , \mathcal{T}_p , and \mathcal{T}_f respectively. Let X be the random variable for the boolean bias of boolean feature \mathcal{B} , we use $f(X|\theta_p)$ and $f(X|\theta_f)$ to denote underlying probability model that generates the boolean bias of \mathcal{B} for cases from \mathcal{T}_p and \mathcal{T}_f respectively.

DEFINITION 2. (BUG RELEVANCE) *A boolean feature \mathcal{B} is relevant to the program error if its underlying probability model $f(X|\theta_f)$ diverges from $f(X|\theta_p)$.*

The above definition relates the probability model $f(X|\theta)$ with the hidden program error. The more significantly $f(X|\theta_f)$ diverges from $f(X|\theta_p)$, the more relevant \mathcal{B} is to the hidden error. Let $\mathbf{L}(\mathcal{B})$ be a similarity function,

$$(3.1) \quad \mathbf{L}(\mathcal{B}) = \mathbf{Sim}(f(X|\theta_p), f(X|\theta_f)),$$

and $s(\mathcal{B})$, the bug relevance score of \mathcal{B} can be defined as

$$(3.2) \quad s(\mathcal{B}) = -\log(\mathbf{L}(\mathcal{B})).$$

Using the above measure, we can rank features according to their relevance to the error. The ranking problem boils down to finding a proper way to quantify the similarity function. This includes two problems: (1) what is a suitable similarity function? and (2) how to compute it while we do not know the closed form of $f(X|\theta_p)$ and $f(X|\theta_f)$? In the following subsections, we will examine these two problems in detail.

3.3 Feature Ranking

Because no prior knowledge of the closed form of $f(X|\theta_p)$ is known, we can only characterize it through general parameters, such as the population mean and variance, $\mu_p = E(X|\theta_p)$ and $\sigma_p^2 = Var(X|\theta_p)$, i.e., we take $\theta = (\mu, \sigma^2)$. While μ and σ^2 are taken to characterize $f(X|\theta)$, we do not regard their estimates as sufficient statistics, and hence we do not take normality assumptions on $f(X|\theta)$. Instead, the difference exhibited through μ and σ^2 is treated as a measure of the model difference.

Given an *independent and identically distributed (i.i.d.)* random sample $\mathbf{X} = (X_1, X_2, \dots, X_n)$ from $f(X|\theta_p)$, μ_p and σ_p^2 can be estimated by the *sample mean* \bar{X} and *sample variance* S_n ,

$$\mu_p = \bar{X} = \frac{\sum_{i=1}^n X_i}{n}$$

and

$$\sigma_p^2 = S_n = \frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2,$$

where X_i represents the boolean bias of \mathcal{B} from the i th passing case. The population mean μ_f and variance σ_f^2 of $f(X|\theta_f)$ can be estimated in a similar way.

Given the estimations, it may be appealing to take the differences of both mean and variance as the bug relevance score. For example, $s(\mathcal{B})$ can be defined as

$$s(\mathcal{B}) = \alpha * |\mu_p - \mu_f| + \beta * |\sigma_p^2 - \sigma_f^2| \quad (\alpha, \beta \geq 0).$$

However, heuristic methods like the above usually suffer from the dilemma of parameter settings: no guidance is available to properly set the parameters, like the α and β in the above formula. In addition, parameters that work perfectly with one program error may not generalize to other errors or programs. Therefore, in the following, we develop a principled statistical method to quantify the bug relevance, which, as one will see, is parameter-free.

Our method takes an indirect approach to quantifying the divergence between $f(X|\theta_p)$ and $f(X|\theta_f)$, which is supported by a similar rationale to hypothesis testing [15]. Instead of writing a formula that explicitly specifies the difference, we first propose the *null hypothesis*

\mathcal{H}_0 that $\theta_p = \theta_f$ (i.e., no divergence exists), and then under the null hypothesis, we derive a statistic $Y(\mathbf{X})$ that conforms to a specific known distribution. Given the realized random sample $\mathbf{X} = \mathbf{x}$, if $Y(\mathbf{x})$ corresponds to a small probability event, the null hypothesis \mathcal{H}_0 is invalidated, which immediately suggests that $f(X|\theta_p)$ and $f(X|\theta_f)$ are divergent. Moreover, the divergence is proportional to the extent to which the null hypothesis is invalidated.

Specifically, the null hypothesis \mathcal{H}_0 is

$$(3.3) \quad \mu_p = \mu_f \text{ and } \sigma_p = \sigma_f.$$

Let $\mathbf{X} = (X_1, X_2, \dots, X_m)$ be an *i.i.d.* random sample from $f(X|\theta_f)$. Under the null hypothesis, we have $E(X_i) = \mu_f = \mu_p$ and $Var(X_i) = \sigma_f^2 = \sigma_p^2$. Because $X_i \in [-1, 1]$, $E(X_i)$ and $Var(X_i)$ are both finite. According to the Central Limit Theorem, the following statistic

$$(3.4) \quad Y = \frac{\sum_{i=1}^m X_i}{m},$$

converges to the normal distribution $N(\mu_p, \frac{\sigma_p^2}{m})$ as $m \rightarrow +\infty$.

Let $f(Y|\theta_p)$ be the pdf of $N(\mu_p, \frac{\sigma_p^2}{m})$, then the likelihood of θ_p given the observation of Y is

$$(3.5) \quad L(\theta_p|Y) = f(Y|\theta_p).$$

A smaller likelihood implies that \mathcal{H}_0 is less likely to hold and this, in consequence, indicates that a larger divergence exists between $f(X|\theta_p)$ and $f(X|\theta_f)$. Therefore, we can reasonably set the similarity function in Eq. (3.1) as the likelihood function,

$$(3.6) \quad \mathbf{L}(\mathcal{B}) = L(\theta_p|Y).$$

According to the property of normal distribution, the normalized statistic

$$Z = \frac{Y - \mu_p}{\sigma_p / \sqrt{m}}$$

conforms to the standard normal distribution $N(0, 1)$, and

$$(3.7) \quad f(Y|\theta_p) = \frac{\sqrt{m}}{\sigma_p} \varphi(Z),$$

where $\varphi(Z)$ is the pdf of $N(0, 1)$,

Combining Eq. (3.2), (3.6), (3.5), and (3.7), we finally have the bug relevance score for boolean feature \mathcal{B} as

$$(3.8) \quad s(\mathcal{B}) = -\log(\mathbf{L}(\mathcal{B})) = \log\left(\frac{\sigma_p}{\sqrt{m}\varphi(Z)}\right).$$

According to Eq. (3.8), we can rank all condition statements of the buggy function [20]. However, we regard that a ranking of suspicious functions is preferable to that of individual statements because a highly relevant condition statement is not necessarily the error root. For example, the error in Program 2 does not take place in the “if(!junk)” statement. Moreover, we generally have higher confidence in the quality of function ranking than that of individual statements in that function abnormality is assessed by considering all of its component features. In the following section, we discuss how to combine individual $s(\mathcal{B})$ to a global score $s(\mathcal{F})$ for each function \mathcal{F} .

3.4 Function Ranking

Suppose a function \mathcal{F} encompasses k boolean features $\mathcal{B}_1, \dots, \mathcal{B}_k$, and there are m failing test cases. Let X_{ij} denote the boolean bias of the i^{th} boolean feature in the j^{th} test case, we tabulate the statistics in the following table.

	t_1	t_2	\dots	t_m			
\mathcal{B}_1	X_{11}	X_{12}	\dots	X_{1m}	\mathbf{X}_1	Y_1	Z_1
\mathcal{B}_2	X_{21}	X_{22}	\dots	X_{2m}	\mathbf{X}_2	Y_2	Z_2
\dots	\dots	\dots	\dots	\dots	\dots	\dots	\dots
\mathcal{B}_k	X_{k1}	X_{k2}	\dots	X_{km}	\mathbf{X}_k	Y_k	Z_k

In this table, $\mathbf{X}_i = (X_{i1}, X_{i2}, \dots, X_{im})$ represents the observed boolean bias from the m failing runs and Y_i and Z_i are similarly derived from \mathbf{X}_i as in Section 3.3. For each feature \mathcal{B}_i , we propose the null hypothesis $\mathcal{H}_0^i : \theta_f^i = \theta_p^i$, and obtain

$$(3.9) \quad f(Y_i|\theta_p^i) = \frac{\sqrt{m}}{\sigma_p^i} \varphi(Z_i).$$

Given that the function \mathcal{F} has k features, we denote the parameters in a vector form:

$$\begin{aligned} \vec{\mathcal{H}}_0 &= \langle \mathcal{H}_0^1, \mathcal{H}_0^2, \dots, \mathcal{H}_0^k \rangle, \\ \vec{\theta}_p &= \langle \theta_p^1, \theta_p^2, \dots, \theta_p^k \rangle, \\ \vec{Y} &= \langle Y_1, Y_2, \dots, Y_k \rangle. \end{aligned}$$

Under the null hypothesis $\vec{\mathcal{H}}_0$, similar arguments suggest that the bug relevance score $s(\mathcal{F})$ can be chosen as

$$(3.10) \quad s(\mathcal{F}) = -\log(f(\vec{Y}|\vec{\theta}_p)),$$

where $f(\vec{Y}|\vec{\theta}_p)$ is the joint pdf of Y_i 's ($i = 1, 2, \dots, k$).

However, the above scoring function Eq. (3.10) does not immediately apply because $f(\vec{Y}|\vec{\theta}_p)$ is a multivariate density function. Because neither the closed forms

of $f(Y_i|\theta_p^i)$ nor the dependencies among Y_i 's are available, it is impossible to calculate the exact value of $s(\mathcal{F})$. Therefore, in the following discussion, we propose two simple ways to approximate it.

3.4.1 CombineRank

One conventional approach to untangling joint distribution is through the independence assumption. If we assume that boolean features \mathcal{B}_i 's are *mutually independent*, the population of Y_i is also mutually independent with each other. Therefore, we have

$$(3.11) \quad f(\vec{Y}|\vec{\theta}_p) = \prod_{i=1}^k f(Y_i|\theta_p^i) = \prod_{i=1}^k \frac{\sqrt{m}}{\sigma_p^i} \varphi(Z_i)$$

Following Eq. (3.10),

$$(3.12) \quad s(\mathcal{F}) = -\log(f(\vec{Y}|\vec{\theta}_p)) = \sum_{i=1}^k \log\left(\frac{\sigma_p^i}{\sqrt{m}\varphi(Z_i)}\right) \\ = \sum_{i=1}^k s(\mathcal{B}_i)$$

We name the above scoring schema (Eq. (3.12)) COMBINERANK as it sums over the bug relevance score of each individual condition statement with the function. We note that the independence assumption is practically unrealistic because condition statements are usually intertwined, such as nested loops or the `if` statements inside `while` loops. However, given that no assumptions are made about the probability densities, one should not expect for a devices more magic than the independence assumption in decomposing the joint distribution.

From the other point of view, COMBINERANK does make good sense in that the abnormal branchings at one condition statement may likely trigger the abnormal executions of other branches (like an avalanche) and Eq. (3.12) just captures this systematical abnormality and encode it into the final bug relevance score for the function \mathcal{F} . As we will see in the experiments, COMBINERANK works really well in locating buggy functions.

3.4.2 UpperRank

In this subsection, we propose another approach to approximating $f(\vec{Y}|\vec{\theta}_p)$, and end up with another score schema UPPERANK. The main idea is based on the following apparent inequality

$$(3.13) \quad f(\vec{Y}|\vec{\theta}_p) \leq \min_{1 \leq i \leq k} f(Y_i|\theta_p^i) = \min_{1 \leq i \leq k} \frac{\sqrt{m}}{\sigma_p^i} \varphi(Z_i).$$

Therefore, if we adopt the upper bound as an alternative for $f(\vec{Y}|\vec{\theta}_p)$, $s(\mathcal{F})$ can be derived as

$$(3.14) \quad s(\mathcal{F}) = -\log\left(\min_{1 \leq i \leq k} \frac{\sqrt{m}}{\sigma_p^i} \varphi(Z_i)\right) = \max_{1 \leq i \leq k} s(\mathcal{B}_i).$$

This scoring schema, named UPPERANK, essentially picks the most suspicious feature as the “representative” of the function. This is meaningful because if one boolean expression is extremely abnormal, the function containing it is very likely to be abnormal. However, since UPPERANK uses the upper bound as the approximation to $f(\vec{Y}|\vec{\theta}_p)$, its ranking quality is inferior to that of COMBINERANK when multiple peak abnormalities exist, as illustrated in the following example.

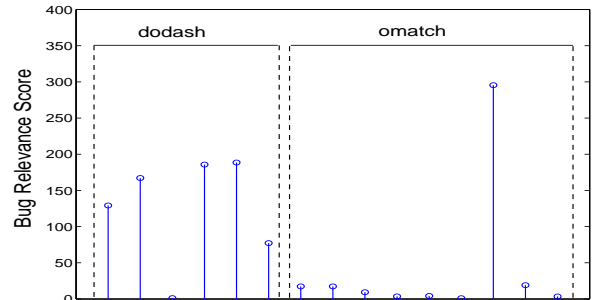


Figure 3: COMBINERANK vs. UPPERANK

EXAMPLE 2. Figure 3 visualizes the bug relevance score of boolean features in both function `dodash` and `omatch`, calculated on the faulty Version 5 of the `replace` program. From left to right, the first six stubs represent scores for the six boolean features in `dodash` and the following nine are for features in `omatch`. In this example, the logic error is inside `dodash`. As one can see, UPPERANK will rank `omatch` over `dodash` due to the maximal peak abnormality. However, it might be better to credit the abnormality of each feature for function ranking, as implemented by COMBINERANK. Therefore, COMBINERANK correctly ranks `dodash` as the most bug relevant. For this reason, we generally prefer COMBINERANK to UPPERANK, as is also supported by experiments.

4 Experiment Results

In this section, we evaluate the effectiveness of both COMBINERANK and UPPERANK in isolating logic errors. We implemented these two algorithms using C++ and Matlab and conducted the experiments on a 3.2GHz Intel Pentium 4 PC with 1GB physical memory that runs Fedora Core 2.

Version	Buggy Function	Fail Runs	COMBINERANK	UPPERRANK	Error Description: How to Fix	Cat.
1	dodash	68	2	1	change <code>*i</code> to <code>*i -1</code>	⊙
2	dodash	37	1	1	add one <code>if</code> branch	★
3	subline	130	3	3	add one <code>&&</code> subclause	★
4	subline	143	8	11	change <code>i</code> to <code>lastm</code>	△
5	dodash	271	1	3	change <code><</code> to <code><=</code>	★
6	locate	96	3	3	change <code>>=</code> to <code>></code>	★
7	in_set_2	83	5	4	change <code>c == ANY</code> to <code>c == EOL</code>	△
8	in_set_2	54	3	2	add one <code> </code> subclause	★
9	dodash	30	1	1	add one <code>&&</code> subclause	★
10	dodash	23	1	2	add one <code>&&</code> subclause	★
11	dodash	30	1	1	change <code>></code> to <code><=</code>	★
12	Macro	309	5	5	change 50 to 100 in <code>define MAXPAT 50</code>	△
13	subline	175	5	5	add one <code>if</code> branch	★
14	omatch	137	1	1	add one <code>&&</code> subclause	★
15	makepat	60	1	1	change <code>i+1</code> to <code>i</code> in <code>result = i+1</code>	⊙
16	in_set_2	83	5	4	remove one <code> </code> subclause	★
17	esc	24	1	1	change <code>result = NEWLINE</code> to <code>= ESCAPE</code>	△
18	omatch	210	2	3	add one <code>&&</code> subclause	★
19	change	3	15	8	rewrite the function <code>change</code>	◇
20	esc	22	1	1	change <code>result = ENDSTR</code> to <code>= ESCAPE</code>	△
21	getline	3	12	5	rewrite the function <code>getline</code>	◇
22	getccl	19	7	3	move one statement into <code>if</code> branch	◇
23	esc	22	1	2	change <code>s[*i]</code> to <code>s[*i + 1]</code>	⊙
24	omatch	170	2	4	add one <code>if</code> branch	★
25	omatch	3	2	2	change <code><=</code> to <code>==</code>	★
26	omatch	198	6	6	change <code>j</code> to <code>j + 1</code>	⊙
28	in_set_2	142	4	3	remove one <code> </code> subclause	★
29	in_set_2	64	6	5	remove one <code> </code> subclause	★
30	in_set_2	284	1	1	remove one <code> </code> subclause	★
31	omatch	210	2	3	change <code>>=</code> to <code>!=</code>	★
Error Category Legend		★: Incorrect Branch Expression	△: Misuse of Variables or Constants	⊙: Off-By-One	◇: Misc.	

Table 1: Summary of Buggy Versions and Ranking Results

4.1 Subject Programs

We experimented on a package of standard test programs, called *Siemens programs*¹. This package was originally prepared by Siemens Corp. Research in study of test adequacy criteria [12]. The Siemens programs consist of seven programs: `print_tokens`, `print_tokens2`, `replace`, `schedule`, `schedule2`, `tcas`, and `tot_info`. For each program, the Siemens researchers manually injected multiple errors, obtaining multiple faulty versions, with each version containing one and only one error. Because these injected errors rightly represent common mistakes made in practice, the Siemens programs are widely adopted as a benchmark in software engineering

research [12, 25, 9, 10]. Because these injected errors are mainly logic faults, we choose them as the benchmark to evaluate our algorithms.

Except `tcas` (141 lines), the size of these programs ranges from 292 to 512 lines of C code, excluding blanks. Because debugging `tcas` is pretty straightforward due to its small size, we focus the experiment study on the other six programs. In Section 4.2, we first analyze the effectiveness of our algorithms on the 30 faulty versions of the `replace` program. The `replace` program deserves detailed examination in that (1) it is the largest and most complex one among the six programs, and (2) it covers the most varieties of logic errors. After the examination of `replace`, we discuss about the experiments on the other five programs in Section 4.3.

¹A variant is available at <http://www.cc.gatech.edu/aristotle/Tools/subjects>.

4.2 Experimental Study on Replace

The `replace` program contains 32 versions in total, among which Version 0 is error free. Each of the other 31 faulty versions contains one and only one error in comparison with Version 0. In this setting, Version 0 serves as the oracle in labelling whether a particular execution is correct or incorrect.

Table 1 lists the error characteristics for each faulty version and the ranking results provided by COMBINERANK and UPPERANK. Because we focus on isolating logic errors that do not incur segmentation faults, Version 27 is excluded from examination. In Table 1, the second column lists the name of the buggy function for each version and the third column shows the number of failing runs out of the 5542 test cases. The final ranks of the buggy function provided by COMBINERANK and UPPERANK are presented in the fourth and fifth columns respectively. Taking version 1 for an example, the buggy function is `dodash` and the error causes incorrect outputs for 68 out of the 5542 test cases. For this case, COMBINERANK ranks the `dodash` function at the second place and UPPERANK identifies it as the most suspicious. Finally, the last two columns briefly describe each error and the category the error belongs to (Section 4.2.2). Because we cannot list the buggy code for each error due to the space limit, we concisely describe how to fix each error in the “Error Description” column. Interested readers are encouraged to download “Siemens Programs” from the public domain for detailed examination.

4.2.1 Overall Effectiveness

The rankings in Table 1 indicate that both COMBINERANK and UPPERANK work very well with the `replace` program. Among the 30 faulty versions under examination, both methods rank the buggy function within the top five for 24 versions except Versions 4, 19, 21, 22, 26, and 29. This implies that the developer can locate 24 out of the 30 errors if he only examines the top five functions of the ranked list. Although this proposition is drawn across *various* errors on a *specific* subject program, we believe that this does shed light on the effectiveness of the isolation algorithms.

In order to quantify the *isolation quality* of the algorithms, we choose to measure how many errors are located if a programmer examines the suggested ranked list from the top down. For example, because COMBINERANK ranks the buggy function at the first place for 11 faulty versions, the programmer will locate 11 out of the 30 errors if he only examines the first function of the ranked list for each faulty version. Moreover,

because the buggy function of another five versions is ranked at the second place by COMBINERANK, the programmer can locate 16 out of the 30 errors if he takes the top-2 functions seriously. In this way, Figure 4 plots the number of located errors with respect to the top- k examination.

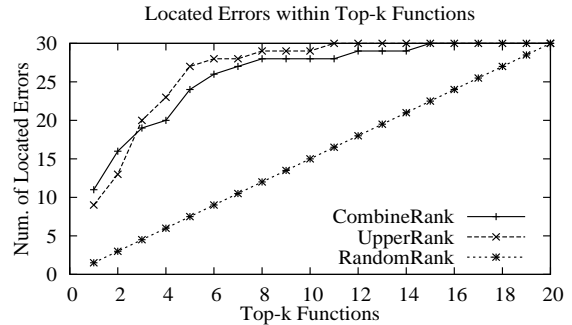


Figure 4: Isolation Quality: `replace`

In order to assess the effectiveness of the algorithms, we also plot the curve for random rankings in Figure 4. Because a random ranking puts the buggy function at any place with equal probability, the buggy function has a probability of $\frac{k}{m}$ to be within the top- k of the entire m functions. Furthermore, this also suggests that given n faulty versions, $\frac{k}{m}n$ errors are expected to be located if the programmer only examines the top- k functions. For the `replace` program under study, where $m = 20$ and $n = 30$, only 1.5 errors are expected to be located if top-1 function is examined. In contrast, UPPERANK and COMBINERANK locates 9 and 11 errors respectively. Furthermore, when top-2 functions are examined, the number for UPPERANK and COMBINERANK is leveraged to 13 and 16 respectively whereas it is merely 3 for random rankings. Considering the tricky nature of these logic errors, we regard that the result shown in Figure 4 is excitingly good because our method infers about the buggy function purely from the execution statistics, and assumes no more program semantics than the random ranking. As one will see in Section 4.3, similar results are also observed on other programs.

4.2.2 Effectiveness for Various Errors

After the discussion about the overall effectiveness, one may also find it instructive to explore for what kinds of errors our method works well (or not). We thus break down errors in Table 1 into the following four categories and examine the effectiveness of our method on each of them.

1. **Incorrect Branch Expression (IBE)**: This category generally refers to errors that *directly* influ-

