How Detrimental is Coincidental Correctness to Defect Detection?

Anonymous Author(s)

Abstract*— According to the PIE and RIP models, three conditions must be satisfied for program failure to occur: 1) the defect's location must execute or be reached; 2) the program's state must become infected; and 3) the infection must propagate to the output. *Weak coincidental correctness* (or *weak CC*) occurs when the program produces the correct output, while condition 1) is satisfied but 2) and 3) are not satisfied. *Strong coincidental correctness* (or *strong CC*) occurs when the output is correct, while both conditions 1) and 2) are satisfied, but not 3). In the literature, typically *coincidental correctness* (*CC*) refers to *strong CC.

Researchers have recognized the presence of *CC* and analytically demonstrated that it is a safety-reducing factor for spectrum-based fault localization (SBFL). However, they did not empirically validate that fact, which we do in this paper. Specifically, using the *Defects4J* benchmark, we comparatively evaluated the performance of SBFL using 52 different suspiciousness metrics when: a) both *weak* and *strong CC* tests are present (T_{wsCC}); b) no *weak* nor *strong CC* tests are present (T_{noCC}); c) *weak CC* tests are present (T_{wCC}); and d) *strong CC* tests are present (T_{sCC}). Similarly, using five multi-fault Java programs, we evaluated the performance of greedy Test Suite Reduction (TSR) in the presence and absence of *CC*. That is, we empirically studied the impact of *CC* on defect detection using two commonly used techniques.

Using 49 out of the 52 metrics, our results showed with statistical significance that SBFL performs better when using T_{wCC}, T_{sCC}, and T_{noCC} than when using T_{wsCC}. They also showed that T_{noCC} yields the best performance followed by T_{sCC}, and then T_{wCC}. However, the *effect sizes* were mostly trivial, except for three metrics. Compared to T_{wsCC}, our TSR results showed that T_{noCC}, T_{sCC}, and T_{wCC} resulted in respectively %49, %47, and %6 more detected defects. Therefore, our empirical study suggests that *CC* is detrimental to defect detection, and that *weak CC* is more detrimental than *strong CC*.

Keywords*— *coincidental correctness, failed error propagation, fault masking, spectrum-based fault localization, coverage-based fault localization, test suite reduction, test suite minimization

I. INTRODUCTION

Voas presented the PIE model [33] that identifies three conditions to be satisfied for program failure to occur: 1) the defect is **e**xecuted, 2) the program is **i**nfected, and 3) the infection has **p**ropagated to the output. Amman and Offutt supported this same notion in their RIP (reachability-infection-propagation) model described in [3]. They later extended that model into the RIPR model by adding the *Reveal* condition, which requires the incorrect final infected state to be actually observed (e.g., by the tester or a test oracle) [4]. *Coincidental Correctness* (*CC*) [14][17][34] occurs when the program produces the correct output, while conditions 1) and 2) are satisfied but not 3). Some researchers referred to *CC* using different terms such as *fault masking* [12] and *failed error propagation* [5]. Others [28] differentiated two variants of it that we adopt in this paper: a) *Strong Coincidental Correctness* (or *strong CC*), which is identical to the original definition; and b) *Weak Coincidental Correctness* (or *weak CC*), which occurs when the program produces the correct output, while condition 1) is satisfied but 2) and 3) are not satisfied.

Coincidental correctness might result in overestimating the reliability of programs as it hides bugs that might subsequently surface following unrelated code modifications. In addition, it might reduce the effectiveness of various quality enhancing techniques. In fact, several researchers have recognized the negative impact of coincidental correctness on the effectiveness of defect detection techniques [6][7][8][13][16][17][34]. Others have empirically shown that both *weak* and *strong CC* are prevalent in widely used benchmarks [28], and have analytically demonstrated that *weak CC* is a safety reducing factor in spectrum-based fault localization (SBFL) [28]; i.e., when *weak CC tests* are present, the defect will be assigned a suspiciousness score smaller than when they are not present. However, we are not aware of any major empirical study that assessed the impact of *CC* (in either of its forms) on SBFL or any other defect detection technique. This paper achieves that by studying the impact of *weak CC* and *strong CC* on the effectiveness of two widely used defect detection techniques, namely, SBFL and coverage-based Test Suite Reduction (TSR) [40][32][27].

Our study considers the following four categories of test suites:

1) T_{wsCC}: both *weak* and *strong CC* tests are present.
2) T_{noCC}: no *weak* nor *strong CC* tests are present.
3) T_{wCC}: *weak CC* tests are present (but no *strong CC* tests).
4) T_{sCC}: *strong CC* tests are present (but no *weak CC* tests).

For SBFL, we use the *Defects4J* benchmark and 52 existing suspiciousness metrics. For TSR, we use five multi-fault Java programs and a commonly used greedy TSR technique. Our goal is to assess how detrimental *CC* is on defect detection by answering the following research questions:

RQ1: Does SBFL perform better when using T_{wCC}, T_{sCC}, or T_{noCC} than when using T_{wsCC}?

RQ2: Does TSR perform better when using T_{wCC}, T_{sCC}, or T_{noCC} than when using T_{wsCC}?

The main contributions of our work are as follows:

- Quantifying the impact of coincidental correctness on two commonly used defect detection techniques, namely, SBFL and TSR.
- Considering both forms of coincidental correctness, *weak CC* and *strong CC*.
- Identifying the SBFL metrics that are most resilient to *CC*, and those that are most vulnerable. For example, *Overlap* [29] and *Russel & Rao* [29] seemed unaffected by *CC*, whereas *Fossum* [36], *Tarwid* [36], and *Ochiai* [2] were highly affected.
- Most importantly, this paper provides empirical evidence that coincidental correctness is harmful to defect detection, which provides motivation for researchers to investigate effective solutions to mitigate its effect.

Section II provides a motivating example (borrowed from [1]) illustrating how *CC* could potentially reduce the effectiveness of SBFL, TSR, and test case prioritization. Sections III and IV respectively present our SBFL and TSR empirical results. Section V surveys related work, and Section VI concludes.

II. Motivating Example

This example illustrates how *CC* can have a negative impact on SBFL, TSR, and test case prioritization. The function shown in Figure 1 computes the median of three input numbers; it is frequently used in SBFL literature [23]. The defect is at Line 6, which assigns `y` to `m` as opposed to assigning it `x`. Figure 1 also shows six test cases and their corresponding statement coverage information. The only failing test case is t_6, which as expected, executes the defect at Line 6. Meanwhile, t_1 also executes the defect at Line 6 but outputs the correct result, which makes it a *CC* test case. t_1 is a *weak CC* test as opposed to a *strong CC* test since it executes the defect but does not cause an infection at Line 6. It erroneously assigns `y` to `m` as opposed to assigning `x`, but since `x` and `y` are both 3, the program state does not get infected.

The fact that t_1 is a *CC* test diminishes the correlation between the execution of Line 6 and failure, which translates into lessening the effectiveness of SBFL. In other words, including t_1 in the test suite yields a suspiciousness score at Line 6 that is lower than the score computed when t_1 is excluded. For example, the value of the *Tarantula* suspiciousness score [23] for Line 6 is 0.833 when t_1 is included and 1.0 when excluded.

A widely adopted TSR approach discards redundant tests while ensuring that the reduced test suite also covers the program elements covered by the original test suite. In our example, the test suite would be reduced to $T_1 = \{t_1, t_2, t_3, t_4\}$ or to $T_2 = \{t_2, t_3, t_4, t_6\}$ since these are the only minimal test suites that cover all statements covered by the original test suite. Consequently, there will only be a 50% chance that the reduced test suite will reveal the defect since: 1) T_1 and T_2 are equally likely to be generated; and 2) only T_2 includes the failing test t_6. Excluding the weak *CC* test t_1 from the original test suite would mean that only T_2 will be generated by test suite reduction; clearly, a more desirable outcome.

	Test Cases					
	t_1	t_2	t_3	t_4	t_5	t_6
int median(int x, int y, int z) {	3, 3, 5	1, 2, 3	3, 2, 1	5, 5, 5	5, 3, 4	2, 1, 3
1: int m = z;	✓	✓	✓	✓	✓	✓
2: if (y < z)	✓	✓	✓	✓	✓	✓
3: if (x < y)	✓	✓			✓	✓
4: m = y;		✓				
5: else if (x<z)	✓				✓	✓
6: m = y; // bug: m = x;	✓					✓
7: else			✓	✓		
8: if (x > y)			✓	✓		
9: m = y;			✓			
10: else if (x>z)				✓		
11: m = x;						
12: return m;	P	P	P	P	P	F
}						

Figure 1. Java code and corresponding statement coverage information

A classical test case prioritization approach gives higher execution priority to the test cases that cover the most not yet covered program elements. A large number of prioritization outcomes are possible for our example, of which we list only two that represent extreme cases, namely, $T_1 = \langle t_1, t_3, t_4, t_2, t_5, t_6 \rangle$ and $T_2 = \langle t_6, t_3, t_4, t_2, t_5, t_1 \rangle$. Clearly, T_2 is superior to T_1, since the defect is revealed by the first executed test in T_2 as opposed to the last executed test in T_1. Here also, excluding t_1 from the original test suite would result in a more effective prioritization. Actually, in case t_1 is excluded, t_6 will always be executed either first or second.

III. Impact of CC on SBFL

A. SBFL Techniques

Spectrum-based fault localization (or SBFL) assigns each covered program element a suspiciousness score reflecting its correlation with failure [2][13][22][25][26][30], it then provides the developer with a list of these elements ranked based on their likelihood of being faulty. Typically, SBFL techniques use statement coverage, but they might also use other types of coverage, such as branch and def-use. The experiments conducted in this paper involve statement coverage and 52 SBFL suspiciousness metrics that we identified in recent SBFL literature. The computation of these metrics depends on one or more of the following entities:

- a_{ep}: The number of passing test cases that execute the profile element.
- a_{ef}: The number of failing test cases that execute the profile element.
- a_{np}: The number of passing test cases that do no execute the profile element.
- a_{nf}: The number of failing test cases that do not execute the profile element.

- P or (a_{ep} + a_{np}): The total number of passing test cases.
- F or (a_{ef} + a_{nf}): The total number of failing test cases.

For example, the *Ochiai* [2] suspiciousness metric takes on the following form: $\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$

Table 1 lists all of the 52 suspiciousness metrics we used. Note that we excluded any metric that did not involve P, or a_{ep}. The reason is that for such metric the presence or absence of *CC* test cases will have no effect on the computed value. For example, given a test suite that contains n_{CC} coincidentally correct test cases, i.e., tests that executed the faulty location but did not induce a failure. To account for the presence of these *CC* tests, a_{ep} must be replaced by (a_{ep} - n_{CC}) to arrive at a more faithful suspiciousness value [28] of the faulty location. Considering the *Ample* [13] and *Binary* [29] metrics below:

$$Ample = \left| \frac{a_{ef}}{a_{ef}+a_{nf}} - \frac{a_{ep}}{a_{ep}+a_{np}} \right|, \quad Binary = \begin{cases} 0 & \text{if } a_{ef} < F \\ 1 & \text{if } a_{ef} = F \end{cases}$$

A non-zero value of n_{CC} will affect *Ample* (through a_{ep}) but not *Binary*, this is why our study includes *Ample* but not *Binary* (see Table 1).

In most cases, the developer is not interested in the raw suspiciousness scores but in the ranked list of suspicious program locations instead. Accordingly, in order to quantify the effectiveness of SBFL, we compute the EXAM score [35], which reflects the percentage of lines that are ranked higher than the faulty location. That is, a lower EXAM score indicates better performance. In case the faulty line is tied with other lines *w.r.t.* suspiciousness score, we rank it in the middle of them.

Finally, our instrumentation is carried out at the Java bytecode level using the ASM framework, whereas the ranking is conducted at the Java line level. The reason stems from the fact that it is easier to present bug locations in terms of readable Java lines as opposed to bytecode lines. Furthermore, the *Defects4J* benchmark provides the fault locations in the context of Java lines.

B. Subject Programs

The subject programs comprised 5 libraries from the Defects4J benchmark: *JFreeChart* (*Chart*), *Closure*, *Apache commons-lang* (*Lang*), *Apache commons-math* (*Math*), and *Joda-Time* (*Time*), with a total count of 357 versions, each containing a single bug. Our study also relies on the availability of test suites in which each test case is classified as *weak CC*, *strong CC*, failing, or true passing (a non-*CC* passing test). Abou Assi *et al* [1] extended the *Defect4J* benchmark in order to provide such information. They manually classified each test case under one of the above categories. We make use of such information to build $\boldsymbol{T_{wsCC}}$, $\boldsymbol{T_{noCC}}$, $\boldsymbol{T_{sCC}}$, and $\boldsymbol{T_{wCC}}$ for each version we included in our study. It is worth noting that compared to the failing tests, the numbers of *CC* tests they identified were considerable. For example, executing the 357 test suites from all five libraries on their respective defective versions involved 580 failures, 4187 *strong CC* tests, 4821 *weak CC* tests, and 563,314 non-*CC* passing tests [1]. Out of the 357 versions, we ignore 30 versions due to incomplete CC information or profiling errors.

C. RQ1: Does SBFL perform better when using T_{wCC}, T_{sCC}, or T_{noCC} than when using T_{wsCC}?

We computed the EXAM score for each of the versions using each of the 52 SBFL metrics and each of the four different categories of test suites. The plots in Figure 2 contrast the EXAM scores for when $\boldsymbol{T_{wsCC}}$ is used to when $\boldsymbol{T_{noCC}}$ is used, i.e., it shows for each metric the effect of discarding both *weak* and *strong CC* tests. Similarly, Figure 3 shows the same type of information but in relation to $\boldsymbol{T_{wCC}}$, i.e., it shows the effect of discarding the *strong CC* tests while keeping the *weak CC* tests. For space consideration, we show the plots associated with $\boldsymbol{T_{sCC}}$ in the online appendix [41]. Tables 2-4 provide summarized statistics about all three sets of plots, namely, the *p-values* and *effect sizes* related to each plot. For example, Table 2 shows with statistical significance but negligible effect size that, for 46 metrics, SBFL performs better when using $\boldsymbol{T_{noCC}}$ than when using $\boldsymbol{T_{wsCC}}$.

Each scatter plot in Figure 2 is associated with one of the 52 SBFL metrics. The green data points correspond to when the EXAM score improved (got smaller) when discarding the *weak* and *strong CC* tests. The red data points correspond to when the EXAM score worsened; the blue data points correspond to when the score remained unchanged. Note how the blue data points fall on the lines dividing the plots. Also, each plot is annotated with the *p-value* and the *effect size* computed according to *Cliff's delta* [31], which is a non-parametric measure that quantifies whether the elements in one group are larger than those of another group. The *effect size* is considered *trivial* (|delta| < 0.147), *small* (0.147 ≤ |delta| < 0.33), *moderate* (0.33 ≤ |delta| < 0.474), or *large* (> 0.474).

We now summarize our results related to discarding both *weak* and *strong CC* tests ($\boldsymbol{T_{noCC}}$) based on Figure 2 and Table 2:

1) Two metrics were not affected by the presence of *weak* and *strong CC* tests, namely, *Overlap* [29] and *Russel & Rao* [29]. Note how the corresponding plots only show blue data points, a *p-value* of 1, and an *effect size* of 0. These two metrics were resilient to *CC* despite the fact that they depend on a_{ep}.
2) Most metrics exhibited few cases in which the EXAM score slightly worsened; i.e., red data points showing right above the dividing line. However, *Michael* [36] additionally exhibited few red points noticeably above the dividing line, reflected in a *p-value* of 0.927, and an *effect size* of 0.004.
3) The remaining 49 metrics benefited from discarding the *weak* and *strong CC* tests to varying extents. We list them in descending order, i.e., in the order of being most vulnerable to the presence of *weak* and *strong CC* tests:
    a. *Fossum* and *Tarwid* [36] exhibited a *p-value* of 0, and *moderate effect sizes* of 0.387 and 0.421, respectively.
    b. *Ochiai* exhibited a *p-value* of 0, and *a small effect size* of 0.147.
    c. Forty-six metrics exhibited a *p-value* of 0 and *trivial effect sizes* in the range [0.004, 0.119].

Figure 3 and Table 4 show the results related to discarding the *strong CC* tests while keeping the *weak CC* tests (**T_{wCC}**):

1) Here also *Overlap* and *Russel & Rao* were not affected by the presence of *weak CC* tests, and similarly exhibiting a *p-value* of 1, and an *effect size* of 0.
2) *Michael* [36] also exhibited few red points noticeably above the dividing line. However, the *p-value* decreased to 0.844 (still not significant), and the *effect size* increased to 0.004 (still *trivial*).
3) *Fossum* and *Tarwid* [36] exhibited a *p-value* of 0, and *small effect sizes* of 0.153 and 0.178, respectively. Noting that in the context of $\boldsymbol{T_{noCC}}$, their effective sizes were *moderate*.
4) The remaining 47 metrics exhibited *trivial effect sizes*, while the *p-value* was 0 for 23 of them, and < 0.01 for 13 of them.

Tarantula [23]: $\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$	*Ochiai* [2]: $\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$	*Ochiai2* [29]: $\frac{a_{ef}a_{np}}{\sqrt{(a_{ef}+a_{ep})(a_{np}+a_{nf})(a_{ef}+a_{nf})(a_{ep}+a_{np})}}$
Kulczynski [10]: $\frac{a_{ef}}{a_{nf}+a_{ep}}$	*Kulczynski2* [29]: $\frac{1}{2}\left(\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ef}}{a_{ef}+a_{ep}}\right)$	*Dstar*^{2} [35]: $\frac{(a_{ef})^2}{a_{nf}+a_{ep}}$
Dstar^3 [35]: $\frac{(a_{ef})^3}{a_{nf}+a_{ep}}$	*O* [29] : $\begin{cases}-1 & \text{if } a_{nf}>0\\ a_{np} & \text{otherwise}\end{cases}$	*O*^{p} [29]: $a_{ef}-\frac{a_{ep}}{a_{ep}+a_{np}+1}$
Wong2 [37]: $a_{ef}-a_{ep}$	*Wong3* [37]: $a_{ef}-h,\quad where$ $h=\begin{cases}a_{ep} & if\ a_{ep}\le 2\\ 2+0.1(a_{ep}-2) & if\ 2<a_{ep}\le 10\\ 2.8+0.001(a_{ep}-10) & if\ a_{ep}>10\end{cases}$	*Wong3'* [29]: $\begin{cases}-1000 & if\ a_{ep}+a_{ef}=0\\ Wong3 & otherwise\end{cases}$
CBI Inc [29]: $\frac{a_{ef}}{a_{ef}+a_{ep}}-\frac{a_{ef}+a_{nf}}{a_{ef}+a_{nf}+a_{np}+a_{ep}}$	*CBI Log* [29]: $\frac{2}{\frac{1}{CBI\ Inc}+\frac{\log(a_{ef}+a_{nf})}{\log(a_{ef})}}$	*CBI Sqrt* [29]: $\frac{2}{\frac{1}{CBI\ Inc}+\frac{\sqrt{a_{ef}+a_{nf}}}{\sqrt{a_{ef}}}}$
Russel & Rao [29]: $\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$	*Tarwid* [36]: $\frac{(F+P)(a_{ef})-(a_{ef}+a_{nf})(a_{ef}+a_{ep})}{(F+P)(a_{ef})+(a_{ef}+a_{n})(a_{ef}+a_{ep})}$	*Jaccard* [2]: $\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$
Zoltar [20]: $\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+\frac{10000a_{nf}a_{ep}}{a_{ef}}}$	*Anderberg* [29]: $\frac{a_{ef}}{a_{ef}+2(a_{nf}+a_{ep})}$	*Sørensen-Dice* [29]: $\frac{2a_{ef}}{2a_{ef}+a_{nf}+a_{ep}}$
Dice [29]: $\frac{2a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$	*Goodman* [29]: $\frac{2a_{ef}-a_{nf}-a_{ep}}{2a_{ef}+a_{nf}+a_{ep}}$	*Barinel* [29]: $\frac{a_{ef}}{a_{ef}+a_{ep}}$
Hamann [29]: $\frac{a_{ef}+a_{np}-a_{nf}-a_{ep}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$	*Simple Matching* [29]: $\frac{a_{ef}+a_{np}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$	*Sokal* [29]: $\frac{2(a_{ef}+a_{np})}{2(a_{ef}+a_{np})+a_{nf}+a_{ep}}$
Rogers & Tan. [29]: $\frac{a_{ef}+a_{np}}{a_{ef}+a_{np}+2(a_{nf}+a_{ep})}$	*Hamming* [29]: $a_{ef}+a_{np}$	*Euclid* [29]: $\sqrt{a_{ef}+a_{np}}$
Rogot1 [29]: $\frac{1}{2}\left(\frac{a_{ef}}{2a_{ef}+a_{nf}+a_{ep}}+\frac{a_{np}}{2a_{np}+a_{nf}+a_{ep}}\right)$	*Rogot2* [29]: $\frac{1}{4}\left(\frac{a_{ef}}{a_{ef}+a_{ep}}+\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{np}}{a_{np}+a_{ep}}+\frac{a_{np}}{a_{np}+a_{nf}}\right)$	*M1* [29]: $\frac{a_{ef}+a_{np}}{a_{nf}+a_{ep}}$
M2 [29]: $\frac{a_{ef}}{a_{ef}+a_{np}+2(a_{nf}+a_{ep})}$	*Arithmetic Mean* [29]: $\frac{2a_{ef}a_{np}-2a_{nf}a_{ep}}{(a_{ef}+a_{ep})(a_{np}+a_{nf})+(a_{ef}+a_{nf})(a_{ep}+a_{np})}$	*Geometric Mean* [29]: $\frac{a_{ef}a_{np}-a_{nf}a_{ep}}{\sqrt{(a_{ef}+a_{ep})(a_{np}+a_{nf})(a_{ef}+a_{nf})(a_{ep}+a_{np})}}$
Cohen [29]: $\frac{2a_{ef}a_{np}-2a_{nf}a_{ep}}{(a_{ef}+a_{ep})(a_{np}+a_{ep})+(a_{ef}+a_{nf})(a_{nf}+a_{np})}$	*BB & Buser* [29]: $\frac{\sqrt{a_{ef}a_{np}}+a_{ef}}{\sqrt{a_{ef}a_{np}}+a_{ef}+a_{ep}+a_{nf}}$	*Scott* [29]: $\frac{4a_{ef}a_{np}-4a_{nf}a_{ep}-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})(2a_{np}+a_{nf}+a_{ep})}$
Fleiss [29]: $\frac{4a_{ef}a_{np}-4a_{nf}a_{ep}-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})+(2a_{np}+a_{nf}+a_{ep})}$	*Overlap* [29]: $\frac{a_{ef}}{\min(a_{ef},a_{nf},a_{ep})}$	*Braun-Banquet* [36]: $\frac{a_{ef}}{\max(a_{ef}+a_{ep},a_{ef}+a_{nf})}$
Dennis [36]: $\frac{(a_{ef}a_{np})-(a_{ep}a_{nf})}{\sqrt{(F+P)(a_{ef}+a_{ep})(a_{ef}+a_{nf})}}$	*Mountford* [36]: $\frac{a_{ef}}{0.5(a_{ef}a_{ep}+a_{ef}a_{nf})+a_{ep}a_{nf}}$	*Fossum* [36]: $\frac{(F+P)(a_{ef}-0.5)^2}{(a_{ef}+a_{ep})(a_{ef}+a_{nf})}$
Pearson [36]: $\frac{(F+P)(a_{ef}a_{np}-a_{ep}a_{nf})^2}{(a_{ef}+a_{ep})(a_{nf}+a_{np})(a_{ep}+a_{np})(a_{ef}+a_{nf})}$	*Gower* [36]: $\frac{a_{ef}+a_{np}}{\sqrt{(a_{ef}+a_{ep})(a_{nf}+a_{np})(a_{ep}+a_{np})(a_{ef}+a_{nf})}}$	*Michael* [36]: $\frac{4(a_{ef}a_{np}-a_{ep}a_{nf})}{(a_{ef}+a_{np})^2+(a_{ep}+a_{nf})^2}$
Pierce [36]: $\frac{a_{ef}a_{nf}+a_{nf}a_{ep}}{a_{ef}a_{nf}+2a_{nf}a_{np}+a_{ep}a_{np}}$	*Harmonic Mean* [29]: $\frac{(a_{ef}a_{np}-a_{nf}a_{ep})\left((a_{ef}+a_{ep})(a_{np}+a_{nf})+(a_{ef}+a_{nf})(a_{ep}+a_{np})\right)}{(a_{ef}+a_{ep})(a_{np}+a_{nf})(a_{ef}+a_{nf})(a_{ep}+a_{np})}$	
Ample [13]: $\left\lvert\frac{a_{ef}}{a_{ef}+a_{nf}}-\frac{a_{ep}}{a_{ep}+a_{np}}\right\rvert$	*Ample2* [29]: $\frac{a_{ef}}{a_{ef}+a_{nf}}-\frac{a_{ep}}{a_{ep}+a_{np}}$	

Table 1- SBFL suspiciousness metrics

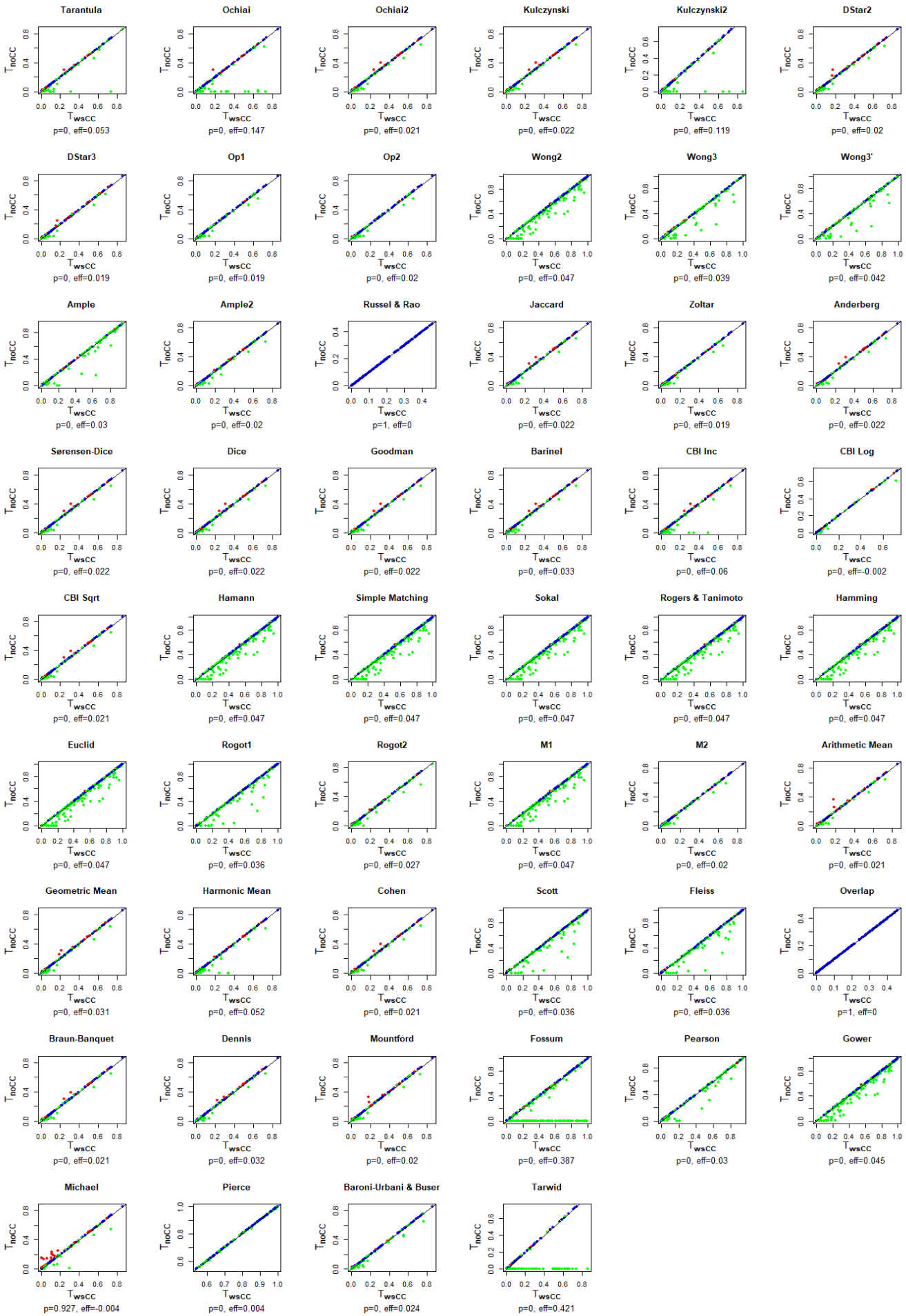

Figure 2 – T_{wsCC} vs. T_{noCC} (no *weak* nor *strong CC* tests present)

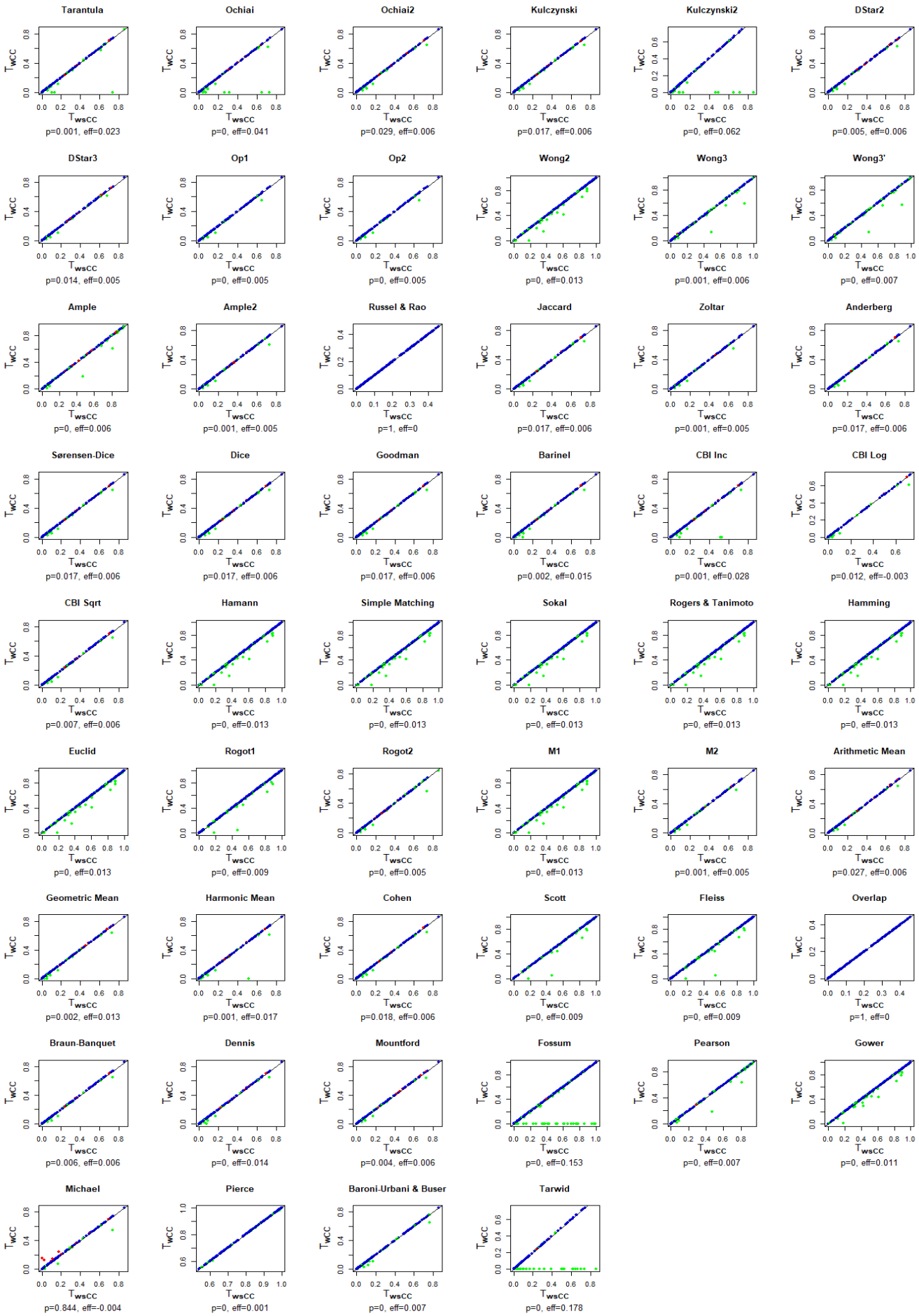

Figure 3 – T_{wsCC} vs. T_{wCC} (no *strong CC* tests present)

As shown in Tables 2-4, the performance of SBFL improved most with T_{noCC}, followed by T_{sCC}, and then T_{wCC}. In all three cases, the impact of discarding the *CC* tests was statistically significant for 49 out of 52 metrics. However, the *effect sizes* were mostly *trivial*, except in the case of *Fossum*, *Tarwid*, and *Ochiai*.

To answer RQ1, our study showed with statistical significance that SBFL performs better when using T_{wCC}, T_{sCC}, and T_{noCC} than when using T_{wsCC}. It also showed that T_{noCC} yields the best performance followed by T_{sCC}, and then T_{wCC}. In other words, it provides empirical evidence that *CC* is detrimental to SBFL, and that *weak CC* is more detrimental than *strong CC*.

T_{noCC} improves SBFL?		p-value			
		0	**]0,0.01[**	**[0.01,0.05[**	**[0.05,1]**
effect size	**trivial** |delta|<0.147	46	0	0	3
	small 0.147 ≤ |delta| < 0.33	1	0	0	0
	moderate 0.33 ≤ |delta| < 0.474	2	0	0	0
	large |delta| ≥ 0.474	0	0	0	0

Table 2 – T_{wsCC} vs. T_{noCC} (no *weak* nor *strong CC* tests present)

T_{sCC} improves SBFL?		p-value			
		0	**]0,0.01[**	**[0.01,0.05[**	**[0.05,1]**
effect size	**trivial** |delta|<0.147	44	3	0	3
	small 0.147 ≤ |delta| < 0.33	2	0	0	0
	moderate 0.33 ≤ |delta| < 0.474	0	0	0	0
	large |delta| ≥ 0.474	0	0	0	0

Table 3- T_{wsCC} vs. T_{sCC} (no *weak CC* tests present)

T_{wCC} improves SBFL?		p-value			
		0	**]0,0.01[**	**[0.01,0.05[**	**[0.05,1]**
effect size	**trivial** |delta|<0.147	23	13	11	3
	small 0.147 ≤ |delta| < 0.33	2	0	0	0
	moderate 0.33 ≤ |delta| < 0.474	0	0	0	0
	large |delta| ≥ 0.474	0	0	0	0

Table 4 – T_{wsCC} vs. T_{wCC} (no *strong CC* tests present)

IV. Impact of CC on TSR

A. Greedy TSR

Given a program *P*, a test suite *T*, and a set of test requirements *TR* that are covered by *T*. Coverage-based *Test suite reduction* (or TSR) aims at finding *T'*, a minimal subset of *T*, that covers all test requirements in *TR*. The conjecture is that (the smaller) *T'* would be as effective as *T* in revealing defects [32]. A widely used form of coverage-based TSR selects test cases from *T* to include in *T'* in a way that maximizes the proportion of profile elements that are covered. It attempts to cover as many of the elements covered by *T* with as few test cases as possible. A coverage-maximizing subset of a test suite is an instance of the *set-cover problem*, which is NP-complete but which admits a greedy approximation algorithm [9][18]. The greedy algorithm selects the test that covers the largest number of elements not covered by the previously selected tests. Note that this algorithm might encounter ties; i.e., different tests might each cover the maximal number of elements. In order to break the tie, we randomly select one of the tied test cases, which means that applying the algorithm several times might yield different minimized test suites. For that reason, when reducing a test suite in our experiments, we apply 10 instances of the above greedy TSR then report the resulting average. Note that more sophisticated approaches were devised for tie breaking [15][21][27]. For example, Lin and Huang [27] used one type of coverage as the primary criterion (branch) and another type of coverage (def-use) for tie breaking. Eghbali and Tahvildari [15] selected one test over another based on the coverage frequency of the program elements covered so far. However, randomly resolving ties is widely accepted in the literature.

B. Subject Programs

In the context of TSR, it is more realistic to use multi-fault programs and system test suites, which required us to look for an alternative to *Defects4J*. Therefore, we conducted our experiments using *NanoXML* releases *r1* through *r5*, and the *JTidy* HTML syntax checker and pretty printer *release 3*. The *NanoXML* releases and test suites were download from the SIR repository (*sir.unl.edu*), whereas *JTidy* was made available by authors of work related to coincidental correctness [28].

JTidy contains two real faults; whereas, each of the *NanoXML* releases is associated with several versions that are seeded with single faults. We created multi-fault versions out of each of the *NanoXML* releases by randomly selecting five of the provided defects. However, we discarded some defects due to one or more of the following constraints: 1) no two defects could involve the same statement; 2) no two defects could consistently be triggered by the same test cases; and 3) a defect must induce a failure in at least one test case. As a result, the *NanoXML* defects varied from 3 to 4, and *NanoXML r4* was entirely discarded.

The *JTidy* test suite comprised 1000 tests, of which five are XML files and the rest are HTML files. Table 5 provides information about our subject programs: a) number of defects; b) test suite sizes; c) total number of failures; d) number of failures per defect; e) number of non-CC passing tests; and f) number of *weak* and *strong CC* tests (determined per Section IV.C). Note how: 1) *Nano r1* and *Nano r3* do not have any *weak CC* tests, and *Nano r5* has no *strong CC* tests; and 2) the number of *weak CC* tests is relatively high, which agrees with the findings in [28]. (All subject programs and test suites are downloadable [41]).

C. *RQ2: Does TSR perform better when using T_{wCC}, T_{sCC}, or T_{noCC} than when using T_{wsCC}?*

Here also we need to build T_{wsCC}, T_{noCC}, T_{wCC}, and T_{sCC} for each of our five subject programs. To do so, we specified two code checkers for each of the 15 defects: 1) a weak checker that detects *weak CC* by monitoring whether the defect is reached; and 2) a strong checker that detects *strong CC* by monitoring whether the program state is infected. We augmented the fixed versions of the subjects with the code checkers, and executed the test suites in order to categorize each test case as *strong CC*, or *weak CC*, or neither (resulting in the tests' breakdown shown in Table 5). Below is an illustrating example involving *JTidy* (org.w3c.tidy.Clean.java):

```
Buggy version:
    if (name_end > style.length() || style.charAt(name_end)…

Fixed version:
    if (name_end >= style.length() || style.charAt(name_end)…

Fixed version augmented with code checkers:
    weakCC = true;              // weak checker triggered
    if (name_end == style.length())
        strongCC = true;     // strong checker triggered
    if (name_end >= style.length() || style.charAt(name_end)…
```

For each subject, we performed test suite reduction using statement profiles. Since the greedy reduction algorithm is not deterministic, we applied it 10 times for each subject and computed the average sizes of the reduced test suites and the average numbers of the revealed defects. Tables 6 reports our results in terms of the average percentage of revealed defects ($df_{\%}$).

As shown in Table 5, the number of failures per bug is in some cases unrealistically high. For that reason we limited that number to 1 randomly picked failure. Table 6 reports the results for the case of 1 failure per bug. It shows for each subject program $df_{\%}$ computed using T_{wsCC}, T_{noCC}, T_{sCC}, and T_{wCC}. It additionally contrasts the results of when using T_{noCC}, T_{sCC}, and T_{wCC} to when T_{wsCC} is used.

Considering Table 6, applying TSR on *Nano r1* yielded a $df_{\%}$ of 69% using T_{wsCC}, 100% using T_{noCC}, and 100% using T_{sCC}. That is an improvement $\Delta df_{\%}$ of 44.9% for both T_{noCC} and T_{sCC} over T_{wsCC}. Note how the entries for T_{wCC} are left blank, and the results for T_{noCC} and T_{sCC} are identical. The reason is due to the fact that *Nano r1* has no *strong CC* tests (see Table 5). The results for *Nano r3* are very similar, both T_{noCC} and T_{sCC} resulted in a $df_{\%}$ of 100%, and T_{wCC} was discarded due the lack of *strong CC* tests.

Applying TSR on *Nano r5* yielded a $df_{\%}$ of 55% using T_{wsCC}, 100% using T_{noCC}, and 100% using T_{wCC}. However, here the entries for T_{sCC} are left blank since *Nano r5* has no *weak CC* tests. The results were very different for *Nano r2*, as neither *weak CC* nor *strong CC* had any impact on $df_{\%}$.

Applying TSR on *JTidy* yielded a $df_{\%}$ of 50% using T_{wsCC}, 100% using T_{noCC}, 80% using T_{sCC}, and 50% using T_{wCC}.

In regard to defect detection ($df_{\%}$), the results for *JTidy*, *Nano r1*, and *Nano r3* suggest that discarding *weak CC* tests drastically improves TSR (their respective $\Delta df_{\%}$ are 60%, 44.9% and 81%). The results for *Nano r5* suggest that discarding *strong CC* tests improves TSR ($\Delta df_{\%}$ is 17.6%). The results for *JTidy* also suggest that both *weak* and *strong CC* tests need to be discarded to improve TSR. Meanwhile, the results for *Nano r2* suggest that discarding any form of *CC* tests has no impact on TSR ($\Delta df_{\%}$ is 0%).

To summarize, compared to T_{wsCC}, our TSR results showed that T_{noCC}, T_{sCC}, and T_{wCC} resulted in respectively %49, %47, and %6 more detected defects.

The above observations are all based on the case of 1 failure per bug and on using statement profiles. To enhance our confidence in our findings, we opted to slightly change our experimental setup by using branch profiling and limiting the number of failures per bug to up 2 and up to 3.

Tables 7 and 8 present the results also when using statement profiles but while using up to 2 and 3 failures per bug, respectively. Tables 9-11 present similar information as Tables 6-8 except that branch profiling is used. Following a comparative examination, we are convinced that the conclusions drawn from Table 6 are fundamentally the same that one can draw from Tables 7-11.

To answer RQ2, our (relatively small) study showed that TSR performs better when using T_{wCC}, T_{sCC}, and T_{noCC} than when using T_{wsCC}. It also showed that T_{noCC} yields the best performance followed by T_{sCC}, and then T_{wCC}. In other words, it

Program	*#Bugs*	*#Tests*	*#Failures*	*#Failures/Bug*	*Passing Test Cases*		
					#no CC	*#weak CC*	*#strong CC*
Nano r1	3	214	38	27,10,1	14	162	0
Nano r2	3	214	41	36,4,1	141	17	15
Nano r3	4	216	29	16,8,4,1	19	168	0
Nano r5	3	216	32	30,1,1	135	0	49
JTidy	2	1000	22	17,5	79	874	25

Table 5 – Information about the number of bugs and the test suites breakdown

	T_{wsCC}	T_{noCC}		T_{sCC}		T_{wCC}	
	$df_{\%}$	$df_{\%}$	$\Delta df\%$	$df_{\%}$	$\Delta df_{\%}$	$df_{\%}$	$\Delta df_{\%}$
JTidy	50	100	100.0	80	60.0	50	0.0
Nano r1	69	100	44.9	100	44.9	67	-2.9
Nano r2	94	94	0.0	94	0.0	94	0.0
Nano r3	55	100	81.8	100	81.8	54	-1.8
Nano r5	85	100	17.6	84	-1.2	100	17.6

Table 6 – Impact of T_{noCC}, T_{sCC}, and T_{wCC} on TSR using statement coverage given 1 failure per bug

	T_{wsCC}	T_{noCC}		T_{sCC}		T_{wCC}	
	$df_{\%}$	$df_{\%}$	$\Delta df\%$	$df_{\%}$	$\Delta df_{\%}$	$df_{\%}$	$\Delta df_{\%}$
JTidy	50	100	100.0	95	90.0	50	0.0
Nano r1	70	100	42.9	100	42.9	71	1.4
Nano r2	100	100	0.0	100	0.0	100	0.0
Nano r3	59	100	69.5	100	69.5	61	3.4
Nano r5	93	100	7.5	94	1.1	100	7.5

Table 7 – Impact of T_{noCC}, T_{sCC}, and T_{wCC} on TSR using statement coverage given up to 2 failures per bug

	T_{wsCC}	T_{noCC}		T_{sCC}		T_{wCC}	
	$df_{\%}$	$df_{\%}$	$\Delta df\%$	$df_{\%}$	$\Delta df_{\%}$	$df_{\%}$	$\Delta df_{\%}$
JTidy	50	100	100.0	100	100.0	50	0.0
Nano r1	72	100	38.9	100	38.9	71	-1.4
Nano r2	100	100	0.0	100	0.0	100	0.0
Nano r3	62	100	61.3	100	61.3	63	1.6
Nano r5	99	100	1.0	98	-1.0	100	1.0

Table 8 – Impact of T_{noCC}, T_{sCC}, and T_{wCC} on TSR using statement coverage given up to 3 failures per bug

	T_{wsCC}	T_{noCC}		T_{sCC}		T_{wCC}	
	$df_{\%}$	$df_{\%}$	$\Delta df\%$	$df_{\%}$	$\Delta df_{\%}$	$df_{\%}$	$\Delta df_{\%}$
JTidy	100	100	0.0	100	0.0	100	0.0
Nano r1	68	100	47.1	100	47.1	68	0.0
Nano r2	94	93	-1.1	95	1.1	96	2.1
Nano r3	52	100	92.3	100	92.3	53	1.9
Nano r5	100	100	0.0	100	0.0	100	0.0

Table 9 – Impact of T_{noCC}, T_{sCC}, and T_{wCC} on TSR using branch coverage given 1 failure per bug

	T_{wsCC}	T_{noCC}		T_{sCC}		T_{wCC}	
	$df_{\%}$	$df_{\%}$	$\Delta df\%$	$df_{\%}$	$\Delta df_{\%}$	$df_{\%}$	$\Delta df_{\%}$
JTidy	100	100	0.0	100	0.0	100	0.0
Nano r1	70	100	42.9	100	42.9	70	0.0
Nano r2	100	100	0.0	100	0.0	100	0.0
Nano r3	59	100	69.5	100	69.5	58	-1.7
Nano r5	100	100	0.0	100	0.0	100	0.0

Table 10 – Impact of T_{noCC}, T_{sCC}, and T_{wCC} on TSR using branch coverage given up to 2 failures per bug

	T_{wsCC}	T_{noCC}		T_{sCC}		T_{wCC}	
	$df_{\%}$	$df_{\%}$	$\Delta df\%$	$df_{\%}$	$\Delta df_{\%}$	$df_{\%}$	$\Delta df_{\%}$
JTidy	100	100	0.0	100	0.0	100	0.0
Nano r1	73	100	37.0	100	37.0	71	-2.7
Nano r2	100	100	0.0	100	0.0	100	0.0
Nano r3	61	100	63.9	100	63.9	60	-1.6
Nano r5	100	100	0.0	100	0.0	100	0.0

Table 11 – Impact of T_{noCC}, T_{sCC}, and T_{wCC} on TSR using branch coverage given up to 3 failures per bug

suggests that *CC* is detrimental to TSR, and that *weak CC* is more detrimental than *strong CC*.

V. THREATS TO VALIDITY

Our SBFL related study is sizable as it involved the *Defects4J* benchmark. However, our TSR related study is relatively smaller, which is due to the lack of sizable benchmarks comprising multi-fault programs. Our intent is to extend the *Defects4J* benchmark with multi-fault versions, starting with the *Closure* library since it already contains test cases that result in large levels of coverage; i.e., the *Closure* test cases are better categorized as system tests than unit tests.

One internal threat to the validity of our experiments relates to the fact that the EXAM score is computed at the Java line number level as opposed to the more granular bytecode level, which might lessen the accuracy of our results. In addition, it should be noted that the EXAM score ranking is just one approach of many presented in the literature, thus, better alternative approaches might exist.

Our work was limited to only two coverage-based defect detection techniques, others should also be investigated; e.g., test case prioritization and test case selection.

VI. Related Work

This section presents work related to coincidental correctness. Wong *et al* [36], and Yoo and Harman [40], respectively provide comprehensive surveys on SBFL and TSR.

Laski *et al* [24] studied coincidental correctness while referring to it as *error masking*. They mutated the internal program state then checked whether the program produced the correct output. Their approach assesses the quality of a test suite and the presence of *CC* within.

Masri and Abou Assi [28] showed that both strong and weak *CC* is prevalent, and analytically demonstrated that *weak CC* is a safety reducing factor for SBFL. They also presented techniques for cleansing test suites from *CC*. Specifically, they segregated test cases into two clusters based on their execution profiles; a passing test was deemed as *CC* if it fell within the cluster that contained most of the failing tests. They also presented a second technique in which only passing tests are partitioned into two clusters, conjecturing that the *CC* test cases will be grouped within the same cluster.

Hierons [17] recognized the negative effect of *CC* when augmenting Partition Analysis with Boundary Value Analysis. He also showed how Boundary Value Analysis can be enhanced in order to reduce the likelihood of *CC* even in an environment that involves non-determinism and floating point numbers.

Baudry *et al* [8] defined a dynamic basic block (DBB) as a set of statements that is covered by the same test cases. They empirically observed that test suites containing more DBB's resulted in improved SBFL. They also observed that the actual faulty DBB's were not always ranked as the most suspicious, which they attributed to *CC*.

Wang *et al* [34] presented a coverage refinement approach to reduce the influence of coincidental correctness on fault localization. The work introduces a concept called *context-pattern*, which is unique for each fault type and describes the program behavior before and after the faulty code. Coverage results for all statements are refined with the context-pattern following a context-pattern matching

Bandyopadhyay and Ghosh [7] considered any passing test that is similar to failing tests as *CC*. They proposed an iterative approach that leverages user feedback to improve the detection of *CC* tests and consequently SBFL.

Clark and Hierons [12] leveraged information theory to study *strong coincidental correctness*, which they termed *fault masking*. They derived an information theoretic measure, termed *squeeziness*, which quantifies the likelihood of a function *f* to nullify the propagation of an infectious state. The *squeeziness* of function $f: I \rightarrow O$ is $S_q(f) = H(I) - H(O)$, which is the loss of information after applying *f* to *I*. Their proposed measure somewhat relates to mutual information and the measure presented in [28] for quantifying the amount of information flowing between two variables connected by a dynamic dependence chain. Clark and Hierons [12] also demonstrated that there is a strong statistical correlation between *squeeziness* and fault masking, whereas the correlation between DRR and fault masking was not as strong.

Androutsopoulos *et al* [5] provided a more thorough information theoretic analysis of *strong CC*, which they termed *Failed Error Propagation* or *FEP*. They devised five metrics that aim at predicting the occurrence of *strong CC*, of which two showed a high predictive power. Their experiments showed that 10% of the 7,140,000 involved test cases were *strong CC* tests.

In a position paper, Clark *et al* [11] proposed information theory as the basis for solving several software engineering problems including the mitigation of coincidental correctness to increase the testability of programs.

VII. Conclusions

We conducted a study that aimed at assessing the impact of coincidental correctness (CC), in both of its forms *weak* and *strong*, on the effectiveness of SBFL and coverage-based TSR. Our observations suggested that *CC* is detrimental to defect detection, and that *weak CC* is more detrimental than *strong CC*. We expect that our findings would motivate researchers to investigate effective solutions to mitigate its effect.

References

[1] Rawad Abou Assi, Chadi Trad, Marwan Maalouf, and Wes Masri. Does the Testing Level affect the Prevalence of Coincidental Correctness? https://arxiv.org/abs/1808.09233 (2018).

[2] Abreu R., Zoeteweij P. and Van Gemund A. J. C. On the Accuracy of Spectrum-based Fault Localization. TAIC-PART, pp. 89-98, 2007.

[3] Ammann P. and Offutt J. Introduction to Software Testing. Cambridge University Press, 2008.

[4] Ammann P. and Offutt J. Introduction to Software Testing. Cambridge University Press, 2nd edition, 2016.

[5] Kelly Androutsopoulos, David Clark, Haitao Dan, Robert M. Hierons, Mark Harman. An analysis of the relationship between conditional entropy and failed error propagation in software testing. ICSE 2014: 573-583.

[6] Thomas Ball, Mayur Naik, Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. POPL 2003: 97-105.

[7] Aritra Bandyopadhyay, Sudipto Ghosh. Tester Feedback Driven Fault Localization. ICST 2012: 41-50.

[8] B. Baudry, F. Fleurey, and Y. Le Traon, Improving test suites for efficient fault localization, In Proc.of ICSE'06, pages 82- 91, May, 2006.

[9] V. Chvatal. 1979. A Greedy Heuristic for the Set-Covering Problem. Math. Oper. Res. 4, 3 (Aug. 1979), 233–235.

[10] S. Choi, S. Cha, and C. C. Tappert, "A survey of binary similarity and distance measures," *J. Systemics, Cybern. Inf.*, vol. 8, no. 1, pp. 43–48, Jan. 2010.

[11] David Clark, Robert Feldt, Simon M. Poulding, Shin Yoo. Information Transformation: An Underpinning Theory for Software Engineering. ICSE (2) 2015: 599-602.

[12] David Clark, Robert M. Hierons. Squeeziness: An information theoretic measure for avoiding fault masking. Inf. Process. Lett. 112(8-9): 335-340 (2012)

[13] Dallmeier V., Lindig C., and Zeller A. 2005.b. Lightweight defect localization for Java. In A. P. Black, editor, ECOOP 2005: 19th European Conference, Glasgow, UK, July 25–29, 2005. Proceedings, volume 3568 of LNCS, pages 528–550. Springer-Verlag.

[14] DeMillo, R. A., Lipton, R. J., and Sayward, F. G. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. Computer 11, 4 (Apr. 1978), 34-41.

[15] S. Eghbali and L. Tahvildari, "Test case prioritization using lexicographical ordering," IEEE Transactions on Software Engineering, vol. 42, no. 12, pp. 1178–1195, 2016.

[16] István Forgács, Antonia Bertolino. Preventing untestedness in data-flow based testing. Softw. Test., Verif. Reliab. 12(1): 29-58 (2002).

[17] R. M. Hierons. Avoiding coincidental correctness in boundary value analysis. ACM Transactions on Software Engineering and Methodology. Volume 15, Issue 3 (July 2006). Pages: 227 - 241.

[18] D.S. Hochbaum, Approximation algorithms for NP-hard problems, PWS Publishing, Boston, MA, 1997.

[19] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal*, Volume 10, No. 4, pages 405-435, 2005.

[20] Tom Janssen, Rui Abreu, Arjan J. C. van Gemund. Zoltar: A Toolset for Automatic Fault Localization. ASE 2009: 662-664.

[21] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse, "Adaptive random test case prioritization," in Proceedings of the International Conference on Automated Software Engineering. IEEE, 2009, pp. 233–244.

[22] Jones J. and Harrold M. J. "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," Proc. 20th IEEE/ACM Int'l Conf. Automated Software Eng. (ASE '05), pp. 273-282, 2005.

[23] Jones J., Harrold M. J., and Stasko J.. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering*, pp. 467-477, May 2001.

[24] J. W. Laski, W. Szermer, and P. Luczycki. Error masking in computer programs. Software Testing, Verication and Reliability, 1995.

[25] Liblit B., Aiken A., Zheng A., and Jordan M. Bug Isolation via Remote Program Sampling. Proc. ACM SIGPLAN 2003 Int'l Conf. Programming Language Design and Implementation (PLDI '03), pp. 141-154, 2003.

[26] Liblit B., Naik M., Zheng A., Aiken A., and Jordan M. Scalable Statistical Bug Isolation. Proc. ACM SIGPLAN 2005 Int'l Conf. Programming Language Design and Implementation (PLDI '05), pp. 15-26, 2005.

[27] J.-W. Lin and C.-Y. Huang, "Analysis of test suite reduction with enhanced tie-breaking techniques," Information and Software Technology, vol. 51, no. 4, pp. 679–690, 2009.

[28] Wes Masri, Rawad Abou Assi. Prevalence of coincidental correctness and mitigation of its impact on fault localization. ACM Trans. Softw. Eng. Methodol. 23(1): 8:1-8:28 (2014).

[29] Naish L., Lee H.J., and Ramamohanarao K. A model for spectra-based software diagnosis. ACM Transactions on Software Engineering and Methodology. 20, 3, Article 11 (August 2011).

[30] Renieris M. and Reiss S. Fault localization with nearest-neighbor queries. In Proceedings of the 18th IEEE Conference on Automated Software Engineering, pp. 30-39, 2003.

[31] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohensd indices the most appropriate choices," in annual meeting of the Southern Association for Institutional Research, 2006.

[32] Rothermel G, Harrold M, Ronne J, Hong C. Empirical studies of test suite reduction. Software Testing, Verification, and Reliability December 2002; 4(2):219–249.

[33] Jeffrey M. Voas: PIE: A Dynamic Failure-Based Technique. IEEE Trans. Software Eng. 18(8): 717-727 (1992).

[34] Wang X., Cheung S.C., Chan W.K., Zhang Z. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. IEEE 31st International Conference on Software Engineering, pp. 45-55, 2009.

[35] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar Method for Effective Software Fault Localization," IEEE Transactions on Reliability, vol. 63, no. 1, pp. 290–308, 2014.

[36] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, Franz Wotawa. A Survey on Software Fault Localization. IEEE Trans. Software Eng. 42(8): 707-740 (2016).

[37] W. E.Wong, Y. Qi, L. Zhao, and K. Y. Cai, "Effective fault localization using code coverage," in *Proc. 31st Annu. Int. Comput. Software Appl. Conf. (COMPSAC)*, Beijing, China, Jul. 2007, pp. 449–456.

[38] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Transactions on Software Engineering and Methodology (TOSEM), 22(4):31, 2013.

[39] Meng Yan, Yicheng Fang, David Lo, Xin Xia, Xiaohong Zhang. File-Level Defect Prediction: Unsupervised vs. Supervised Models. ESEM 2017: 344-353.

[40] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," Software Testing, Verification and Reliability, vol. 22, no. 2, pp. 67–120, 2012.

[41] Online Appendix: (double-blind)