2010 Third International Conference on Software Testing, Verification and Validation

Cleansing Test Suites from Coincidental Correctness to Enhance Fault-Localization

Wes Masri
Department of Electrical and Computer Engineering
American University of Beirut
Beirut, Lebanon
wm13@aub.edu.lb

Rawad Abou Assi
Computer Science Department
American University of Beirut
Beirut, Lebanon
ria21@aub.edu.lb

***Abstract*— Researchers have argued that for failure to be observed the following three conditions must be met: 1) the defect is executed, 2) the program has transitioned into an infectious state, and 3) the infection has propagated to the output. *Coincidental correctness* arises when the program produces the correct output, while conditions 1) and 2) are met but not 3). In previous work, we showed that coincidental correctness is prevalent and demonstrated that it is a safety reducing factor for coverage-based fault localization. This work aims at cleansing test suites from coincidental correctness to enhance fault localization. Specifically, given a test suite in which each test has been classified as failing or passing, we present three variations of a technique that identify the subset of passing tests that are likely to be coincidentally correct. We evaluated the effectiveness of our techniques by empirically quantifying the following: 1) how accurately did they identify the coincidentally correct tests, 2) how much did they improve the effectiveness of coverage-based fault localization, and 3) how much did coverage decrease as a result of applying them. Using our better performing technique and configuration, the safety and precision of fault-localization was improved for 88% and 61% of the programs, respectively.**

Keywords- coincidental correctness, fault localization, software testing

I. INTRODUCTION

Coverage-based fault localization techniques aim at identifying the executing program elements that correlate with failure. They assume that when the defect is executed the program will fail, i.e., it will produce the wrong output; but this is not necessarily always the case. The PIE (propagation-infection-execution) model presented in [25] emphasizes that the execution of a defect is not a sufficient condition for failure, and that the propagation of the infectious state to the output is also required. This is also reinforced in the RIP (reachability-infection-propagation) model described in [3]. It is argued in both models that for failure to be observed the following three conditions must be met: 1) the defect is executed or reached, 2) the program has transitioned into an infectious state, and 3) the infection has propagated to the output. In previous work [19] we termed by *weak coincidental correctness* the case when the program produces the correct output, while only condition 1) is known to be met. We also termed by *strong coincidental correctness* the case when the program produces the correct output, while only conditions 1) and 2) are known to be met. Hereafter, *coincidental correctness* will refer to the latter definition as this is the most commonly adopted definition [7][9][10] [17][24][25].

In [19] we showed that coincidental correctness is prevalent in both of its forms (strong and weak), and demonstrated that it is a safety reducing factor for a *Tarantula* style coverage-based fault localization [12][13]. That is, when coincidentally correct tests are present, the defect will likely be ranked as less suspicious than when they are not present. In [20][21] we conducted an empirical study in which it was found that most dynamic information flows in programs do not convey any measurable information, which means that it is likely that many infectious states might not propagate to the output, thus, leading to coincidental correctness. Several other researchers have also studied and pointed out the prevalence of coincidental correctness and its degrading effect on fault localization [9][24][25]. All of this motivated us to investigate techniques to cleanse test suites from coincidentally correct tests in order to enhance coverage-based fault localization.

Given a test suite in which each test has been categorized as failing or passing, we present variations of a technique that identify the subset of passing tests that are likely to be coincidentally correct. Since our techniques require that tests are classified as failing or passing beforehand, they are generally not useful for testing, but they might possibly be useful for regression testing, something that we intend to investigate in future work. To evaluate the effectiveness of our techniques, we empirically quantified the following: 1) how accurately did they identify the coincidentally correct tests, 2) how much did they improve the effectiveness of coverage-based fault localization, and 3) how much did code coverage decrease as a result of applying them. The experiments were conducted using 18 seeded versions of programs belonging to the Siemens test suite [11] that were converted to Java. The results were promising, especially due to the low rate of false negatives and fault localization improvement achieved.

The main contributions of this paper are as follows:

a. A simple base technique for identifying coincidentally correct tests, which exhibits a low rate of false negatives

978-0-7695-3990-4/10 $26.00 © 2010 IEEE
DOI 10.1109/ICST.2010.22

b. Two variations of the base technique that reduce the number of false positives

Section II motivates our work. Section III describes our base technique (*Technique-I*). Section IV describes our experimental work and presents our results for *Technique-I*. Section V presents extensions to our base technique, *Techniques II*, and *III*. Section VI presents related work. Finally, Section VII presents our future work and conclusions.

II. Motivation

The work we conducted in [19] aimed at identifying the factors that impair coverage-based fault localization. In this section we present the factors we identified in [19] that are motivating to this work, namely, the prevalence of coincidental correctness and its safety reducing effect on fault localization. We also give a brief description of the results of an empirical study we conducted which showed that the absolute majority of information flows do not convey any measurable information [20][21], i.e., in most cases, having two variables or statements connected through a dynamic dependence chain does not necessarily mean that they are correlated. This means that it is likely that many infectious states might not propagate to the output, thus, leading to a potentially high rate of coincidental correctness.

A. Safety Reducing Effect

Here we demonstrate how coincidental correctness has a safety reducing effect on a *Tarantula* [13] style fault localization. The *Tarantula* suspiciousness metric is defined as follows:

$$\boldsymbol{M(e) = F \,/\, (F + P)} \quad \text{where}$$

e = faulty program element
$F = f \,/\, f_T$
$P = p \,/\, p_T$
f = # of failing runs that executed e
f_T = total # of failing runs
p = # of passing runs that executed e
p_T = total # of passing runs

Assume that n tests executed e but did not induce a failure. In this case, the value of $M(e)$ is misleading and to arrive at a more useful value we should subtract n from p and add it to f, the new value would become:

$$\boldsymbol{M'(e) = F' \,/\, (F' + P')} \quad \text{where}$$

$F' = (f+n) \,/\, (f_T + n)$
$P' = (p-n) \,/\, (p_T - n)$

It could be easily shown that $P' \leq P$ and $F' \geq F$, and consequently that $M'(e) \geq M(e)$, i.e., *not accounting for n would underestimate the suspiciousness of e*. To verify:

$M'(e) \geq M(e) \Rightarrow 1/M'(e) \leq 1/M(e) \Rightarrow 1 + P'/F' \leq 1 + P/F \Rightarrow P'/F' \leq P/F \Rightarrow P'/P \leq F'/F$ which holds.

B. Prevalence of Coincidental Correctness

Typically a test suite T is comprised of a set of failing tests T_F, and a set of passing tests T_P which itself might include a subset of coincidentally correct tests T_{CC}. In order to identify the distribution of T_{CC}/T we did the following for each of the seeded versions of our subject programs (described in [19]):

1) We created an oracle version of the program that sets a flag as soon as the condition for failure is met. To help clarify, the snippet of code below was extracted from one of the oracles of one of the subject programs (*NanoXML* [11]). It shows that the seeded fault is an if that replaced a while and that the condition for failure is set whenever the enumerator had more than one element in it:

```
if (e.hasMoreElements()) { //was: while(e.hasMoreElements())
… e.nextElement();…
}
if (e.hasMoreElements()) failure = true; // oracle added check
```

2) We executed the test suite on the original and seeded versions, and compared their respective outputs to determine the failures. Failures determined in this manner are called *output-based failures*.

3) We executed the test suite on the oracle version, and based on the values of the oracle added flags we determined the failures. Failures determined in this manner are called *defect-based failures*.

Therefore, coincidentally correct tests, elements of T_{CC}, are tests that exhibit a defect-based failure but not an output-based failure[1]. Figure 1 summarizes our results for all 148 seeded versions, among which 16 were derived from the *NanoXML* releases and 132 from the Siemens programs that were converted to Java. The horizontal axis represents the percentages of coincidentally correct tests, i.e., T_{CC}/T. Each bar corresponds to a range of size 10%. The vertical axis represents the percentage of seeded versions that exhibited a given range. The following observations could be made about the seeded versions exhibiting coincidental correctness:

a) 28% did not exhibit any
b) 40% exhibited a low level, in the range]0%, 10%[
c) 19% exhibited a medium level, in the range [10%, 60%[
d) 13% exhibited a high level, in the range [60%, 100%[

Also, not shown in Figure 1:

e) None of the seeded versions of *NanoXML v.3* exhibited any
f) Version 21 of *replace* exhibited the highest level of coincidental correctness, at 99.3%
g) The average over all 148 seeded versions is 15.7%

Undoubtedly, the exhibited levels of coincidental correctness are significant, i.e., coincidental correctness is a prevalent factor that is likely to reduce the safety of existing coverage-based techniques (that use output-based failures).

[1] This categorization of tests will be used in Sections III and IV

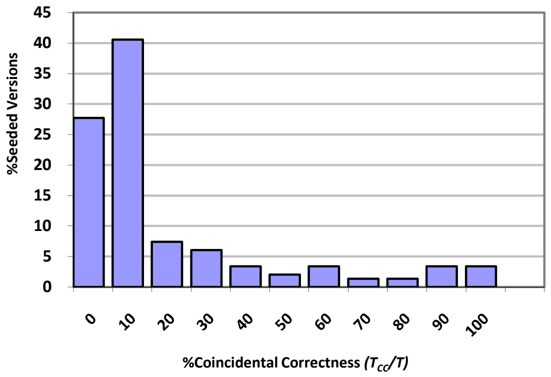

Figure 1. Distribution of coincidental correctness

C. *Prevalence of Weak Information Flows*

In this section we present the results of a study we conducted in [20][21] which show that weak information flows are prevalent, and that actually most of them have zero strength.

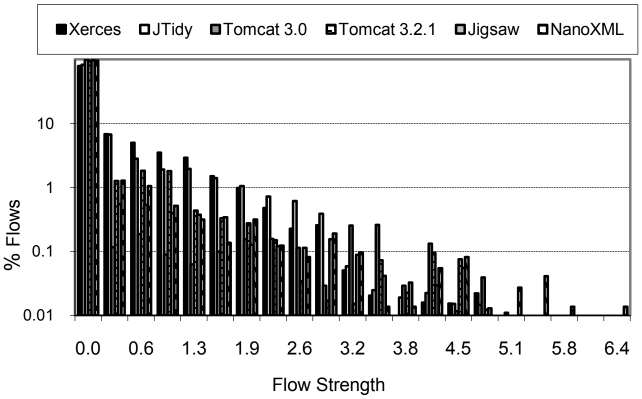

Figure 2. Information flow strength frequency distributions for the subject programs

We experimentally assessed for a set of six programs how often dynamic program dependences (direct/indirect dynamic data and control dependences) are indicative of actual, measurable information flows. First, we identified the dynamic dependences between all variables. Then, we computed the entropy-based strength of the identified flows using an approach based on information theory (a detailed description of the measure is provided in [20][21]). Figure 2 provides the results of the study. Each bar in the bar chart represents, for a specific subject program, the percentage of computed information flows (indicated on the vertical axis, in logarithmic scale) that have a given strength (indicated on the horizontal axis). For example, the first bar from the left shows that, for *NanoXML*, about 92% of the information flows have strength zero. As the figure shows, a very large percentage of traditionally computed information flows have estimated strength equal to zero. The percentages of flows with strength zero exhibited by *NanoXML*, *JTidy*, *Tomcat 3.0*, *Tomcat 3.2*, *Xerces*, and *Jigsaw* are 92.21%, 82.25%, 97.58%, 91.21%, 63.76%, and 94.61%, respectively. Note that some of the information flows did exhibit very high strength measures, but those represented rare cases. These results are in agreement with the other findings regarding the prevalence of coincidental correctness since for failure to be observed, infectious states must flow (propagate) to the output, and these results show that they might not in most cases.

III. CLEANSING COINCIDENTAL CORRECTNESS

Given a test suite in which each test has been categorized as failing or passing (where failures are obviously output-based failures), our techniques aim at identifying the subset of passing tests that are likely to be coincidentally correct. In this section we describe our base technique, *Technique-I*, then present the evaluation metrics we will use in our experiments. The extensions to our base technique are described in Section V.

A. *Description of Technique-I*

Given a test suite T comprised of a set of failing tests T_F, and a set of passing tests T_P which itself might include a subset of coincidentally correct tests T_{CC}; our aim is to identify T_{CC} given T_F and T_P. The premise is that using $\{T-T_{CC}\}$ instead of T would enhance the effectiveness of coverage-based fault localization as discussed in Section II-A.

First, our technique identifies *CCE*, the set of program elements (e.g., basic blocks, branches or def-use pairs) that are likely to be correlated with coincidentally correct tests; an element in this set will be called cc_e. Then, any test that induces one or more cc_e, called a cc_t, will be considered a potential coincidentally correct test. The set of identified cc_t's, our estimate of T_{CC}, will be called *CCT*. Assuming a single defect in the program, we conjecture that *a good candidate for a* cc_e *is a program element that occurs in all failing runs and in a non-zero but not excessively large percentage of passing runs*, that is:

a) $f_T(cc_e) = 1.0$ and

b) $0 < p_T(cc_e) \le \theta$

where $f_T(cc_e)$ is the percentage of T_F executing cc_e, $p_T(cc_e)$ the percentage of T_P executing cc_e, and $\theta < 1.0$.

The percentage $p_T(cc_e)$ cannot be zero because an element that never occurs in any passing runs is clearly not a good candidate for a cc_e, since a cc_t is a passing test. Figure 1 suggests that the majority of the programs exhibit coincidental correctness in the range]0%, 60%[, but many even in the range of [60%, 100%[, so θ must be chosen to be large (e.g., > 0.6) for our technique to be effective on programs with a high rate of coincidental correctness. Also, since initialization code typically executes in all runs, passing and failing, θ must be chosen to be < 1.0 so that program elements associated with initialization code would not be categorized as cc_e's. Based on the above, we believe that θ should be picked from the range]0.6, 0.9].

The following is the pseudo code of our algorithm for determining *CCT*, where for each test case t in T we denote by *E(t)* the set of all (profiled) program elements executed by t:

```
1.    CCE ← ∅
2.    for each e in $\bigcup_{t \in T} E(t)$
3.          if $f_T(e)$=1.0 and $p_T(e) \le \theta$
4.                CCE ← CCE ∪ {$e$}
5.    CCT ← ∅
6.    for each $t$ in $T$
7.          if E($t$) ∩ CCE ≠ ∅
8.                CCT ← CCT ∪ {$t$}
```

Lines 1-4 populate *CCE* with program elements that are totally correlated with failing runs and correlated with a given ratio of passing runs (determined by θ). In the context of a *Tarantula* style metric, these are elements that are suspicious but not entirely so (possibly due to passing runs that are coincidentally correct). Lines 5-8 populate *CCT* with tests that execute one or more cc_e's. This is undoubtedly a simple algorithm, but as the results of Sections IV and V show, it is somewhat effective.

B. Evaluation Metrics

In order to empirically evaluate the effectiveness of our techniques we compute metrics to quantify the generated false negatives and false positives. We also compute metrics that assess the impact of our techniques on the effectiveness of coverage-based fault localization and code coverage (a testing concern).

1) Measure of generated *false negatives*:

$$\frac{|T_{CC} - CCT|}{|T_{CC}|}$$

where T_{CC} is the difference between *defect-based failures* and *output-based failures* (see Section II-B). This measure assesses whether or not we are successfully identifying all of the coincidentally correct tests.

2) Measure of generated *false positives*:

$$\frac{|(T_P - T_{CC}) \cap CCT|}{|T_P - T_{CC}|}$$

This measure assesses whether we are erroneously categorizing tests as coincidentally correct.

3) Measure of *safety change:*

Safety indicates the relative suspiciousness of the faulty code. We define it as the cardinality of the following set:

$$S_1(T) = \left\{ \alpha \,\middle|\, \begin{matrix} \exists\ e\ such\ that\ score(e) = \alpha \\ and\ \alpha \geq score(e_F) \end{matrix} \right\}$$

where *score*(*e*) is the Tarantula suspiciousness metric computed for element *e* [2] (see Section II-A). In other words, S_1 is the set of distinct scores that are greater than or equal to that of the faulty element e_F. We consider e_F to be the element having the highest score among the elements that are associated with the faulty code. We assess the change in safety after cleansing as follows:

$|S_1(T\text{-}CCT)| < |S_1(T)| \Rightarrow$ safety got better

$|S_1(T\text{-}CCT)| > |S_1(T)| \Rightarrow$ safety got worse

$|S_1(T\text{-}CCT)| = |S_1(T)| \Rightarrow$ safety remained the same

4) Measure of *precision change:*

Precision indicates the reduction in the search space for the faulty code. We define it as the cardinality of the following set:

$$S_2(T) = \{e \mid score(e) \geq score(e_F)\}$$

S_2 is the set of elements whose score is greater than or equal to that of the faulty element e_F. We assess the change in precision after cleansing as follows:

$|S_2(T\text{-}CCT)| < |S_2(T)| \Rightarrow$ precision got better

$|S_2(T\text{-}CCT)| > |S_2(T)| \Rightarrow$ precision got worse

$|S_2(T\text{-}CCT)| = |S_2(T)| \Rightarrow$ precision remained the same

5) Measure of *coverage reduction*:

$$\frac{|\bigcup_{t \in T} E(t) - \bigcup_{t \in (T-CCT)} E(t)|}{|\bigcup_{t \in T} E(t)|}$$

This measure assesses the reduction in code coverage as a result of using our technique. No coverage reduction is desirable since it means that, when used in coverage-based testing, the resulting test suite is likely to be: a) no less effective due to equal coverage, b) more efficient due to its smaller size, and c) more effective due to its potential lack of coincidentally correct tests.

As a final note, we observed in our experiments a significant rate of redundancy among elements in terms of the coverage information they provide. We consider an element e_1 to be redundant if $\exists$ another element e_2 such that: $\forall\ t \in T$, t executes $e_1 \Leftrightarrow t$ executes e_2. To eliminate any bias that might be caused by redundant elements, we chose to discard them from our analysis.

IV. Experimental Work (Technique-I)

In our experiments we used three types of profiling elements, namely, basic blocks, branches, and def-use pairs. We used a tool that we developed as part of previous work [22] to collect the execution profiles of our Java subject programs. First, we describe our subject programs and associated seeded versions. Second, we present the averaged results for all seeded versions.

A. Subject Programs

Our experiments involved 18 seeded versions that are part of the Siemens test suite (*sir.unl.edu* [11]). Given that the profiling tool used in this study only targets Java programs, we opted to manually convert the Siemens programs to Java. Table 1 provides information about our seeded programs, including fault types, test suite sizes,

[2] Many fault localization techniques were presented in the literature, we based our metrics on Tarantula for its simplicity noting that its performance is not necessarily optimal

number of failures, number of correct tests, number of coincidentally correct tests (determined using our oracles), and percentage of coincidental correctness. Note that we discarded some seeded versions either because all associated failures resulted in exceptions being thrown, or they did not exhibit any coincidental correctness.

Table 1. Subject Programs

Program	Fault Type	$\|T\|$	$\|T_F\|$	$\|T_P\|$-$\|T_{CC}\|$	$\|T_{CC}\|$	$\|T_{CC}\|/\|T_P\|$
*print_tokens*_v1	wrong case	4070	6	3645	419	0.10
*print_tokens*_v2	added code	4070	48	3590	432	0.18
*print_tokens*_v5	missing assignment	4070	150	2670	1250	0.32
*print_tokens*_v6	constant mutations	4070	186	2953	931	0.24
*tot_info*_v4	altered conditional	1052	33	822	197	0.19
*tot_info*_v5	altered statement	1052	29	858	165	0.16
*tot_info*_v7	altered conditional	1052	123	923	6	0.01
*tot_info*_v9	altered statement	1052	37	831	184	0.18
*tot_info*_v11	altered statement	1052	199	824	29	0.03
*tot_info*_v12	altered return	1052	33	323	696	0.68
*tot_info*_v13	altered conditional	1052	128	917	7	0.01
*tot_info*_v15	altered conditional	1052	199	824	29	0.03
*tot_info*_v17	altered statement	1052	44	384	624	0.62
*tot_info*_v23	wrong initialization	1052	71	384	597	0.61
schedule_v2	altered statement	2650	210	1263	1177	0.48
schedule_v3	altered statement	2650	159	2345	146	0.06
schedule_v4	altered conditional	2650	294	1740	616	0.26
schedule_v8	deleted code	2650	31	2457	162	0.06

B. *Experimental Results*

This section presents the averaged results for all 18 seeded versions using the metrics described in Section III-B. Figure 3 plots the average percentages of false negatives and false positives against θ. As expected, a larger θ exhibits lower rates of false negatives but higher rates of false positives. Note how a $\theta \geq 0.7$ yields zero false negatives at the cost of a high rate of false positives. Specifically, for $\theta =$ 0.7, 0.8, and 0.9, the rates of false negatives are zero, but the rate of false positives are 76.9%, 88.9%, and 91%, respectively. Figure 4 plots the average percentages of code coverage reduction against θ. The code coverage reduction is not very significant even for high values of θ. For example, for $\theta =$ 0.7, 0.8, and 0.9, the average percentages of code coverage reduction are 2.1%, 2.7%, and 3%, respectively. Tables 2, 3, and 4 show that the safety of coverage-based fault localization improved for all 18 programs using all 3 values of θ. Table 2 shows that, for $\theta =$ 0.7, the precision of fault localization deteriorated for 16 programs and improved for only 2. Table 3 shows that, for θ = 0.8, its precision deteriorated for 17 programs and improved for only 1. Finally, Table 4 shows that, for $\theta =$ 0.9, the precision deteriorated for all 18 programs.

In summary, *Technique-I* used with $\theta \geq$ 0.7, yields no false negatives when identifying coincidentally correct tests and improves fault localization safety. On the other hand, it yields a high rate of false positives and worsens fault localization precision. Also, it does not seem to have an overly substantial impact on code coverage.

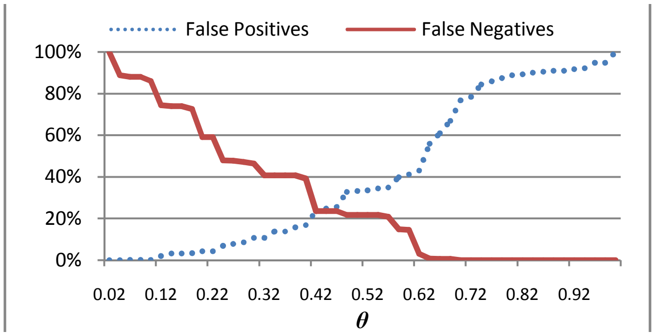

Figure 3. Cleansing accuracy w.r.t. θ

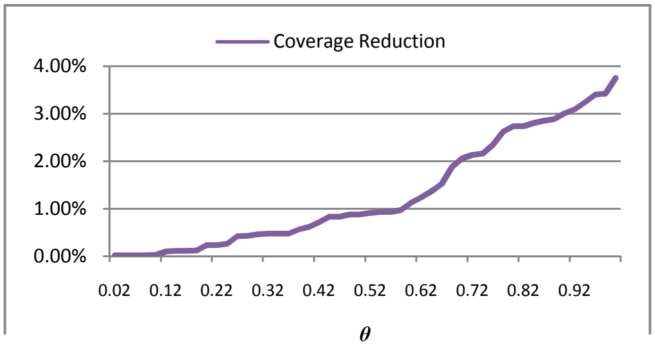

Figure 4. Coverage reduction w.r.t. θ

Table 2. Impact on the effectiveness of coverage-based fault localization (for $\theta = 0.7$)

Safety →

Precision ↑

	Worse	Same	Better
Better	0	0	2
Same	0	0	0
Worse	0	0	16

Table 3. Impact on the effectiveness of coverage-based fault localization (for $\theta = 0.8$)

Safety →

Precision ↑

	Worse	Same	Better
Better	0	0	1
Same	0	0	0
Worse	0	0	17

Table 4. Impact on the effectiveness of coverage-based fault localization (for $\theta = 0.9$)

Safety →

↑ **Precision**

	Worse	Same	Better
Better	0	0	0
Same	0	0	0
Worse	0	0	18

V. Refining the Base Technique

Technique-I is clearly very safe. In order to reduce its rate of false positives, i.e., to make it more precise, we explored extending it in two different ways leading to *Technique-II*, and *Technique-III* that we describe next.

A. *Technique-II*

This technique assumes that a cc_t containing a large number of cc_e's and/or cc_e's with a high average weight is more likely to be a coincidentally correct test than another that does not. For this purpose we will compute the Tarantula suspiciousness score for each cc_e, which would fall in the range]0.5, 1.0]. This score will be referred to as the weight of the cc_e. After applying *Technique-I* using a given θ, the resulting *CCT* is reduced as follows:

a) The cc_t's are ranked in descending order based on the following measure: ((average weight of the covered cc_e's) + (percent of cc_e's covered))

b) The lower ranked cc_t's are discarded

We applied *Technique-I* on all 18 programs using a θ = 0.7, 0.8, and 0.9 then ranked the cc_t's according to the measure. Figure 5 shows, for $\theta = 0.7$, the average rates of false negatives and false positives against the percentages of the top ranked cc_t's that were not discarded. For example, if the top 60% ($k = 60$) of cc_t's are considered, the average rate of false negatives and false positives would be 2.2% and 57.9%, respectively.

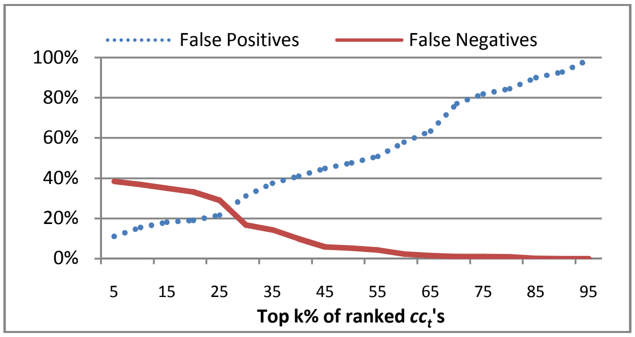

Figure 5. Cleansing accuracy w.r.t. top ranked cc_t's considered (using $\theta = 0.7$)

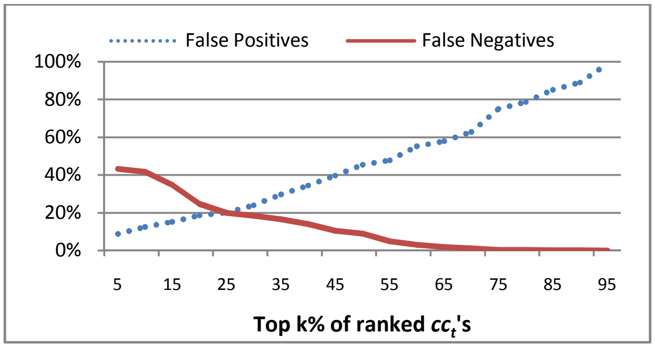

Figure 6. Cleansing accuracy w.r.t. top ranked cc_t's considered (using $\theta = 0.8$)

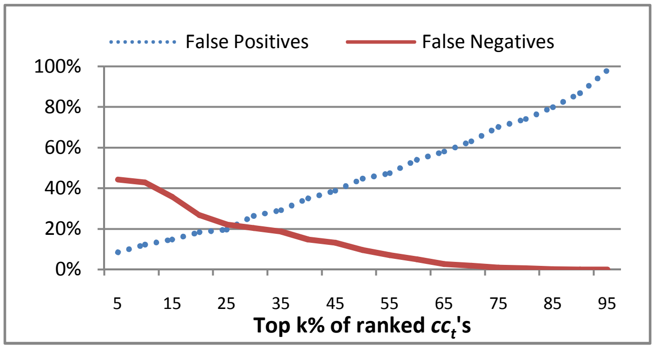

Figure 7. Cleansing accuracy w.r.t. top ranked cc_t's considered (using $\theta = 0.9$)

Figures 6 shows the results for $\theta = 0.8$. For $k = 60$, the average rate of false negatives and false positives is 3.0% and 55.2%, respectively. Finally, Figure 7 shows the results for $\theta = 0.9$. For $k = 60$, the average rate of false negatives and false positives is 5.1% and 54%, respectively.

The above results indicate that *Technique-II* is a slight but not decisive improvement over *Technique-I*. That is, by discarding the lower 40% (for example) of the ranked cc_t's, the rate of false positives might be considerably reduced at the expense of a minor increase in the rate of false negatives. Section V-C further contrasts the performance of *Technique-II* to *Technique-I* and presents results that indicate that it achieves considerable improvement over it.

B. *Technique-III*

This technique partitions the cc_t's into two clusters based on the similarity of the suspicious cc_e's (high weight cc_e's) they execute, then discards the cluster that is likely to contain most of the coincidentally correct tests. The conjecture is that coincidentally correct tests will cluster together because: a) typically, some cc_e's are relevant to the fault and others are not, and b) the coincidentally correct tests exercise these fault relevant cc_e's whereas the correct tests don't. After the clusters are identified, the issue then is which of the two clusters should be considered as the one containing the coincidentally correct tests. We make that selection based on the cluster that has the higher average weight of suspicious cc_e's, and in case of a tie, we choose the larger cluster. Note that when evaluating our selection scheme, we consider that

Table 5. Clustering and cluster selection results

Program		$\theta = 0.7$						$\theta = 0.8$						$\theta = 0.9$					
		Selection		# C	# CC	%FN	%FP	Selection		# C	# CC	%FN	%FP	Selection		# C	# CC	%FN	%FP
		expect	actual					expect	actual					expect	actual				
print_tokens_1	c_1		✓	3309	0	100	90.8	✓	✓	2757	306	27	75.6			1422	142	33.9	61
	c_2	✓		194	419					888	113			✓	✓	2223	277		
print_tokens_2	c_1			1758	0	0	1.1			1758	0	0	1.1			1758	0	0	1.1
	c_2	✓	✓	39	432			✓	✓	39	432			✓	✓	39	432		
print_tokens_5	c_1	✓	✓	0	1210	3.2	0			2380	0	0	0			2590	0	0	0
	c_2			7	40			✓	✓	0	1250			✓	✓	0	1250		
print_tokens_6	c_1	✓	✓	2237	576	38.1	75.8	✓	✓	2456	575	38.2	83.2	✓		2436	570	61.2	17.5
	c_2			488	355					497	356				✓	517	361		
tot_info_4	c_1		✓	346	0	100	42.1	✓	✓	574	197	0	69.8			292	0	0	64.5
	c_2	✓		438	197					248	0			✓	✓	530	197		
tot_info_5	c_1	✓	✓	760	165	0	88.6			330	0	0	61.5			277	0	0	67.7
	c_2			8	0			✓	✓	528	165			✓	✓	581	165		
tot_info_7	c_1	✓	✓	334	6	0	36.2	✓	✓	259	6	0	28.1	✓		619	6	100	24.9
	c_2			437	0					588	0				✓	230	0		
tot_info_9	c_1	✓	✓	268	184	0	32.3	✓	✓	274	184	0	33			204	0	0	66.7
	c_2			480	0					480	0			✓	✓	554	184		
tot_info_11	c_1			422	0	0	39			205	0	0	65.5			179	0	0	69.1
	c_2	✓	✓	321	29			✓	✓	540	29			✓	✓	569	29		
tot_info_12	c_1	✓		247	471	67.7	12.1			224	212	30.5	30.7			299	67	9.6	7.4
	c_2		✓	39	225			✓	✓	99	484			✓	✓	24	629		
tot_info_13	c_1	✓	✓	251	7	0	27.4	✓	✓	325	7	0	35.4	✓	✓	732	7	0	79.8
	c_2			547	0					513	0					111	0		
tot_info_15	c_1			422	0	0	39			205	0	0	65.5			179	0	0	69.1
	c_2	✓	✓	321	29			✓	✓	540	29			✓	✓	569	29		
tot_info_17	c_1			298	0	0	0			300	0	0	0			203	0	0	27.9
	c_2	✓	✓	0	624			✓	✓	0	624			✓	✓	107	624		
tot_info_23	c_1			298	0	0	0			300	0	0	0			203	0	0	27.9
	c_2	✓	✓	0	597			✓	✓	0	597			✓	✓	107	597		
schedule_v2	c_1			677	223	18.9	9.5	✓	✓	288	1172	0.4	22.8	✓	✓	346	1170	0.6	27.4
	c_2	✓	✓	120	954					785	5					778	7		
schedule_v3	c_1			655	0	0	44.8	✓	✓	1510	146	0	64.4	✓	✓	1510	146	0	64.4
	c_2	✓	✓	1051	146					527	0					708	0		
schedule_v4	c_1	✓	✓	905	616	0	52	✓	✓	907	616	0	52.1	✓	✓	905	616	0	52
	c_2			15	0					566	0					729	0		
schedule_v8	c_1		✓	917	10	93.8	37.3	✓	✓	1341	159	1.9	54.6	✓	✓	1400	159	1.9	57
	c_2	✓		1204	152					957	3					958	3		

it has succeeded if it selects the cluster with the larger number of coincidentally correct tests.

We applied *Technique-I* on all 18 programs using a θ = 0.7, 0.8, and 0.9 then used the *k-means* clustering algorithm to partition each of the resulting *CCT*'s into two clusters. Each cc_t was represented as a real vector of dimension $|CCE'|$, where *CCE'* is a subset of *CCE* containing the top 50% of cc_e's in terms of weight. The i^{th} position is set to 1 if the i^{th} cc_e is executed by the test case and set to 0 otherwise. We used the Euclidean distance as a metric and considered the two cc_t's having the highest separating distance to be the initial means. Table 5 presents the results for clustering and cluster selection. For each program and θ combination, it shows the number of correct tests (**#C**) and coincidentally correct tests (**#CC**) in each of the two identified clusters. It also shows the decision of the selection scheme (**actual**) compared to the expected selection decision (**expect**). The selection process failed in 6 out of 54 cases, as highlighted in the table; four times for θ = 0.7, and twice for θ = 0.9. Note how in 2 out these 6 cases the clustering outcome was rather poor to start with (*tot_info_12* and *print_tokens_6*). Also shown, are the rates of false negatives (**%FN**) and false positives (**%FP**). Note how when the cluster selection failed, the rates of false negatives are high, and are actually the worst in the result set. For example, in the case of *tot_info_4*, the rate of false negatives is 100%, whereas it would have been 0% if the correct cluster was selected.

Table 6. Comparative results summaries for all three techniques
(for *Technique-II*, the top 60% of the ranked tests were considered)

a)

False Negatives	***θ* = 0.7**				***θ* = 0.8**			***θ* = 0.9**			
	Tech-I	**Tech-II**	**Tech-III**	**Tech-III'**	**Tech-I**	**Tech-II**	**Tech-III**	**Tech-I**	**Tech-II**	**Tech-III**	**Tech-III'**
Max	0%	21.6%	100%	38.1%	0%	18.2%	38.2%	0%	18.2%	100%	38.8%
75th%	0%	0%	33.3%	2.4%	0%	2.4%	0.3%	0%	7.5%	1.5%	0.4%
Median	0%	0%	0%	0%	0%	0%	0%	0%	1.6%	0%	0%
25th%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Min	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Average	0%	2.2%	23.4%	5.5%	0%	3%	5.4%	0%	5.1%	11.5%	4.7%

b)

False Positives	***θ* = 0.7**				***θ* = 0.8**			***θ* = 0.9**			
	Tech-I	**Tech-II**	**Tech-III**	**Tech-III'**	**Tech-I**	**Tech-II**	**Tech-III**	**Tech-I**	**Tech-II**	**Tech-III**	**Tech-III'**
Max	96.1%	100%	90.8%	88.6%	100%	100%	83.2%	100%	100%	79.8%	82.5%
75th%	90%	68.7%	44.1%	51.2%	98.3%	65.5%	65.2%	99.2%	65%	66.1%	67.5%
Median	87%	61.5%	36.6%	37.5%	90.5%	59.7%	43.8%	93%	58.5%	54.5%	62.7%
25th%	74%	51.6%	10.1%	6.3%	85.4%	49.1%	24.1%	90.7%	48%	25.5%	27.8%
Min	0.3%	15.9%	0%	0%	50.1%	19.5%	0%	50.1%	19.5%	0%	0%
Average	76.9%	57.9%	34.9%	35%	88.9%	55.2%	41.3%	91%	54%	43.6%	49.6%

c)

Safety	***θ* = 0.7**				***θ* = 0.8**			***θ* = 0.9**			
	Tech-I	**Tech-II**	**Tech-III**	**Tech-III'**	**Tech-I**	**Tech-II**	**Tech-III**	**Tech-I**	**Tech-II**	**Tech-III**	**Tech-III'**
Better	18	16	15	15	18	16	16	18	16	16	17
Same	0	0	1	0	0	0	0	0	0	0	0
Worse	0	2	2	3	0	2	2	0	2	2	1

d)

Precision	***θ* = 0.7**				***θ* = 0.8**			***θ* = 0.9**			
	Tech-I	**Tech-II**	**Tech-III**	**Tech-III'**	**Tech-I**	**Tech-II**	**Tech-III**	**Tech-I**	**Tech-II**	**Tech-III**	**Tech-III'**
Better	2	9	11	11	1	10	11	0	10	13	13
Same	0	0	0	0	0	1	0	0	0	1	1
Worse	16	9	7	7	17	7	7	18	8	4	4

e)

Code Coverage Reduction	***θ* = 0.7**				***θ* = 0.8**			***θ* = 0.9**			
	Tech-I	**Tech-II**	**Tech-III**	**Tech-III'**	**Tech-I**	**Tech-II**	**Tech-III**	**Tech-I**	**Tech-II**	**Tech-III**	**Tech-III'**
Average	2.1%	1%	0.8%	0.4%	2.7%	0.8%	0.7%	3%	0.7%	0.5%	0.5%

C. Comparing Techniques I, II, and III

In this section we contrast the results of all three of our techniques and of a (contrived) variation of *Technique-III* in which we assume that the cluster selection was always a success, which we refer to as *Technique-III'*. In the results presented in Table 6, we considered the top 60% (k = 60) of the ranked tests for *Technique-II*.

Table 6-a) indicates that in regard to false negatives, *Technique-I* performed best, then *Technique-II* followed by *Technique-III'* and finally *Technique-III*. Table 6-b) shows that *Technique-III* generated the least number of false positives and *Technique-I* the most. Table 6-c) shows that the safety of fault localization improved best for *Technique-I*, then somewhat equally for the rest. Considering the precision of fault localization, *Technique-III* and *Technique-III'* performed best followed by *Technique-II*. Finally, code coverage reduction was not very significant for any of the techniques, and was the most significant for *Technique-I*.

To summarize, if a minor level of false negatives could be tolerated (< 10%), *Technique-III'* configured with θ = 0.7

should be considered as the best performer followed by *Technique-III* configured with $\theta = 0.8$. Since *Technique-III'* is basically *Technique-III* in which the cluster selection was always successful, we consider *Technique-III* to be our best performer, keeping in mind that improving the cluster selection will be our top priority in future work. We also consider *Technique-II* to be superior to *Technique-I* given that it generates much lower rates of false positives.

VI. Related Work

A coverage refinement approach is presented in [26] 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 after a context-pattern matching. Control and data flow information is needed to identify a matching. The paper lists context patterns for specific types of fault such as: missing function calls and missing assignments. It also presents results of experiments that showed that coincidental correctness is a common problem and harmful for coverage-based fault localization.

In [9] the issue of coincidental correctness was covered in the context of black box testing, specifically, partition analysis and boundary value analysis. The work studies the cases where shifts in boundaries got undetected due to coincidental correctness. Coincidental correctness may take place in boundary value analysis whenever two functions of two neighbor domains return the same result when fed with the same input and thus restraining the discovery of boundary shifts. To solve the problem the paper suggests conditions under which the tests input should be selected. As a general condition to reduce coincidental correctness, the test cases generated should always be able to produce unique output for every domain function.

The paper in [4] proposes a new concept called the testing-for-diagnosis (TFD) criterion that aims at reducing the number of test cases available in a certain test suite. This reduction should not affect the coverage capability that was offered by the initial test suite but should elevate the diagnosis accuracy. As a result, the test suite minimization leads to an improvement in the efficiency without reduction in the effectiveness of the diagnosis strategy (fault localization technique). A new type of coverage was introduced to satisfy the TFD criterion. The new coverage is called the dynamic basic block coverage where a dynamic basic block is defined as the block of statements that are covered by the same test cases. However, unlike other types of coverage, the maximum number of DBBs in a certain program cannot be statically known in advance, so a bacteriologic algorithm was used for generating test cases. Although the experiments conducted in the paper supported the relevance of DBBs to diagnosis accuracy, further investigation is still needed especially that it is thought that the effectiveness of this approach was compromised due to coincidental correctness as suggested by the authors.

A fault-based framework for the generation of test cases was described in [24]. The basic idea of the tool is the definition of a criterion that covers conditions under which an initial potential failure will be revealed without considering the issue of propagation. The tool requires the creation of classes to describe the propagation for specific types of errors. Those classes are used to identify what type of input is capable of stimulating the occurrence of potential failure (failing state that can be uncovered using a state oracle). However, a potential failure does not become an actual failure unless its effect propagates to the output thus the tool is unable to prevent coincidental correctness. As a result the tool provides necessity conditions for error detection without supplying the sufficiency conditions.

In the work presented in [17], which is based on [10], the authors described coincidental correctness as a problem that occurs whenever the weak mutation hypothesis (*WMH*) is not holding. *WMH* states that whenever a fault was executed and its effect is detectable at the fault location then the output will be affected. The work was primarily an empirical study to identify how often *WMH* holds. On the other hand, the work in [25] focused on identifying points in the tested code that are likely to develop the type of undetected errors that lead to coincidental correctness. The PIE analysis presented in the paper investigates three factors related to faulty elements in the tested program: fault execution, creation of faulty states (infection), and failure propagation to the output. Fault execution analysis can be done by studying the probability of execution of certain locations in the code based on a predefined set of input. Infection can be analyzed based on a pre-estimation for a fault behavior. Finally, the propagation of fault can be studied by injecting failure states and then studying their propagation estimate.

VII. Conclusions and Future Work

We presented variations of a technique that, given a test suite in which each test has been categorized as failing or passing, identify the subset of passing tests that are likely to be coincidentally correct. Using our better performing technique and configuration (*Technique-III*, $\theta = 0.8$), the rate of false negatives averaged 5.4%, the rate of false positives averaged 41.3%, the safety of fault-localization was improved for 88% of the programs, its precision improved for 61% of the programs, and code coverage reduction averaged 0.7%.

Finally, as part of future work we intend to conduct a more comprehensive empirical study and explore the following:

1) Improve cluster selection in *Technique-III.*

2) Improve the clustering process in *Technique-III* in order to further reduce the rate of false positives.

3) Address the case when multiple defects are present by exploring the following: a) use cluster analysis to partition T_F into subsets where each subset would contain failures with similar execution profiles; i.e., failures that are likely to be induced by the same defect, b) apply our current techniques on the union of each subset and T_P.

4) Reduce *CCT* by keeping only the cc_t's that execute a relative high number of cc_e's that are at the source of zero-strength flows to the output (see Section II-C). The conjecture is that if a cc_e actually causes an infectious state, and it is at the source of a zero-strength flow to the output, then the infectious state would likely not propagate to the output, thus leading to coincidental correctness.

5) Assess the impact of cleansing coincidental correctness on other fault localization approaches [1][2][4][5][6][8][14][15][16][18], and on software testing techniques such as regression testing.

ACKNOWLEDGMENT

This research was supported in part by NSF grant no. 0819987 and ASTF.

REFERENCES

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

[2] Agrawal H., Horgan J., London S. and Wong W. Fault localization using execution slices and dataflow sets. IEEE International Symposium on Software Reliability Engineering, ISSRE, pp. 143-151, 1995.

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

[4] 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.

[5] Cleve H. and Zeller A. Locating causes of program failures. In Proceedings of the International Conference on Software Engineering, pages 342-351, St. Louis, Missouri, May 2005.

[6] Dallmeier V., Lindig C., Zeller A. Lightweight Bug Localization with AMPLE. International Symposium on Automated Analysis-Driven Debugging , AADEBUG, pp. 99-103, 2005.

[7] 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.

[8] Denmat T., Ducass´e M. and Ridoux O. Data Mining and Crosschecking of Execution Traces. Int'l Conf. Automated Software Eng., ASE, pp. 396-399, Long Beach, CA, 2005.

[9] 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.

[10] Howden, W. E. 1982. Weak Mutation Testing and Completeness of Test Sets. IEEE Trans. Softw. Eng. 8, 4 (Jul. 1982), 371-379.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Liu C., Fei L., Yan X., Han J. and Midkiff S. Statistical Debugging: A Hypothesis Testing-based Approach. IEEE Transaction on Software Engineering, Vol. 32, No. 10, pp. 831-848, Oct., 2006.

[17] B. Marick, The Weak Mutation Hypothesis, In Proc. of ISSTA'91, Page 190-199, Oct., 1991.

[18] Masri, W. Fault Localization Based on Information Flow Coverage. Software Testing, Verification and Reliability, To appear 2009.

[19] Masri W., Abou-Assi R., El-Ghali M., and Fatairi N. An Empirical Study of the Factors that Reduce the Effectiveness of Coverage-based Fault Localization. International Workshop on Defects in Large Software Systems, DEFECTS, Chicago, IL, 2009.

[20] Masri W., Podgurski A. 2006. An Empirical Study of the Strength of Information Flows in Programs. Fourth International Workshop on Dynamic Analysis (WODA 2006), Shanghai, China, May 2006.

[21] Masri, W. and Podgurski, A. Measuring the Strength of Information Flows in Programs. ACM Transactions on Software Engineering and Methodology (TOSEM). Vol. 19, issue 2, October 2009.

[22] W. Masri, A. Podgurski and D. Leon. "An Empirical Study of Test Case Filtering Techniques Based On Exercising Information Flows". IEEE Transactions on Software Engineering, July, 2007, vol. 33, number 7, page 454.

[23] 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.

[24] D, J. Richardson, and M.C. Thompson, An Analysis of Test Selection Criterria Using the RELAY Model of Fault Detection, IEEE TSE,19(60):533-553, 1993.

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

[26] 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.