Evaluation of Measures for Statistical Fault Localisation and an Optimising Scheme*

David Landsberg[2], Hana Chockler[1], Daniel Kroening[2], and Matt Lewis[2]

[1] King's College London
[2] University of Oxford

Abstract. *Statistical Fault Localisation* (SFL) is a widely used method for localizing faults in software. SFL gathers coverage details of passed and failed executions over a faulty program and then uses a measure to assign a degree of *suspiciousness* to each of a chosen set of program entities (statements, predicates, etc.) in that program. The program entities are then inspected by the engineer in descending order of suspiciousness until the bug is found. The effectiveness of this process relies on the quality of the suspiciousness measure. In this paper, we compare 157 measures, 95 of which are new to SFL and borrowed from other branches of science and philosophy. We also present a new measure optimiser Lex_g, which optimises a given measure g according to a criterion of single bug optimality. An experimental comparison on benchmarks from the Software-artifact Infrastructure Repository (SIR) indicates that many of the new measures perform competitively with the established ones. Furthermore, the large-scale comparison reveals that the new measures Lex_{Ochiai} and Pattern-Similarity perform best overall.

1 Introduction

Software engineers use fault localization methods in order to focus their debugging efforts on a subset of program entities (such as statements or predicates) that are most likely to be causes of the error. Since the attempts to reduce the number of faults in software are estimated to consume $50 - 60\%$ of the development and maintenance effort [5], accurate and efficient fault localization techniques have the potential to greatly reduce the overall effort of software development.

In statistical fault localisation (SFL), statistical information on passing and failing executions of a faulty program is gathered and analysed [1, 3, 13–15, 32]. Based on the resulting data, SFL assigns a degree of suspiciousness to each member of a chosen set of program entities of the program under test. Essentially, the degree of suspiciousness depends on the number of appearances of this entity in the passing and failing executions. There are many approaches to computing this degree, and naturally, entities that cause the error are hoped to have the highest degree of suspiciousness. The program entities are inspected by the user in descending order of suspiciousness until the bug is found. SFL has been considered a highly effective and efficient way for localising faults in software [30].

* Supported by UK EPSRC EP/J012564/1 and ERC project 280053.

Our contributions The contributions of this paper are as follows:

1. We introduce and motivate 95 new measures (borrowed from other areas of science and philosophy) to SFL. These measures are divided into five categories: similarity, prediction, causation, confirmation and custom.
2. We formally prove that over 50 measures are equivalent to others for the purpose of ranking suspicious entities.
3. We experimentally compare the measures on the Siemens test suite along with five larger programs: space, grep, gzip, sed and flex[3]. We show that many of the new measures perform competitively, with an optimised version of PatternSimilarity outperforming all pre-existing SFL measures on the benchmarks.
4. We introduce a new measure-optimising scheme Lex_g and show Lex_{Ochiai} outperforms all other measures on the benchmarks.

Along with providing two new best performing measures, to the best of our knowledge, this research provides one of the largest scale SFL studies to date in three ways. Firstly, it contains the largest experimental study over C programs in SFL, consisting of the largest number (and largest sized) C programs. Secondly, it introduces and compares the largest number of measures. Thirdly, it contains results for the largest number of ranking equivalence proofs (see [20]).

Related work Research in SFL is largely driven by the construction or introduction of new suspiciousness measures. Experimental results assess the quality of measures by applying them to known benchmarks [1, 2, 16, 18, 20, 24, 25, 29]. Theoretical results have included formal properties and equivalence proofs of different measures [19, 20, 30].

A similar paper to ours is the paper by Lucia et al. [18], which compares association measures on C and Java Programs. However, Lucia et al. conclude that there is no measure which is clearly the best, whereas we show our new measure Lex_{Ochiai} is a robust overall top performer. Another similar paper is the paper by Naish [20], who set the standards for proving equivalences between suspiciousness measures, discuss optimal measures and compare a (smaller) set of measures against a (smaller) set of benchmarks.

A recent paper by Yoo et al. [31] analyses fault localisation in conjunction with prioritisation. The problem studied in [31] is deciding the best course of action when a fault is found. Their approach is complementary to ours (and applied to a similar set of benchmarks) and can also be applied in conjunction with the measure we construct in this paper.

Xie et al. develop a theoretical approach to proving that some measures are better than others [30]. However, their proof relies on several critical simplifying assumptions (in particular, the bug must be contained in a single line of code or block). Bugs that are more realistic often break this assumption, invalidating the proof on realistic examples. We impose no such theoretical restrictions.

[3] From the Software Artifact Infrastructure Repository at `http://sir.unl.edu`.

On a more general level, Parnin et al. [21] raise the question of whether fault localisation techniques are useful at all. The paper compared the efficiency of fault localisation with and without the automated tool Tarantula [13]. They reported that experts are faster at locating bugs using the tool for simple programs, but not for harder ones. However, their study is limited in its ability to generalize, as their experiments included only two small, single-bug programs (Tetris and NanoXML, 2K/4K LOC respectively) and is limited to the Tarantula tool.

Paper structure The rest of the paper is organised as follows. In Section 2, we present the informal ideas and formal definitions of SFL and discuss the 62 previously used suspiciousness measures. In Section 3, we discuss 95 measures that have not yet been applied to SFL and demonstrate that these new measures are well suited to SFL. We briefly outline proofs of equivalence of many of these measures when applied to SFL. Section 4 presents the experimental results of applying the non-equivalent measures to the benchmarks. We summarize our results in Section 5. Due to lack of space, the complete ranking equivalence proofs, results tables and tables containing definitions of measures are only in the extended version of this paper. The extended paper, the data set and the code used to perform the experiments are available from `http://www.cprover.org/sfl/`.

2 Definitions and Notations

In this section, we introduce the basic definitions and notations of SFL, and survey the established measures. We also present a small motivating example.

2.1 Definitions

Let a program under test (PUT) $\mathbf{P}$ be an ordered set of *program entities*, such that $\mathbf{P} = \langle C_1, \ldots, C_n \rangle$, where $n \in \mathbb{N}$. Program entities can be statements, branches, paths, or blocks of code (see, for example, [14, 16, 28]). Let a test suite $\mathbf{T}$ be an ordered set of test cases $\mathbf{T} = \langle t_1, \ldots, t_m \rangle$, where m is the size of the test suite. Each test case t_i is a Boolean vector of length n (where n is the number of program entities) such that $t_i = \langle b_1^i, \ldots, b_n^i \rangle$, where $b_j^i \in \{0, 1\}$, where we have $b_j^i = 1$ iff C_j is covered by t_i. We represent each program entity C_i by the set of test cases where C_i is 1. The last program entity C_n is the error statement E, which is 1 if the test case *fails* and 0 if it *passes*. A convenient way to store this information is using *coverage matrices*, in which the i-th row of the j-th column represents whether test case t_i covers program entity C_j, an example of which is given in Table 1.

For each program $\mathbf{P}$, test suite $\mathbf{T}$ and program entity C_i we can construct this program entity's *contingency table* [23]. This table can be symbolically represented as a vector of four elements denoted as $\langle a_{ef}^i, a_{ep}^i, a_{nf}^i, a_{np}^i \rangle$, where a_{ef}^i is the number of failing test cases in $\mathbf{T}$ that cover C_i, a_{ep}^i is the number of passing

test cases in $\mathbf{T}$ that cover C_i, a^i_{nf} is the number of failing test cases in $\mathbf{T}$ that do not cover C_i and a^i_{np} is the number of passing test cases in $\mathbf{T}$ that do not cover C_i. For each program entity, we can calculate its contingency table for a test suite. See Table 2 for an example. We let $F_i = a^i_{ef} + a^i_{nf}$, $P_i = a^i_{ep} + a^i_{np}$ and $T_i = F_i + P_i$. For each test suite and C_i and C_j, $F_i = F_j$ and $P_i = P_j$. When the context is clear we drop numerical indices, writing, for instance, C and a_{ef} instead of C_i and a^i_{ef}.

A *suspiciousness measure* m maps a contingency table $\langle a^i_{ef}, a^i_{ep}, a^i_{nf}, a^i_{np}\rangle$ to a real number [20]. Roughly speaking, for a test suite and faulty program, the higher the output of the measure the more suspicious the program entity C_i is assumed to be with respect to containing a bug. The output of each suspiciousness measure is the *suspiciousness score* that is assigned to each program entity (we also say that a program entity is *ranked* according to its suspiciousness score). The program entities are then ordered according to their degree of suspiciousness and are investigated in descending order by the user until the bug is found.

A probability space for each test suite is defined as follows. Given a program $\mathbf{P}$ and test suite $\mathbf{T}$, we identify a probability space (Ω, S, Pr), where the sample space $\Omega = \{t_1, \ldots, t_n\}$ is the set of test cases, the set of events S is the power-set of the set of program entities, where Pr: $S \to [0, 1]$ is a probability function with the usual signature. Assuming the axioms and language of classical probabilistic calculus and given the definitions of a^i_{ef}, a^i_{nf}, a^i_{ep}, a^i_{np} above, we can identify $Pr(C_i \cap E)$, $Pr(\neg C_i \cap E)$, $Pr(C_i \cap \neg E)$ and $Pr(\neg C_i \cap \neg E)$ with $\frac{a^i_{ef}}{T_i}$, $\frac{a^i_{ep}}{T_i}$, $\frac{a^i_{np}}{T_i}$ and $\frac{a^i_{nf}}{T_i}$ respectively. Using probabilistic calculus, this is sufficient to generate the other probabilistic expressions we need. Probabilistic expressions may also be translated into algebraic form in the obvious way. For example, $P(E|C_i)$ is equal to $\frac{a^i_{ef}}{a^i_{ep}+a^i_{ef}}$.

Naish's notion of *single-bug optimality* [19] is based on the observation that if a program contains only a single bug, then all failing traces cover that bug. Formally, a measure m is *single-bug optimal* if (1) when $a_{ef} < F$, the value returned is less than any value returned when $a_{ef} = F$ and (2) when $a_{ef} = F$ and $a_{np} = k$, the value returned is greater any value returned when $a_{np} < k$ [19].

We use Naish's notion of *ranking equivalence* between suspiciousness measures, defined as follows. Two suspiciousness measures m_1 and m_2 are said to be *monotonically equivalent* if $(m_1(\boldsymbol{x}) < m_1(\boldsymbol{y})) \Leftrightarrow (m_2(\boldsymbol{x}) < m_2(\boldsymbol{y}))$ for all vectors $\boldsymbol{x}$ and $\boldsymbol{y}$. Many suspiciousness measures turn out to be monotonically equivalent on domains in which the measures share the same program and test suite [20]. In other words, they are monotonically equivalent on domains in which the number of failing test cases F and the number of passing test cases P is the same for the vectors $\boldsymbol{x}$ and $\boldsymbol{y}$. This property is called *ranking equivalence* [20].

2.2 Motivating Example

In this section, we present a simple motivating example to illustrate a typical instance of SFL. Consider the faulty program *minmax.c* in Figure 1 (taken from [11]). The program has six program entities $\langle C_1, C_2, C_3, C_4, C_5, E\rangle$, where E is the specification. The program fails to satisfy the specification $least \leq most$.

```
int main () { // C1
  int inp1, inp2, inp3;
  int least = inp1;
  int most = inp1;

  if (most < inp2)
    most = inp2;  // C2
  if (most < inp3)
    most = inp3; // C3
  if (least > inp2)
    most = inp2; // C4 (Bug!)
  if (least > inp3)
    least = inp3; // C5
  assert(least <= most);
  // E (Specification)
}
```

Fig. 1: minmax.c

The reason for the failure is the bug at C_4, which should be an assignment to *least* instead of an assignment to *most*. To locate the fault, we collected coverage data from ten test cases t_1 to t_{10}. Three of them fail and seven pass. The coverage matrix for these test cases is given in Table 1. We compute contingency tables for each program entity using the coverage matrix and give the table for C_4 as an example (Table 2). We then apply a suspiciousness measure to assign a degree of suspiciousness to each of the program entities. We use the Wong-II measure [29] $a_{ef} - a_{ep}$ as a simple example.

	C_1	C_2	C_3	C_4	C_5	E
t_1	1	0	1	1	0	1
t_2	1	0	0	1	1	1
t_3	1	0	0	1	0	1
t_4	1	1	0	0	0	0
t_5	1	0	1	0	0	0
t_6	1	0	0	0	1	0
t_7	1	0	0	1	1	0
t_8	1	0	0	0	0	0
t_9	1	1	0	0	1	0
t_{10}	1	1	1	0	0	0

Table 1: Coverage matrix for minmax.c

	E	$\neg E$
C_4	3	1
$\neg C_4$	0	6

Table 2: Contingency table for C_4

The user then investigates the program entities in descending order of suspiciousness until the fault is found (ignoring E). In this example, Wong-II ranks C_4 the highest with a score of 2 and thereby successfully identifies the bug within the most suspicious program entity.

2.3 Established Measures

We include 62 measures selected by Naish [20] and Lo [18] in our comparison. To motivate many of these measures to SFL, Naish et al. discuss desirable formal properties [19]. One important property that has been discussed is *monotonicity*: for a fixed number of passed and failed tests, a measure should strictly increase as a_{ef} increases and strictly decrease as a_{ep} decreases [13,19,30]. Examples of some prominent measures from [20] are Wong-I $= a_{ef}$, Wong-II $= a_{ef} - a_{ep}$, Naish $= a_{ef} - \frac{a_{ep}}{a_{ep}+a_{np}+1}$, Zoltar $= \frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+\frac{10000 a_{nf} a_{ep}}{a_{ef}}}$, Jaccard $= \frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$, Ochiai $= \frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$ [1], Tarantula $= \frac{a_{ef}}{a_{ef}+a_{np}}$ [13][4], Kulczynski-II $= \frac{1}{2}(\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ef}}{a_{ef}+a_{ep}})$, and M2 $= \frac{a_{ef}}{a_{ef}+a_{np}+2(a_{nf}+a_{ep})}$.

3 New SFL Measures

In this section we introduce 95 new suspiciousness measures that have not yet been applied in SFL. We organise them into five different groups: *similarity*, *predictive*, *causal*, *confirmation* and *custom* measures. Their application to SFL is motivated in terms of different proposed criteria about what a suspiciousness measure should exactly capture. We discuss such criteria at the beginning of each paragraph and the reader is referred to the full paper for the definitions of the new measures. We identify over 50 measures which are ranking equivalent and summarise interesting monotonic simplifications of some measures. The introduction of many measures has the benefit of consolidating the results concerning top performing measures. Furthermore, we introduce a new measure optimiser, which we later show can be used to construct the best performing measure on our benchmark suite.

3.1 New SFL measures from the literature

Similarity measures. The first proposed criterion is that a suspiciousness measure should measure how similar a program entity C is to the error E. This motivates the use of similarity measures in SFL and has been discussed in the literature [19, 20]. Indeed, many of the measures of the previous section are similarity measures (such as Jaccard) that were originally used in different domains. The new similarity measures we include in our experiments are available in the survey of [4] and are as follows: 3w-Jaccard, Baroni-Urbani-Buser-I and II, Braun-Blanquet, Bray-Curtis, Cosine, Cole, Chord, Dennis, Dispersion, Driver&Kroebner, F&M, Faith, Forbes-I, Forbes-II, Fossum, Gower, Gower-Legendre, Hellinger, Johnson, Lance-Williams, MCconnaughey, Michael, Mountford, Nei-Li, Otsuka, PatternSimilarity, ShapeSimilarity, SizeSimilarity, Vari, Simpson, Sorgenfrei, and Sokal&Sneath-I, II, III, IV, V and Tarwid.[5] The

[4] Strictly, this is an algebraic simplification of the original Tarantula measure.

[5] In some cases, a distance measure m has been converted to a similarity measure for our purposes, using the convention of $-$distance $\equiv$ similarity.

measure PatternSimilarity $= -\frac{4(a_{ep}a_{nf})}{(a_{ef}+a_{ep}+a_{nf}+a_{np})^2}$, which is used in clustering [4], is of particular interest and we discuss it later in more detail.

Prediction measures. The second proposed criterion is that a suspiciousness measure should measure the degree by which the execution of a program entity C *predicts* the error E. This motivates the use of what we loosely call *prediction* measures. Many of these measures are commonly used in epidemiology and diagnosis to estimate how well a test result predicts a disease or successful treatment [8, 10]. The prediction measures we include in our experiments are as follows: Positive predictive value (PPV) $= P(E|C)$, Negative predictive value (NPV) $= P(\neg E|\neg C)$, Sensitivity $= P(C|E)$, Specificity $= P(\neg C|\neg E)$, Youden's J, Positive Likelyhood, Tetrachoric, Relative risk, Z-ratio, Peirce, Pearson-I, II and III [23], Pearson-Heron I and II, Anderberg-II, Tanimoto, Mutual Info, Simpson, Gilbert&Wells and Goodman&Kruskal (see the surveys [4, 8, 10, 26]).

Causal measures. The third proposed criterion is that a suspiciousness measure should measure the degree by which the execution of a program entity C has the power to cause the error E. This motivates the use of measures of causal power/strength to SFL. Such measures are principally found in the domain of philosophy of science [9] and artificial intelligence [22] and many of their formal properties have been shown [9]. Examples of causal measures are Suppes $= P(E|C) - P(E|\neg C)$, Eels $= P(E|C) - P(E)$, Lewis $= \log \frac{P(E|C)}{P(E|\neg C)}$, Fitelson $= \log \frac{P(E|C)}{P(E)}$ [9]. The other causal measures considered are: Pearl-I, II, III and IV Fitelson II and III Korb I, II and III, Cheng and Good.

Confirmation Measures. The fourth proposed criterion is that a suspiciousness measure should measure the degree by which the execution of a program entity H is a hypothesis which explains the error E. This motivates the use of measures of explanation (sometimes called evidential/inductive/confirmation measures) to the domain of SFL. Such measures have been developed in the domain of philosophy of science [12] and many of their formal properties have been proven [7]. Some example confirmation measures are Earman $= P(H|E) - P(H)$, Joyce $= P(H|E) - P(H|\neg E)$, Milne $= \log \frac{P(H|E)}{P(H)}$, and Good-II $= \log \frac{P(H|E)}{P(H|\neg E)}$ [12]. The other measures considered are Carnap-I and II, Crupi, Rescher, Kemeny, Popper-I, II, and III, Levi, Finch-I, Gaifman and Rips.

3.2 Ranking equivalent measures

Naish proved that many different suspiciousness measures are in fact equivalent for the purposes of ranking suspiciousness entities [20]. We extend Naish's work by providing many of the remaining equivalence proofs (over 50) for the measures in this paper (see the full paper for the proofs). Proving ranking equivalences is essential in determining a maximal set of inequivalent measures to investigate and allows us to ignore the remainder in experimentation. Furthermore, using equivalence proofs we can find some elegant monotonic simplifications which

identify the underlying "essence" of some of the new suspiciousness measures, which may be used to guide future development. For instance, of our new measures (established measures are bracketed), we have found that Sensitivity is ranking equivalent to a_{ef} (as is Wong-I), Specificity to a_{np}, PPV to $\frac{a_{ef}}{a_{ep}}$ (as is Tarantula), NPV to $\frac{a_{np}}{a_{nf}}$, YulesQ to $\frac{a_{ef}a_{np}}{a_{ep}a_{nf}}$, F1 to $\frac{a_{ef}}{a_{ep}+a_{nf}}$ (as is Jaccard), SizeSimilarity to a_{ef}-a_{ep} (as is Wong-II) and PatternSimilarity to $-a_{nf}a_{ep}$.

3.3 A new custom measure

We propose the fifth criterion for suspiciousness measures: that a measure should be tailored to particular features concerning software errors (similar ideas had been proposed in [20] and Wong-III [29]).

We motivate our measure as follows. Firstly, following work by Naish [20], we state that our measure should be *single bug optimal* as defined in Section 2.1, because of deeper reasoning pertaining to Occam's razor. That is, we think the simplest hypothesis for explaining the error should be investigated first, and as the simplest hypothesis is that the program contains a single bug, the measure should be single bug optimal.

Secondly, although we state that the measure should be single bug optimal, we diverge from Naish [20] insofar as we do not make the *single fault assumption* – that the program contains only a single bug. This is because there exist programs with multiple bugs and our goal is to construct a measure that provides a complete solution to the problem of SFL. Consequently, it still remains to work out how to rank the suspiciousness of entities when no bug is covered by all bad traces. In this case, the suspiciousness of each program entity is determined by an existing measure g which is chosen on its ability to deal with multiple bugs and by success in experimentation. We account for this in the second condition of our measure below. For a measure g, we define our *measure optimising scheme* Lex_g as follows. Let $\boldsymbol{x}$ be the vector $\langle a_{ef}, a_{ep}, a_{nf}, a_{np} \rangle$. Then,

$$Lex_g(\boldsymbol{x}) = \begin{cases} a_{np} + 2 & \text{if } a_{ef} = F \\ g(\boldsymbol{x}) & \text{otherwise.} \end{cases} \tag{1}$$

In Equation 1, g stands for an internal measure, and can represent any measure appropriately scaled from 0 to 1. The intuition behind our optimising scheme is that g should rank each program entity in terms of suspiciousness as it usually does, except in the case where that entity is covered by every failing trace, in which case it should be investigated by the user as a matter of top priority. Based on its performance in our experiments, we choose Ochiai as the internal measure g, hence called Lex_{Ochiai}.

The following theorem states the optimality of our scheme (see the full paper for the proof).

Theorem 1. *Lex_g is single bug optimal.*

Lex_g can be considered as an optimising scheme which "converts" an appropriately scaled measure g into a single-bug optimal measure. The name of our

scheme is derived from its underlying idea — to lexically order two different classes of entities in terms of suspiciousness.

4 Experiments

In this section we describe the results of empirical evaluation of the measures. First we describe the experimental setup; then, we discuss our two means of assessment – an average scoring method and a Wilcoxon rank sum significance test. We conclude the section with the presentation and the analysis of the results.

Program	Vs	LOC	TC	FTC	PE	FPV	Program	Vs	LOC	TC	FTC	PE	FPV
tcas	41	173	1608	38	53	1.61	**replace**	32	563	5542	96	218	1.79
schedule	9	410	2650	80	146	1.45	**gzip**	3	7996	214	126	1223	3.33
schedule2	10	307	2710	27	126	1.10	**space**	38	9126	13585	1439	976	5.21
tot_info	23	406	1052	83	116	1.04	**sed**	2	11990	360	210	2378	5.50
print_tks	7	563	4130	69	179	1.14	**grep**	3	13229	750	304	1785	17.33
print_tk2	10	508	4115	206	196	1.00	**flex**	2	14230	567	71	3092	29.50

Table 3: Table of benchmarks

4.1 Experimental setup

The benchmarks are listed in Table 3. For each program, the table specifies the number of faulty versions (Vs), the number of lines of code in the original version of the program (LOC), the number of test cases (TC), the average number of failing test cases per version (FTC), the number of program entities of the original version of the program (PE), and the average number of faulty lines of code per version (FPV)[6].

The benchmarks are obtained from the Software Information Repository [6]. The versions of sed, grep, flex, and gzip used are the same as ones used in Lo [17], the versions of the Siemens and space test suites are the same as the ones used in Naish [20]. The Siemens test suite consists of tcas, schedule, schedule2, totinfo, print tokens, print tokens2 and replace. Overall, the experimental setup consists of 180 program versions, with over a million lines of code in total, and an average of 2.88 buggy lines per version.

The Siemens test suite is a widely used set of benchmarks in the domain of SFL [1, 2, 16, 24, 25, 29]. Space was additionally included by Naish [20]. The second set of benchmarks, consisting of versions of *gzip*, *grep*, *flex* and *sed*, has been used to assess SFL in [17]. In this paper, we demonstrate experimental

[6] Note that some program versions contain no faults. This happens when the fault appears in a non-executable line of code, such as a macro definition. These versions are removed from the experiment following [20].

results on the union of the sets of benchmarks used in these papers, making our evaluation, to the best of our knowledge, the largest set of C programs used to evaluate measures for SFL. Moreover, the set of measures in our evaluation is, to the best of our knowledge, the largest set of measures ever compared over any set of benchmarks in SFL.

Each test case was executed for each of the faulty programs and the result (pass or fail) recorded together with the set of the lines of code that were executed during this test (this data was extracted using GCOV). The pass or fail result was decided based on the output of the program and its comparison with the original program on the same input. Crashes were recorded as failures. The collected coverage data was used as an input to the measures, which assigned a suspiciousness score to each program entity (statements) in the (mutant) program and sorted the lines of code in the descending order of suspiciousness. To assign a score, we added a small *prior constant* (0.5) to each cell of each program entity's contingency table in order to avoid divisions by zero, as is convention [20].

We experimented over a range of different prior constants (PC) in between 0 to 1, and did not discover any significant or noteworthy differences in results. The exception was for the PatternSimilarity measure (for which we used the ranking equivalent measure $-a_{nf}a_{ep}$ in our implementation). We discovered that this measure was optimised if we set the PC to $a_{nf} = 0.1$ and $a_{ep} = 0.5$. The optimised version of PatternSimilarity is henceforth called PattSim2, and the unoptimised PattSim1.

4.2 Methods of assessment

We use two means of assessment: an average scoring method and a Wilcoxon rank-sum significance test. We discuss the details here.

To score how well a measure performs on a benchmark, we introduce the *best*, *worst* and *average* scoring methods [1, 20, 29]. Formally, where m is a measure, n is the number of program entities in the program, b is a bug with the highest degree of suspiciousness of any bug and *bugs* is the number of faulty lines in the program, we define $best(m) = (|\{x|m(x) > m(b)\}|/n) \times 100$ and $worst(m) = (|\{x|m(x) > m(b)\}|/n) \times 100$ and $avg(m) = best(m) + ((worst(m) - best(m))/(bugs + 1))$. For our evaluation we use the *avg* scoring method [20], which gives us the percentage of non-buggy program entities which we'd expect an engineer to examine before locating a bug, given the number of faulty lines in the program. To get the *avg* score for a benchmark, we take the mean *avg* of the scores of all the versions in that benchmark. To get the overall *avg* score, we take the mean of the 12 benchmark scores.

We performed a second means of assessment using Wilcoxon rank-sum tests. The Wilcoxon rank-sum test is a non-parametric statistical test which tests whether one population of values is significantly larger than another population [27]. Using this test, we were able to establish which measures were significantly better than others, by comparing each measure's 12 average scores for

each benchmark.[7] To establish a baseline for localisation efficiency, we included a measure (Rand) which assigns each program entity a random suspiciousness score.

4.3 Results

Name	Score	Name	Score	Name	Score	Name	Score
***Lex*$_{\mathbf{Ochiai}}$**	13.74	Ample2	16.48	**Keynes**	18.22	InfoGain	19.93
PattSim2	13.88	**Dennis**	16.66	**Good2**	18.22	JMeasure	19.95
Zoltar	13.92	**Popper1**	16.87	**Finch1**	18.22	Ochiai2	20.40
Naish	14.01	**Korb3**	16.93	**Forbes1**	18.22	**SokSneath5**	20.40
PattSim1	14.21	2WaySupport	16.93	Tarantula	18.22	MI	20.53
WongIII	14.23	YulesQ	17.11	Interest	18.22	**Peirce**	22.37
Kulc2	14.41	**NPV**	17.15	AddedValue	18.22	Leverage	23.20
M2	14.52	**Rescher**	17.15	SebagSch	18.22	BinaryNaish	23.34
Ochiai	15.25	**Lewis**	17.16	OddMultiplier	18.22	WongI	23.43
Conviction	15.65	AMean	17.17	Example	18.22	Confidence	23.43
Certainty	15.65	Stiles	17.27	Zhang	18.22	Fleiss	23.61
Crupi	15.88	GMean	17.27	**Korb2**	18.24	Scott	23.86
Michael	16.00	Phi	17.27	1WaySupport	18.24	Faith	24.49
Klosgen	16.31	Jaccard	17.40	Laplace	18.42	LeastCont.	25.83
Mountford	16.41	CBISqrt	17.68	**Suppes**	18.62	WongII	25.85
YoudensJ	16.48	**Popper2**	17.74	**Pearson1**	18.62	Gower	26.41
Earman	16.48	Cohen	17.98	**SokSneath4**	18.65	GoodKrus	26.64
Carnap1	16.48	Kappa	17.98	HMean	18.65	**Specificity**	27.20
Carnap2	16.48	CBIlog	18.11	**Good**	19.26	Anderberg2	27.25
Levi	16.48	**Likelyhood**	18.22	**PearlII**	19.26	**FagerMc**	29.10
Dispersion	16.48	GilbertW	18.22	**Cheng**	19.26	**Rand**	31.74

Table 4: Overall *avg* scores for measures

We now present our experimental data and quantify their significance. We first discuss Table 4. The average scores for those suspiciousness measures with a higher score than the random measure are listed in Table 4. Equivalent measures are represented by one measure per equivalence class (with preference given to measures already established in SFL), and the new measures are in bold. Note that (thirteen) additional potential sets of equivalences are suggested by the equal scores in the table.

We first make some general observations about the table. Some prominent established measures appear quite low on the list, such as Jaccard and Tarantula. It is interesting that Tarantula (which is equal to $P(E|C)$), performs worse than NPV (which is equal to $P(\neg E|\neg C)$). Also, some established measures appear quite high on the list, such as Zoltar and Naish. Thus, our larger-scale

[7] Following a convention on small sample sizes, we applied continuity correction by adjusting the Wilcoxon rank statistic by 0.5 towards the mean value when computing the z-statistic.

comparison accentuates the successes and failures of established measures. The top-performing measures from each of our newly introduced categories of similarity, confirmation, predictive, causal and custom measures are PattSim2, Crupi, NPV, Lewis and Lex_{Ochiai}, respectively. As we can see, many new measures are competitive with established ones. Finally, Rand's average score was consistently between 30–38% on reruns, which is what one might expect given an average of 2.88 bugs per program version.

We now discuss PattSim2. We saw that PattSim2 has an elegant monotonic reduction to $-a_{nf}a_{ep}$, and performs particularly well despite its relative simplicity, coming in the second place. Note that the difference between the results for PattSim2 and PattSim1 was a consequence of changing the prior constant (PC) (the details of which are discussed in the experimental setup section) in order to try and optimise the PatternSimilarity measure. We experimented in this way with this measure, because we noticed (as a theoretical observation) that by lowering the prior constant for a_{nf} it became a measure that converged to being single bug optimal. PattSim2 is a statistically significant improvement over PattSim1 using $p = 0.02$. We emphasise that we did not observe that changing the PC for our other top measures resulted in improvements in terms of their relative position in Table 4. We believe this is because the simplified ranking equivalent version of PatternSimilarity $= -a_{nf}a_{ep}$ used in our experiments is an extremely simple measure, and is consequently altered significantly by small adjustments (such as PC), where other measures are not.

We now discuss Lex_{Ochiai}. This is our new optimising scheme Lex_g with Ochiai as the internal measure g, and it is the top performer. Most of the measures can be used as a submeasure g for Lex_g and achieve a better score than all other measures below Lex_{Ochiai} in the table. To this end, Lex_{Stiles}, Lex_{M2}, and $Lex_{Wong-III}$ are the runners up. Thus, Lex_g can be viewed as a good measure optimiser on our benchmarks. Lex_{Ochiai} achieved the best score for three of the twelve benchmarks (Tcas, Totinfo, Schedule2, Replace, Gzip, Flex, Grep), the second best score for two (PrintTokens, PrintTokens2) and performed less well (towards the bottom end) in the remaining three, but never went below a score of 18.77 for such benchmarks (meaning it still has a good score in cases where other measures are ranked higher). Lex_{Ochiai} still maintains the top overall average score if the test is run on the small programs alone (i.e. the Siemens test suite, with a score of 17.83) and comes a close third with a score of 8.02 on the larger programs alone (after PattSim2 and Zoltar which score 7.53 and 7.76, respectively). If the worst score is used instead of the average, Lex_{Ochiai} still has the top overall score (22.32). Overall, these results support the claim that Lex_{Ochiai} is a robust and top-performing suspiciousness measure.

We now discuss our significance tests. Firstly, Rand was significantly better than Loevinger, TwoWaySupportVariation, CollectiveStrength, and GiniIndex, using $p = 0.05$. We believe this is sufficient to conclude that these measures are ineffective in SFL. Lex_{Ochiai} was significantly better than all measures using $p = 0.29$. Using $p = 0.05$, it was significantly better than everything that scored below and including Peirce in the fourth column of Table 4, and was additionally

significantly better than Good, PearlII, Cheng and Infogain. Thirdly, PattSim2 was significantly better than everything below and including Leverage in the fourth column (apart from Fleiss and Scott), and was additionally better than MI, Jmeasure, Infogain, Cheng, PearlII, Good, Hmean, SokSneath4, Pearson1, Suppes, Mountford and PattSim1. In general, PattSim2 was significantly better than all the measures below it on Table 4 using $p = 0.21$ (apart from Zoltar $p = 0.92$).

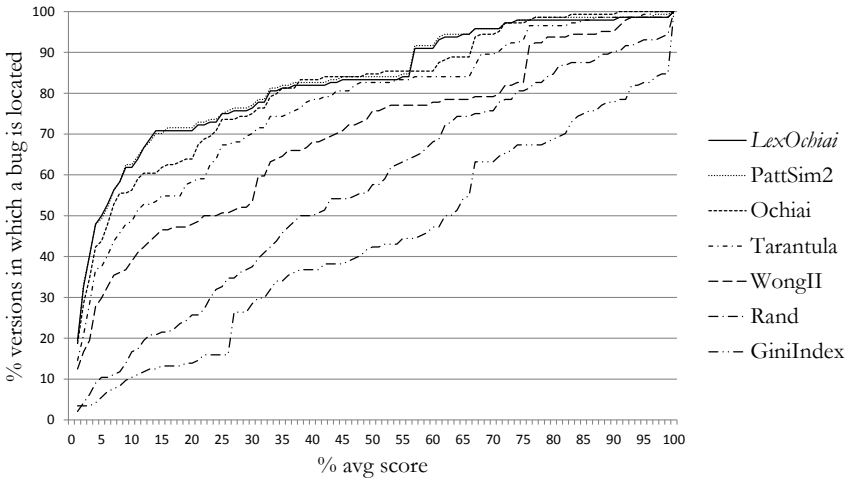

Fig. 2: Graphical comparison of prominent measures

Finally, we discuss Fig. 2, which compares the performance of some prominent measures graphically. For each measure, a line is plotted as a function of the avg scores for each of the 180 program versions. If $y\%$ of those versions have an average score $\leq x\%$, a point is plotted on the graph at (x, y). For example, for the Rand measure, 50% of the versions have an average score lower than 58%. Lex_{Ochiai} and PattSim2 have an almost aligned performance.

5 Conclusions and Future Work

We presented what is, to the best of our knowledge, the largest comparative analysis of suspiciousness measures on C programs for SFL to date, comparing 157 different measures on 12 C programs, constituting over a million lines of examined code. Out of these measures 95 are new to the domain of SFL. We taxonomised these measures into five different classes: similarity, association, causation, confirmation and custom measures. We demonstrated that each class is applicable to SFL, and that many measures are in fact equivalent in terms of ranking, thus reducing the space of measures for experimental consideration.

We defined a new custom measure optimiser Lex_g, which can admit any other measure g as its inner measure. Our experimental results demonstrate that our new measures Lex_{Ochiai} and PattSim2 achieve the best average scores over other measures and are significantly better than many of them with $p = 0.05$.

We conjecture that the top performance of Lex_{Ochiai} is owed to a strong a priori component (single bug optimality), together with an experimentally vindicated a posteriori component (using the Ochiai measure as a submeasure). Our second best performer, PattSim2, is ranking equivalent to $-a_{nf}a_{ep}$, demonstrating the success of an extremely simple measure.

We will extend our work in several directions. On the experimental side, we plan to perform experiments on benchmarks that have multiple bugs. Given publicly available multiple-bug benchmarks are rare, this includes the creation of such benchmarks. On the theoretical side, we would like to investigate conditions for multiple-bug optimality, and develop measures that satisfy those conditions. Finally, we would like to create an easy-to-use tool that implements the measures discussed in this paper.

References

1. R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 39–46. IEEE, 2006.
2. R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *TAICPART-MUTATION*, pages 89–98. IEEE, 2007.
3. L. C. Briand, Y. Labiche, and X. Liu. Using machine learning to support debugging with Tarantula. In *International Symposium on Software Reliability (ISSRE)*, pages 137–146. IEEE, 2007.
4. S. S. Choi, S. H. Cha, and C. Tappert. A Survey of Binary Similarity and Distance Measures. *Journal on Systemics, Cybernetics and Informatics*, 8:43–48, 2010.
5. Collofello and Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *Journal of Systems and Software*, pages 745–770, 1989.
6. H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Eng.*, pages 405–435, 2005.
7. E. Eells and B. Fitelson. Symmetries and asymmetries in evidential support. *Philosophical Studies*, 107:129–142, 2002.
8. B. Everitt. *The Cambridge Dictionary of Statistics*. CUP, 2002.
9. B. Fitelson and C. Hitchcock. Probabilistic measures of causal strength. In P. M. Illari, F. Russo, and J. Williamson, editors, *Causality in the Sciences*. Oxford University Press, Oxford, 2011.
10. R. Fletcher and W. Suzanne. *Clinical epidemiology: the essentials*. Lippincott Williams and Wilkins, 2005.
11. A. Groce. Error explanation with distance metrics. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 108–122. Springer, 2004.
12. F. Huber. Confirmation and induction. `http://www.iep.utm.edu/conf-ind/`.
13. J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *ASE*, pages 273–282. ACM, 2005.

14. B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. *SIGPLAN Not.*, pages 15–26, 2005.
15. C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Trans. Softw. Eng.*, 32(10):831–848, 2006.
16. C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: Statistical model-based bug localization. *SIGSOFT Softw. Eng. Notes*, pages 286–295, 2005.
17. Lucia, D. Lo, L. Jiang, and A. Budi. Comprehensive evaluation of association measures for fault localization. In *International Conference on Software Maintenance (ICSM)*, pages 1–10. IEEE, 2010.
18. Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process*, 26(2):172–219, 2014.
19. L. Naish and H. J. Lee. Duals in spectral fault localization. In *Australian Conference on Software Engineering (ASWEC)*, pages 51–59. IEEE, 2013.
20. L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, pages 1–11, 2011.
21. C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 199–209. ACM, 2011.
22. J. Pearl. Probabilities of causation: three counterfactual interpretations and their identification. *Synthese*, 1-2(121):93–149, 1999.
23. K. Pearson. On the theory of contingency and its relation to association and normal correlation, 1904.
24. B. Pytlik, M. Renieris, S. Krishnamurthi, and S. Reiss. Automated fault localization using potential invariants. *Arxiv preprint cs.SE/0310040*, 2003.
25. M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39, 2003.
26. P.-N. Tan, V. Kumar, and J. Srivastava. Selecting the right interestingness measure for association patterns. In *Knowledge Discovery and Data Mining (KDD)*, pages 32–41. ACM, 2002.
27. F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.
28. W. E. Wong and Y. Qi. Effective program debugging based on execution slices and inter-block data dependency. *Journal of Systems and Software*, 79(7):891–903, 2006.
29. W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai. Effective fault localization using code coverage. In *Computer Software and Applications Conference (COMPSAC)*, pages 449–456. IEEE, 2007.
30. 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 Trans. Softw. Eng. Methodol.*, pages 31:1–31:40, 2013.
31. S. Yoo, M. Harman, and D. Clark. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Trans. Softw. Eng. Methodol.*, 22(3):19, 2013.
32. Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang. Capturing propagation of infected program states. In *ESEC/FSE*, pages 43–52. ACM, 2009.