Journal of Empirical Software Engineering
DOI 10.1007/s10664-014-9304-1

Fault Density, Fault Types, and Spectra-based Fault Localization

Nicholas DiGiuseppe · James A. Jones

2014-03-18

Abstract This paper presents multiple empirical experiments that investigate the impact of fault quantity and fault type on statistical, coverage-based fault localization techniques and fault-localization interference. Fault-localization interference is a phenomenon revealed in earlier studies of coverage-based fault localization that causes faults to obstruct, or interfere, with other faults' ability to be localized. Previously, it had been asserted that a fault-localization technique's effectiveness was negatively correlated to the quantity of faults in the program. To investigate these beliefs, we conducted an experiment on six programs consisting of more than 72,000 multiple-fault versions. Our data suggests that the impact of multiple faults exerts a significant, but slight influence on fault-localization effectiveness. In addition, faults were categorized according to four existing fault-taxonomies and found no correlation between fault type and fault-localization interference. In general, even in the presence of many faults, at least one fault was found by fault localization with similar effectiveness. Additionally, our data exhibits that fault-localization interference is prevalent and exerts a meaningful influence that may cause a fault's localizability to vary greatly. Because almost all real-world software contains multiple faults, these results affect the practical use and understanding of statistical fault-localization techniques.

Keywords Debugging · Fault Localization · Fault Behavior

Nicholas DiGiuseppe
University of California, Irvine
E-mail: nicholas.digiuseppe@uci.edu

James A. Jones
University of California, Irvine
E-mail: jajones@uci.edu

1 Introduction

As can be observed by inspecting almost any public bug-reporting system, nearly every software project contains multiple active faults. Unfortunately debugging is often an arduous and time consuming task, requiring that before faults can be fixed, they first must be located, i.e., *fault localization*. Research suggests that of all debugging tasks, fault localization is the most time consuming and requires the most expertise [33]. To help reduce these costs, researchers developed automated fault-localization techniques. This work analyzes a subset of fault-localization techniques that utilize execution-based, statistical data (e.g., [1, 21, 23, 25]) — *coverage-based fault localization* (CFL). We target the scope of these studies to specifically CFL techniques due to the research community's recent emphasis on this area. For example, a fault-localization literature survey by Wong and Debroy discusses 19 separate articles that explicitly address specialized CFL techniques [35], and many of the earlier CFL techniques are cited several hundreds of times by other scholarly work that extends the techniques (e.g., [21, 23]). Such statistics are presented as indications of the need for such study and for the utility of such scoping. We do not attempt to generalize such findings to other, non-CFL, fault-localization techniques, such as Delta Debugging [37] — to generalize such results to other types of techniques, further study would be needed. Such CFL techniques attempt to identify correlations between software failure and program locations.

Notwithstanding years of steady improvements to CFL techniques, much of our understanding regarding their utilization remains elementary. Although CFL research has explored many possible alternatives to improve automation and detection techniques, little research has been done to improve understanding of CFL effectiveness or utility. Numerous studies compare different CFL techniques, but relatively little research has been done to investigate factors that influence CFL results, such as fault quantity in a program or fault type.

Previous research claims that CFL loses almost all effectiveness in the presence of multiple faults (discussed in more detail in Section 2). Unfortunately almost all real-world programs contain multiple faults. Yet CFL techniques can only be beneficial for real-world debugging if they are effective in the presence of multiple faults, or can be assisted with additional techniques. This work empirically investigates and evaluates how fault quantity and fault type influence CFL techniques.

Further, although there has been work done to classify different fault types, to the authors' knowledge, no studies have been done to determine the impact of these fault types upon fault localization techniques or fault interaction. This work empirically investigates and evaluates how different fault types impact a fault's ability to be localized, and whether a fault experiences more or less fault-localization interference.

Fault-localization interference (FLI) — a phenomenon identified during this experiment — occurs when a fault's ability to be localized decreases due to the presence of other faults. To better understand the FLI phenomenon, this work empirically analyzes its frequency and magnitude in real software, along

with its correlation to fault type. Our results suggest that FLI has a significant impact on CFL results and is prevalent in software containing multiple faults. Notwithstanding FLI prevalence and impact, our evaluation indicates that CFL remains effective for at least one fault, where effectiveness is measured by having a low *expense* score.

Our study demonstrates that for our subjects, the effectiveness of CFL techqniques reduced only slightly as the quantity of faults increased substantially, which largely contradicts the community's prior assumptions of substantial effectiveness losses. This work is an extension of our earlier studies [12]through an addition of more generalizable experiment designs, a deeper investigation into the earlier findings, and two additional, new studies.

The main contributions of this paper are:

1. An empirical analysis of coverage-based fault localization (CFL) techniques that: (1) challenges and in many cases refutes commonly held beliefs regarding CFL effectiveness, and (2) finds evidence of the factors that caused researchers to believe otherwise. Our results demonstrate that CFL techniques degrade roughly 2% in effectiveness at high fault quantities. Additionally, our results provide developers with accurate information about the practicality of utilizing CFL techniques.
2. An investigation of *fault-localization interference* (FLI): when one fault interferes with another fault's localizability. A presentation of empirical data to help characterize FLI in addition to measuring its prevalence and magnitude. Additionally this work presents an investigation into correlations between fault type and FLI behavior. Our results demonstrate that fault-localization interference is: (1) prevalent, (2) has a significant impact upon CFL results, and (3) rarely has the effect of impairing a CFL technique's ability to localize at least one fault.
3. Analyses explaining the impact of fault type upon fault-localization techniques and interference. All studied faults are categorized according to four previously published fault taxonomies to perform an investigation into the correlations that may exist between fault type and a fault's ability to be localized. Our results demonstrate that these fault types have no significant correlation to a fault's ability to be localized, to cause interference on other faults, or to be affected by interference from other faults.
4. An analysis of our results that has implications for the practical applicability of CFL approaches. These implications inform decisions about which automated techniques are needed in an organization, while considering their trade-offs.

In the next section we present a thorough background which motivates these studies. Then, in Section 3 we explain our experiment's design. Next, in Section 4 we analyze those results and their implications. Then, in Section 5 we identify the threats to validity of this work. Finally, in Section 6, we conclude.

2 Background and Motivation

To provide the necessary background that motivates this work we: enumerate how this work extends our previous research (Section 2.1), discuss four general fault-taxonomies (Section 2.2), overview CFL techniques (Section 2.3), summarize current perceptions of CFL techniques in the presence of multiple faults (Section 2.4), provide an example that demonstrates the assumptions for these perceptions (Section 2.5), define the interaction of multiple faults with regard to CFL (Section 2.6), and describe the need for further study on this topic (Section 2.7).

2.1 Previous Work

This paper expands earlier work presented by the authors at ISSTA 2011 [12]. Although both bodies of research address similar questions, this work is substantially different in four ways.

First, this paper presents a more generalizable and mature experiment. This work uses twice the number of subjects, and more than four times the number of faulty versions as the previous work, and additionally, expands the maximum fault quantity for all subjects by 70% (resulting in the execution of more than an order of magnitude more test cases — more than 1.6 billion). These additional subjects, versions, executions and maximum fault quantity magnify the maturity and generalizability of the earlier work and provide a more thorough understanding of the research questions addressed.

Second, this work presents an entirely new facet in this study — an investigation of fault-type correlation with CFL and FLI. Four fault taxonomies are utilized with our subjects to evaluate how fault-type influences CFL *expense*, FLI *frequency*, and FLI *magnitude* (i.e., the fault with the highest fault-correlation score, the likelihood of a fault experiencing FLI, and the degree of change in fault-correlation score for a fault experiencing FLI).

Third, another entirely new study — an investigation into FLI magnitude. Although our previous research examined the occurrence rate of FLI against fault quantity, the degree of FLI's impact on CFL was unexplored. This experiment investigates FLI magnitude by calculating the expected range of FLI against fault quantity.

Fourth, a deeper and more thorough investigation of earlier findings — an investigation of the *expense* range for the prominent fault (i.e., the fault with the highest fault-correlation score. The previous paper only discussed the mean for the prominent fault, establishing a pattern of expected behavior. This experiment expands our previous research by investigating the statistical changes that occur as fault densities change, providing a deeper understanding of fault density's impact upon CFL *expense*.

2.2 Fault Taxonomies

To classify each fault type, four fault taxonomies are used which are taken from Smith and Robsom, Firesmith, Hayes, and Hayes *et al.* [16, 17, 22, 31]. Hereafter these taxonomies are referred to as `Smith92`, `Firesmith92`, `Hayes94`, and `Hayes11` respectively. These taxonomies were selected because they are all for general faults (as opposed to specific faults like security, or network) and have a different emphases. Each of these four taxonomies is designed to represent any fault, regardless of domain, though this comes with a comparatively smaller degree of specificity. Each of these taxonomies highlights a different type of program behavior: `Smith92` emphasizes the difference between conceptual inaccuracies, and implemented inaccuracies; `Firesmith92` emphasizes consistency among visibility, components, and use of resources; `Hayes94` utilizes general categories of software quality (e.g., abstraction, encapsulation); and `Hayes11` proposes a detailed hierarchy of faults based upon location, usage and intention. For transparency, Section 7 presents a more detailed look at how these taxonomies classify faults.

2.3 Coverage-based Fault Localization

For elucidation of discussion we introduce the terms of fault execution, infection, propagation, and failure (i.e., the "PIE model") as introduced by Voas [34] and later further interpreted by Zeller [38]; the reader is referred to these works for a more formal treatment. A fault is the line(s) of code that contains statements that are inconsistent with developer intent. Infection is the initial state that is caused by executing the fault that is inconsistent with developer intent (e.g., variables contain incorrect values or an incorrect control path is executed). Propagation is the effect of subsequent incorrect states caused by the initial infection. Finally, failure is the propagation of incorrect state that becomes manifest in externally observable ways.

Jones *et al.* proposed a fault-localization technique, Tarantula [18,20,21], which utilizes whole test suites (or any subset thereof) to infer likely locations for faults based upon the relative participation of the passing and failing test cases with run-time events (i.e., observable or recordable actions during a program's execution, such as instruction execution or dynamic dataflows). Variant implementations of Tarantula have been created to target multiple run-time events, such as instruction execution (*e.g.,* [20, 21]), data-flows [30], and database interactions [4]; and variants use different underlying metrics (*e.g.,* [20, 36]). Liblit *et al.* proposed *Statistical Bug Isolation*, which monitors and utilizes randomly sampled subsets of coverage to reduce runtime overhead [23]. Additionally, Liblit *et al.* analyze the statistical-likelihood that a predicate that evaluates to "True" is correlated with failure by identifying the *context* and the general failure correlation for that predicate. By capturing both values, Liblit *et al.* calculate the *increase* of a predicate, i.e., the likelihood that a predicate evaluating to "True" causes a failure. Liu *et al.*

proposed SOBER, a technique that expands upon Liblit's work to localize the same faults while requiring less code [25]. SOBER extends the probability model introduced Liblit *et al.*, which analyzes the context of a predicate across all runs, by analyzing the context of a predicate *within* a single run. All of these techniques require some level of instrumentation on the code in order to derive runtime statistics.

The main insight shared by these CFL techniques is that, execution events that correlate with failures are more likely to be the cause (i.e., fault or bug) of those failures. Said differently, execution events that occur mostly in failing test cases, but rarely in passing test cases, are more *suspicious* of being the fault. CFL analyzes dynamic correlations between instructions and the passing or failing of test cases. This correlation approximates the likelihood that an instruction causes failure.

However, in a program containing multiple faults, the intersection of instructions executed by multiple failures that are caused by different faults might not correspond to any fault. These instructions may correspond to non-fault-relevant code, and thus, may mislead the inferencing technique and user. Due to the possibility of this misleading inferencing, the next section presents the observations from researchers in the fault localization community.

2.4 Assumptions of CFL for Multiple Faults

This sections presents the observations and accepted conclusions regarding multi-fault programs and misleading inferencing. Jones *et al.* reported that the "effectiveness of the technique declines on all faults as the number of faults increases" (they also note that these results may be misleading and require further study) [21]. Later, Jones *et al.* investigated the use of failure clustering to remove "noise" caused by one fault inhibiting the localization of another [19].

Other researchers have made similar claims. For instance, Denmat *et al.* state that the Tarantula technique (and thus other similar CFL techniques), makes implicit hypotheses requiring independence of multiple faults — every failure is caused exclusively by a single fault — and when these hypotheses do not hold, the technique does not provide "good results" [7]. Zheng *et al.* developed specialized techniques targeting programs containing multiple faults because, in the presence of multiple faults, traditional CFL techniques "cannot distinguish between useful bug predictors and predicates that are secondary manifestations of bugs," such as fault infection and propagation that does not lead to failure [39]. Referring to coverage-based fault-localization techniques, Srivastav *et al.* stated that "multiple faults in a software many times prevent debuggers from efficiently localizing a fault" [32]. Further, Debroy and Wong state that "incorrect matching of failed test to fault, ... may in turn result in poor fault localization" [6]. Thus, programs that contain multiple faults, which past studies suggest results in increased difficulty for failed-test-to-fault matching may in turn face poor localization [10, 14].

Drawing such conclusions is not unreasonable or unfounded. Indeed, studies (e.g., [18, 19, 21, 39]) have shown that for *specific* faults, the presence of other faults may impair the ability of CFL techniques to properly localize them. Said differently, these studies conclude that CFL performed poorly if, in a multi-fault program, it was unable to localize some *specific* fault (i.e., an a-priori, preselected fault from those existing within the application). This presumed poor localization has led to one motivation for failure clustering — localizing multiple specific faults (e.g., [19, 24, 28, 39]).

To help localize a specifically chosen fault, one approach has been to utilize a technique called failure clustering (e.g., [8, 9, 13, 19, 24, 28, 39]). Failure clustering attempts to group together test cases that fail due to the same fault(s). Indeed, failure clustering can be an effective precursor to CFL techniques, as evidenced by earlier studies (e.g., [19, 39]). Such clustering techniques can minimize, although not eliminate, some fault-localization interference. In theory, the effectiveness of the subsequent CFL technique may benefit from less "noise" in localizing the most prevalent fault for each cluster.

Unfortunately, failure clustering adds a level of computational cost and an additional imposition of tool support and development-practice changes. In many circumstances developers are not looking for a specific fault, but instead would be satisfied to fix any fault. Consider the following scenario. During a verification stage prior to software release, developers seek to ensure the software meets minimum requirements by enabling it to successfully run a test suite. In these circumstances, developers seek to fix all faults causing incorrect functionality or failure. Further, in the agile development model, after adding any functionality, developers cannot "move on" without ensuring that all test cases pass.

However, regardless of the applicability of failure clustering to assist with CFL effectiveness in the presence of multiple faults, we seek to better understand how such effectiveness is influenced by such multiple faults. Our experiments in this paper seek to provide such findings, and may be useful in helping to determine the issues and trade-offs involved in deciding whether to employ other techniques, such as failure clustering.

The following section provides an example that demonstrates the concern of these previous researchers — how multi-fault programs could mislead localization techniques to reduce their effectiveness.

2.5 Example

Consider, the code (displayed in the first column) presented in Figure 1. The program snippet listed in the first column contains two faults, labeled "`bug1;`" and "`bug2;`". In Figure 1, the four columns to the right of the code list the test cases: t1 and t2 are passing test cases, and t3 and t4 are failing test cases. Test cases t3 and t4 fail due to different faults, `bug1` and `bug2`, respectively. The next three columns lists the *suspiciousness* scores, differing only by the

	t1	t2	t3	t4	*suspiciousness (t1,t2,t3,t4)*	*suspiciousness (t1,t2,t4)*	*suspiciousness (t1,t2,t3)*
`if ( b ) {`	•	•	•	•	70	60	60
`bug 1;`	•		•		70	0	70
`} else {`		•		•	70	70	0
`bug 2;`		•		•	70	70	0
`}`	•	•	•	•	70	60	60
pass/fail?	P	P	F	F			

Fig. 1: Example code snippet containing two faults. This example demonstrates the possibility of multiple faults creating noise that interferes with fault localization effectiveness.

subset of the test suite used to generate them; the test cases used are listed in the first row.

The *suspiciousness* score is calculated with the Ochiai metric, which originated in the molecular biology domain and was proposed by Abreu *et al.* to enhance the Tarantula technique [1]. The equation for Ochiai is

$$\text{suspiciousness}(i) = 100\left(\frac{\text{failed}(i)}{\sqrt{\text{totalfailed}(\text{failed}(i)+\text{passed}(i))}}\right) \quad (1)$$

where *passed*(i) is the number of passed test cases in which instruction i is executed, *failed*(i) is the number of failed test cases in which instruction i is executed, *totalfailed* is the number of failed test cases in the test suite, and *suspiciousness(i)* is the approximation that instruction i is the fault, ranging from 0 to 1, where 1 is the most suspected instruction, and 0 is the least.

Thus, a developer is expected to first inspect the instruction with the highest *suspiciousness* score, and, upon not finding the fault, inspect the instruction with the next highest score, and so forth until the fault is found. By following this ranked list of instructions, CFL techniques attempt to reduce the number of instructions, (i.e., the search space), a developer must examine to find the fault.

The first column in Figure 1 shows the *suspiciousness* scores when considering all test cases in the test suite. When utilizing all test cases, the *suspiciousness* of all five instructions is 70%. Because all instructions are equally suspicious, the technique is ineffective at localizing either fault — in other words, it did not reduce the search space to find either fault.

However, if test case t3 is excluded from the test suite and suspiciousness calculation (represented by the middle *suspiciousness* column), the technique successfully localizes `bug2`. In this case, `bug2` contains the highest suspiciousness score, and is thus, successfully localized. Likewise, if test case t4 is excluded from the test suite and suspiciousness calculation (represented by the right *suspiciousness* column), the technique successfully localizes `bug1`. In this case, `bug1` contains the highest *suspiciousness* score, and is thus, successfully localized. This example demonstrates the potential ineffectiveness of CFL because of multiple faults leading to poor inferencing.

This example was constructed to depict the scenarios envisioned that caused the assumptions of CFL ineffectiveness in the presence of multiple faults.

2.6 Fault-localization Interference

The example in Figure 1 demonstrates how the presence of one fault can interfere with the localization of another. Fault-localization interference (FLI) is denoted as *any decrease* in fault-localization effectiveness (due to the presence of another fault). More formally:

> **Definition 1: *Fault-Localization Interference (FLI).*** *For a program P, the rank of fault f_x is defined as the position in a sorted list of all faults in the program when sorted according to a fault-localization metric, such as* suspiciousness*, and is denoted as $r_P(f_x)$. In such a ranking, the rank of 1 is the most localizable fault, and a rank of n is the least localizable fault in a program containing n faults. Fault-localization interference (FLI) is the phenomenon in which the rank of any fault f_i is increased due to the introduction of another fault f_j. Let program P_F contain the set of faults F, where $F = \{f_1, \ldots, f_n\}$, and where fault $f_i \in F$ and fault $f_j \notin F$. Similarly, let the set of faults $F' = F \cup \{f_j\}$. Fault-localization interference occurs if for any fault $f_i \in F$ the introduction of fault f_j causes $r_{P_F}(f_i) < r_{P_{F'}}(f_i)$.*

Note that this increase of a fault's position in the ranking can be due to a decrease in the *suspiciousness* score of the faulty instruction, or the increase of the *suspiciousness* scores of other instructions rendering them more suspicious than the faulty instruction. Additionally, we say that a fault experiencing FLI becomes *obfuscated*.

In Figure 1, if either `bug1` or `bug2` were effectively removed, which is simulated by not executing the test case which covers it, the remaining fault could be localized more effectively. An example of this can be seen by using only a subset of the test suite (i.e., the two far right columns) such that the subset only covers a single fault. If t3 were removed (which covers bug 1), the ability to localize bug 2 increases, and vice-versa if t4 is removed. Although testing does not involve the removal of test cases, this example is designed to demonstrate that test cases covering multiple faults introduce the potential for degradation in CFL results. In this example, each fault caused fault-localization interfer-

ence for the other, however it is unclear whether this type of interference (i.e., all faults become obfuscated) is common in actual software.

2.7 Motivation for Further Study

Although the previous example exhibited that in the presence of multiple faults none are localizable, the results of a case study presented by Jones *et al.* show that often, interference causes some faults to be obfuscated — that is, made less localizable with the CFL technique — while others (usually the faults causing the interference) remain highly localizable [21]. One goal of this study is to determine the prevalence and nature of fault-localization interference. In other words, *how often does fault-localization interference occur, and when it does, how often does it take the form that causes ineffective localization for all faults?*

This question of FLI type is vital because, if a CFL technique effectively localizes at least one fault even in the presence FLI, then it could be useful. Past studies examined the localizability of *all* faults and used poor localization of *any one* individual fault as evidence that fault localization decreases in effectiveness for multi-fault programs.

Existing studies examine the localizability of *specific*, individual faults; although in practice, developers many times aren't aware of the quantity or identity of the specific faults causing failures. In such situations, the localizability of *any* fault can enable a debugging process that can lead to a fault-free program. Indeed, because past work only analyzes CFL in the context of *specific* faults, little is known about whether CFL can be effective in practice. Due to the popularity of CFL research, it is essential to know whether these techniques can be implemented in practice, and the limitations that exist upon their practicality.

3 Experiment

To understand the impact of fault quantity and fault type on CFL techniques and FLI, we conducted multiple empirical evaluations. Our experiment design was chosen to answer seven research questions (see Table 1) that highlight the impact of fault quantity and fault type on CFL and FLI. The seven research questions elucidate CFL's potential for use in practice. Each question examines a different aspect of CFL practicality as follows: RQ1–2 evaluate how CFL effectiveness is altered by multiple faults, RQ3 analyzes correlations between CFL effectiveness and fault type, RQ4 produces an in-depth awareness of some of the functional challenges of CFL as faults are added or removed, RQ5–6 investigate the frequency and impact of FLI occurrence, and RQ7 analyzes the correlation between FLI and fault type. Considering these questions as a whole, they approximate when, how, and with what limitations CFL can be practically implemented.

Table 1: Research questions addressed in this experiment.

RQ1:	What is the impact of fault quantity on CFL *expense* scores?
RQ2:	What is the impact of fault quantity on CFL *suspiciousness* scores?
RQ3:	Is fault type a predictor of CFL effectiveness?
RQ4:	What are the practical ramifications of FLI on CFL?
RQ5:	How often does fault-localization interference occur?
RQ6:	What is the typical magnitude of fault-localization interference?
RQ7:	Does fault type affect the likelihood of causing interference or being subject to interference?

3.1 Variables, Measures and Definitions

This work's primary objective is to investigate the impact fault quantity and fault type have on CFL techniques and fault-localization interference. This experiment manipulates two independent variables: the quantity of faults in a program and the fault type. We alter the quantity of faults to range between one and seventeen, and classify existing faults once for each taxonomy: `Smith92`, `Firesmith92`, `Hayes94`, and `Hayes11` (see Section 3.2 for more details on the faults within our subjects). This maximum fault quantity was chosen because combinations with more faults drastically reduce the number of combinations that are possible among the total 20 faults, which would reduce the ability to generalize experimental findings.

Because our goal is to investigate a fault's *ability to be localized*, this experiment utilizes three dependent variables: (1) *suspiciousness*, (2) *expense*, and (3) *interference*. *Suspiciousness* captures the failure-correlation value assigned to instructions by the Ochiai metric. *Expense* captures the percentage of the program a developer must examine (measured in normalized lines of code) to find the fault if examining the program in decreasing order of *suspiciousness*, or more simply, the failure-correlation value assigned to an instruction in relation to all other instructions. *Interference* captures the increase in *expense* due to the presence of another fault.

Our experiment utilizes the *expense* metric originally presented by Renieris and Reiss [29] and used by many other researchers (e.g., [5,20,25]). This metric represents *expense* as a percentage of lines that a developer needs to examine before finding the fault as defined by Equation 2 where $r(f)$ is the rank of the faulty statement f in a sorted list L of all executed statements S, where L is sorted according to a suspiciousness metric in decreasing order.

$$\textit{expense(f)} = 100 \cdot \frac{r(f)}{|S|} \tag{2}$$

When a fault is composed of multiple instructions, the *expense* is calculated as the first faulty instruction found when sorted by decreasing suspiciousness.

We define S as the set of *all* executed statements in the program to allow us to study the reduction in the search space in the program's statements

brought by such automation. Although other definitions of the denominator (such as the union of the set of statements executed by all failing test cases) could be used, for the purposes of studying the effects of fault interaction and interference, such alterations from the traditional metric target separate issues from those studied in this paper. Moreover, the defined expense metric is used for every fault in every faulty version, and is more importantly used for comparison between versions of the program.

To enable the presentation of our experimental protocols and their motivations, we provide the following definitions. Because the goal of this work is to study the effects imposed by the presence and quantity of faults in a program on CFL effectiveness, we define *fault density* in terms of the number of faults in the program and the size of the program.

Definition 2: *Fault Density.* *The ratio of faults in a program to the size of a program (expressed as the number of statements, i.e., the lines of code, in the program). More formally, it can be represented by* $\frac{|F|}{|S|}$ *where* F *is the set of faults in the program and* S *is the set of statements in the program.*

Note that *fault density* is directly related to the *quantity of faults*, so each can be thusly considered in the context of the other in the ensuing discussion. Hence, as a program contains more faults, its *fault density* increases, in other words, an execution is more likely to encounter at least one fault (because more lines of the program are faulty). In addition, when referring to these faults, we define two more terms, *prominent fault* and *non-prominent faults*.

Definition 3: *Prominent Fault.* *Program* P*, composed of the set of instructions* I*, contains the set of faults* $F = \{f_1, \ldots, f_n\}$*. Each fault* f_x *is composed of discrete sets of instructions* $I_x \subset I$*, where for any* x *and* y *such that* $x \neq y$*,* $I_x \cap I_y = \emptyset$*. Suspiciousness metric* $S(i)$ *gives a heuristic measurement of fault-proneness for instruction* i*. Fault* $f_{\text{pro}} \in F$ *is composed of instructions* $I_{\text{pro}} \subset I$*, and all other faults* $F_{\text{non}} = F - \{f_{\text{pro}}\}$ *are each composed of instructions* I_{non_f}*. Fault* f_{pro} *is called the "prominent fault" if* $S(i) > S(j) \mid \exists i \in I_{\text{pro}}, \forall j \in \bigcup_{f \in F_{\text{non}}} I_{\text{non}_f}$*.*

Consequently, all other faults than the prominent fault may be considered "non-prominent faults."

Definition 4: *Non-Prominent Fault.* *Each fault* $f_j \in F_{\text{non}}$*, defined in Definition 3 is called a "non-prominent fault."*

In order to determine the extent to which faulty instructions can be distinguished from non-faulty instructions with the fault-localization technique, we define "latent suspiciousness" as the mean suspiciousness of the instructions in the program that are not faulty.

Definition 5: ***Latent Suspiciousness.*** *For program P, composed of the set of instructions $I = I_{\text{faulty}} \cup I_{\text{nonfaulty}}$, and suspiciousness metric $S(i)$, the "latent suspiciousness" is defined as the arithmetic mean of $S(i \in I_{\text{nonfaulty}})$.*

In order to determine the prevalence of fault-localization interference, we define "fault-localization interference frequency" as the ratio (or percentage) of occurrences in which the introduction of a fault caused a substantial drop in the localizability of at least one of the existing faults.

Definition 6: ***Fault-Localization Interference Frequency.*** *For a program P that contains the set of faults $F = \{f_1 \dots f_n\}$, we compute the expense $e_P(f_i)$ for each such fault in F. A new version of the program P' is created by introducing a new fault f_{new} into P. With P' we again compute the expense $e_{P'}(f_i) \mid f_i \in F$. An interference is noted when a fault exhibits an increase of expense: $\exists f_i \in F \mid e_{P'}(f_i)) > e_P(f_i)$. However, we denote a "substantial" interference when at least one fault exhibits an increase of expense by at least an order of magnitude: $\exists f_i \in F \mid e_{P'}(f_i)) > 10e_P(f_i)$. The fault-localization interference frequency is the ratio of the number of new-fault-introduction versions that caused substantial interference to any fault existing prior to the new fault introduction, to the total number of new-fault-introduction versions.*

In order to determine the extent to which interference is imposed, we define the concept of *fault-localization magnitude*. For each fault that experiences FLI, the difference in *expense* between the single fault version and the multi-fault version at varying levels of fault density is measured. In other words, to calculate the *expense increase*, we leverage the version containing *only* the fault in question, and another containing multiple faults (including the fault in question) and compare the increase in *expense* for each individual fault against all multiple fault version. More formally, we define Fault-Localization Magnitude as follows.

Definition 7: ***Fault-Localization Magnitude.*** *For a program P that contains the set of faults $F = \{f_1 \dots f_n\}$, we compute the expense $e_P(f_i)$ for each such fault in F. A new version of the program P' is created by introducing a new fault f_{new} into P. With P' we again compute the expense $e_{P'}(f_i) \mid f_i \in F$. An interference is observed (utilizing either the strict definition for interference or the definition for substantial interference) such that: $\exists f_i \in F \mid e_{P'}(f_i) = e_P(f_i) + m \mid m > 0$. Here, m is the magnitude for said fault localization interference.*

3.2 Objects for Analysis

To determine the effectiveness of CFL techniques in the presence of multiple faults, this study leverages six C-language programs that are popular in

CFL research: Flex (version 2.5.4) Gzip (version 1.0.7), Replace, Schedule, Sed(version 3.02) and Space ([1,5,11,12,19–21,25,29]). These programs vary in size as follows: Flex has 14273 lines of code (LoC), Gzip has 7928 LoC, Replace has 563 LoC, Schedule has 509 LoC, Sed has 10154 LoC, and Space has 6445 LoC. Each was obtained from the "Subject-artifact Infrastructure Repository" (SIR) along with its faults, and test cases [15].

Flex is a GNU utility that generates a lexical analyzer. Gzip is another GNU utility that performs compression and decompression. Replace and Schedule are both toy-sized programs from the so-called "Siemens suite" provided by the SIR; Replace performs textual matching and replacing and schedule prioritizes events based upon their score. Sed is a Unix utility stream editor. Space is an interpreter for an array definition language created by the European Space Agency.

As in prior work, faulty versions that exhibit no test-case failures are excluded from our experiment (e.g., [3, 11, 12, 20, 21, 25, 39]). When these omissions caused our program to have less than 20 faulty versions, or in cases where the number of faults provided by SIR was less than 20, additional faults were added through random mutation as defined by Offutt *et al.* [26]. Offutt's approach attempts to create a representative set of mutants using randomized line-selection, randomized mutant-operator-selection, and a pre-defined set of operators described for mutant insertion [26]. Additionally, the the reference version was altered in such a way that each fault can be activated or deactivated at compile-time, to enable multiple faults to be simultaneously present. Although mutants are faults that did not arise due to original developer mistakes, recent work by Ali *et al.* found that, "single-mutant-line mutants...behaved very similarly to the real faults...with respect to Tarantula," and that their results, "showed no reason to believe that mutants are unsuitable as candidates for faulty versions for the purpose of studying FL [fault localization] algorithms," [2].

3.3 Experimental Setup

This experiment captures the coverage of each test case with the instrumenter included in the Gnu C compiler (`gcc`) and its corresponding `gcov` utility. For each test case, the executed instructions and the pass/fail status are used as input for our CFL technique — the version of the Tarantula fault-localization tool, with the Ochiai suspiciousness metric (see Reference [20] for more specifics on this version of Tarantula). The Ochiai metric is used because a study by Abreu *et al.* found it to be the most effective metric for CFL inferencing [1].

The output from each faulty version is then compared with output from a fault-free, oracle version provided by SIR. When the output of the faulty version and the oracle version differ, the test case is marked as a failure; otherwise, it is marked as a pass.

Table 2: Number of versions, size of test suite, and number of executions for each of our six subject programs.

	Flex	Gzip	Replace	Schedule	Sed	Space	**Total**
# of 1-Fault Versions	21	20	25	20	20	33	139
# of 2-Fault Versions	189	182	185	165	188	175	1184
# of 3-Fault Versions	502	505	598	500	660	550	3315
# of 4-Fault Versions	1000	1000	700	500	950	670	4820
# of 5-Fault Versions	1000	1000	700	500	950	670	4820
# of 6-Fault Versions	1000	1000	700	500	950	670	4820
# of 7-Fault Versions	1000	1000	700	500	950	670	4820
# of 8-Fault Versions	1000	1000	700	500	950	670	4820
# of 9-Fault Versions	1000	1000	700	500	950	670	4820
# of 10-Fault Versions	1000	1000	700	500	950	670	4820
# of 11-Fault Versions	1000	1000	700	500	950	670	4820
# of 12-Fault Versions	1000	1000	700	500	950	670	4820
# of 13-Fault Versions	1000	1000	700	500	950	670	4820
# of 14-Fault Versions	1000	1000	700	500	950	670	4820
# of 15-Fault Versions	1000	1000	700	500	950	670	4820
# of 16-Fault Versions	1000	1000	700	500	950	670	4820
# of 17-Fault Versions	1000	1000	700	500	950	670	4820
Total # Versions	14,714	14,707	10,608	7,684	14,168	10,138	72,017
Size of Test Suite	527	214	5,542	2,560	363	13,527	22,733
Number of Executions (in Millions)	7.7	3.1	58.7	19.6	5.1	137.1	1,637.1

To create faulty versions, the following process is implemented: randomly choose a single fault, then iteratively add another randomly-chosen fault without replacement until seventeen faults are reached (saving each version whenever another fault is introduced). For example, first evaluate a version containing only fault 7, followed by a version containing faults 7 and 14, followed by a version containing faults 7, 10, and 14, ..., until seventeen faults are reached. For each version, the test suite is executed, and the CFL technique is utilized. By following such additive sequences of faults, *expense*, *suspiciousness*, and *interference* can be captured, evaluated, and analyzed as the *fault density* is increased.

To determine a stopping point, each program ran for a fixed amount of time. Thus, larger, more complex programs, or programs with larger test suites have fewer faulty versions. The exact number of faulty versions generated for each subject, along with the corresponding quantity of faults is shown in Table 2. For example, Gzip contains 20 single-fault versions, and by generating unique combinations of two faults, 180 two-fault versions were produced (e.g., a two-fault version may contain faults 4 and 17). Similarly, n-fault versions were generated by randomly choosing unique combinations of the n individual faults.

In all, these experiments generated 72,017 faulty versions from all six programs and executed the test suite for each faulty version, resulting in the execution of over 1.6 billion test cases.

3.4 Formal Analysis Methodology

To elucidate understanding regarding the method of analysis used for our results, we briefly discuss the statistical methodology used to evaluate the results in this study, the justification for these methods, and the significance threshold at which we reject the null hypothesis.

For all our tests we choose an alpha of 0.05 to compare against our p-values. This value is chosen because of its common acceptance as a standard measure to calculate significance within the software engineering field.

Regarding our statistical tests, first, to determine whether our data is normal, we implement the Shapiro-Wilk test. For this test, if the p-value is less than a chosen alpha, then the null hypothesis is rejected, meaning that the data is not from a normally distributed population.

To test the whether the difference within the results from our experiments are significant, we utilize the Mann-Whitney U test. This test is used to test two populations to determine whether they are likly drawn from the same population. The motivation to use this test comes in part from the fact that, as will be discussed in Section 4, our results are not normal. The Mann-Whitney U test is specifically designed for non-parametric (or non-normal) data. The necessary assumptions for this test are that the observations are independent, the responses are ordinal, and the distributions of both groups are equal under the null hypothesis. Our data satisfies these criteria, and thus when a p-value is less than our alpha (0.05), we conclude that the populations are statistically distinct from one another.

Lastly, to formally test correlation, we perform the Kendall Tau test. This test measures the association between two measured quantities, and produces a score between negative one and one. A score close to one indicates a strong agreement between the values (i.e., positive correlation), a score close to negative one indicates a strong disagreement between the values (i.e., a negative correlation), and a score near zero indicates no relationship between the data (i.e., no correlation). This test is a non-parametric test, meaning it does not make assumptions about the data being normal, and under the null hypothesis wherein the two data samples are independent, the result should be zero.

4 Experimental Results

Each research question is answered and exhibited within its own section. As such, the following seven subsections each describe one of our seven research questions. It is of note that these results are all described using the median scores due to the non-normality of the distribution of data. A Shapiro-Wilk test rejected the null-hypothesis (i.e., confirming the data was not drawn from a normal distribution) for each of our subjects for all our dependent variables.

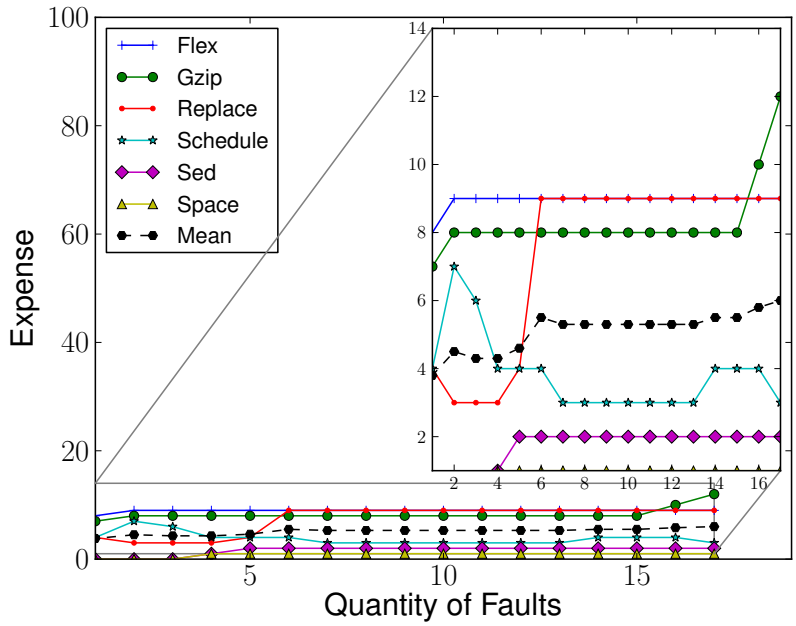

Fig. 2: The aggregate, least *expense* fault for all six subjects as the quantity of faults increases from one to seventeen. The inset plot presents a magnified view.

4.1 Research Question 1: What is the impact of fault quantity on CFL *expense*?

This experiment investigates the impact of *fault density* on CFL *expense* by measuring three data-types against fault quantity: (1) the prominent fault's *expense*, (2) the variation in the the prominent fault's *expense*, and (3) all faults' *expense*. In this way the results approximate CFL effectiveness by performing an analysis on the prominent fault's *expense*, the effect of the number of faults on all fault's localization, and how many faults can be identified simultaneously (which can improve efficiency through an increase in the parallelization of debugging).

4.1.1 Prominent Fault's Expense

Figure 2 exhibits the median *expense* results for the prominent fault in each subject at each fault quantity. The horizontal axis represents the program's fault quantity, and the vertical axis represents the median *expense* to find the prominent fault. Each separate line is a median *expense* (across all n-fault

Table 3: P-values testing the statistical significance of *expence* results drawn from the single-fault version and an n-fault version for the prominent fault.

Starting Quantity	Ending Quantity	P-Value
1	2	0.001
1	3	p<0.0001
1	4	p<0.0001
1	5	p<0.0001
1	6	p<0.0001
1	7	p<0.0001
1	...	...

versions) for a particular program's prominent fault (at a given fault quantity), whereas the median (across all n-fault versions) of the mean *expense* (across all subject programs) is presented as the dotted line. Note that, although the results are discrete for quantity of faults, the figure connects associated points with lines for comprehensibility of the trends imposed upon *expense* by the fault quantity. The plot displays a scale of 0% to 100% — to represent the full possible range — and a magnified scale in the inset plot to more precisely demonstrate our program's behavior. Although a random guess would on average result in an *expense* of 50%, this study reports the full *expense* range returned by the CFL technique — we do this because the goal of this study is to study the effect of fault quantity on effectiveness and that effect may cause worse-than-50% effectiveness.

For example, with the program Replace, the median *expense* for the prominent fault in versions containing a single fault is 4%, and the median for two-fault versions is 3%. Somewhat surprisingly, with one of the six subjects (Schedule), the prominent fault is equal or easier to find (i.e., has a lower rank) half the time when the program contains multiple faults. For the remaining five programs (Flex, Gzip, Replace, Sed, and Space) the prominent fault always has a higher *expense* when the program contains multiple faults.

The results show that mean *expense* across all subjects has an overall increase of 2% between one and seventeen faults. A two-tailed t-test for increasing quantities of faults is displayed if Table 3.

Table 3 demonstrates that when considering our subjects as a whole, that any different quantity of faults is statistically likely to perform differently than the same program containing only a single fault, as all p-values are less than 0.05. Hence, when comparing the samples from a single-fault version to a multi-fault version with any quantity greater than one, we must reject the null hypothesis that they can be derived from the same sample. Figure 2 highlights that when considering our subjects as a whole, the difference in median increases from roughly 4% to 6%, and the statistical tests show that that increase is significant.

However, existing assumptions (see Section 2.4) were that CFL techniques "prevented" localization [32] or "cannot distinguish" faults [39] in the presence of multiple faults. Despite modest, but significant, increases in effectiveness as

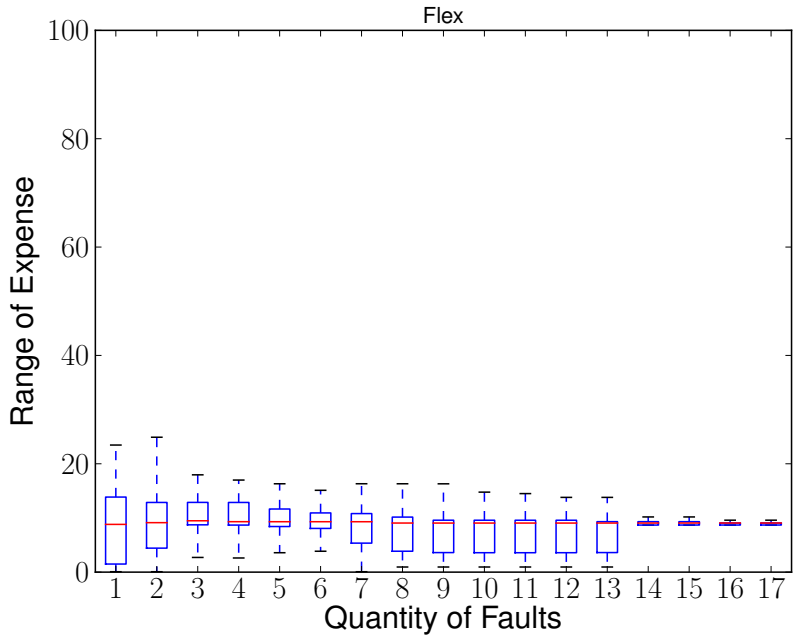

Fig. 3: Range of *expense* for the prominent fault at each fault quantity for Flex, the other subjects results are presented in the Appendix (see Section 7).

the number of faults increases beyond four faults, the prominent fault was still found in less than 6% of the program, which does not fully confer with prevailing community assumptions. This data contradicts the intuition-based assumption that CFL results are unusable in the presence of multiple faults and instead suggests that a high-fault density has only modest effect upon CFL *expense* to localize at least one fault (i.e., the prominent fault).

4.1.2 Range of Expense for the Prominent Fault

To gain a deeper understanding of the prominent fault, Figure 3 presents a box plot exhibiting *expense* values for the prominent fault at each fault quantity. The horizontal axis is the quantity of faults, and the vertical axis is the range of *expense*. Each box shows the median (the center line, colored red) and the first and third quartile (the box), and the min and max values (the dotted lines) of the *expense* observed from each version's prominent fault.

Due to the large quantity of figures for this data type, only Flex's *expense* data is shown in this figure, which is a sample from our six subjects and enables a more thorough analysis of the pattern displayed by the majority of subjects. It should be noted however that a statistical test revealed that each subject's results at each quantity of faults was different (i.e., that each subject's range

of *expense* for the prominent fault is statistically independent from each of our other subjects with p-values less than 0.05). For completeness, the remaining five *expense* boxplots (corresponding to the remaining five subjects) can be found in the Appendix (see Section 7) as Figure 11.

Figure 3 demonstrates a general negative correlation between the variance of *expense* and the quantity of faults; more simply, the variance of *expense* shrinks as the quantity of faults increases. For example, the *expense* range for the prominent fault when Flex contains a single fault is roughly 1–22%, however when Flex contains seventeen faults, it is roughly 12–12.5% (a Kendall Tau test produces a score of -0.97).

This same pattern is demonstrated by the subjects Replace, Sed, and Schedule. This figure demonstrates two significant trends: first, the *expense* variance decreases at higher fault densities, and second, the *expense* range at the highest quantity of faults (14–17 for Flex, 11–16 for Replace, 9–17 for Schedule, and 11–15 for Sed) is statistically shrinking (with a Kendall Tau test returning a scores below -0.8 for each subject).

Our data also suggests that the relatively constant *expense* median is also due to FLI. Indeed, when more faults are executed, there is an increased probability that at least one fault causes FLI, and faults that cause FLI tend to be localized first [12]. This rationale — FLI causing faults are found first — explains the fairly consistent *expense* median: these faults are never obfuscated, so regardless of fault quantity they are localized with similar *expense* values. The lack of influence on *expense* range at higher fault quantities suggests that CFL techniques eventually reach a *saturation* level such that their *expense* for locating a fault neither can increase nor decrease until some faults are removed. This suggests that although CFL results initially decrease in effectiveness with more faults (as demonstrated previously), this decrease is halted at higher fault quantities. However, the exact specification of "higher" seems to be somewhat program-specific, as Flex reaches this state at 14 faults, Replace, and Sed reach this point at 11 faults, Schedule reaches this point at 9 faults, and neither Gzip or Space reached this point by 17 faults.

This pattern of *expense*-range shrinking, is in contrast to that which is exhibited by the subjects Gzip and Space. When considering the variance of the subject Space at various fault quantities, we find statistically different values between each fault quantity between one and nine faults (a p-value less than 0.008), and between each fault quantity thereafter we cannot reject the null hypothesis (p-values greater than 0.05), meaning that current data cannot validate that these samples are from different populations Further, we see that at between one through nine faults, Space's *expense* range increases (with a higher median, first quartile, and max). Similarly, for Gzip's variance between fault densities one through six we cannot reject the null hypothesis (p-value greater than 0.05) and thus cannot validate that these samples come from different populations. However, Gzip demonstrates a statistically different value at higher fault densities at faults 11–17, (with a p-value less than 0.05). Further, when conducting a Kendall Tau correlation test, we find that between a single-fault version, and a sixteen-fault version, Gzip has a score of 0.66 and

Space has a score of 0.99 (indicating an increasing range as the quantity of faults increases). Thus, these programs exhibit opposite behavior: Space has a constant range at high quantities of faults (quantity greater than nine), and Gzip has a constant range at low fault quantities (quantities less than eight).

We speculate that Space's behavior is actually similar to that of our other four subjects. Although its range is much larger, the range stabilizes similar to that of Flex, Replace, Sed, and Schedule. This is likely due to a FLI (as mentioned above). However, it suggests that the potential range of *expense* is somewhat program specific (e.g., Schedule has a larger range than Replace).

As for Gzip, an anecdotal investigation into this behavior suggested that it may be different due to the nature of its functionality. Gzip only performs two main functions, compressing or decompressing. If either of these functionalities contain a fault, the process almost always fails; however, having multiple faults within the same functionality does not cause new failures (as all test cases that executed that process failed at smaller fault quantities). This means that CFL has a difficult time in distinguishing non-faulty, but always executed code, with faulty, but always executed code. This would explain the relative stability.

4.1.3 All Faults' Expense

Figure 4 presents the *expense* of all faults, including those that do not have the lowest *expense* score (i.e., all faults which are not localized first) in Figure 4. This figure presents all the faults' *expense* values for every fault in Sed; note that the remaining five figures for the other subjects can be found in the Appendix in Section 7 labeled as Figure 12.

In Figure 4, the horizontal axis represents the fault quantity within the program, and the vertical axis represents *expense*. Plot points correspond with the median for all n-fault versions — (e.g., at the "1" position on the horizontal axis, the plot point represents the median *expense* for *all* 1-fault versions). Furthermore, at the "1" position on the horizontal axis, only a single point is drawn because (by definition), there can be only one fault to localize. For example, the two-fault plot line has no value at the "1" position on the horizontal axis — each line makes its introduction at one point further along the horizontal axis. All plot points correspond with multiple different faults — points are classified by their position in the sorted *expense* measure, not by a fault identifier. Finally, in interpreting this graph, when using a find-fix-rerun process only the lowest *plot line* is of interest (i.e., this line should be followed to the *left*) and when using a find-fix-find-fix-repeat (without rerunning CFL) process jump *up* between lines.

All our subjects exhibit that the prominent fault has a statistically lower *expense* than even the second-most prominent fault (a p-value less than 0.001 for all subjects at all quantities of faults). A deeper investigation into the data used to construct this graph revealed that the prominent fault was often different at each quantity of faults. In other words, the faults creating the lowest *expense* were often different at each fault quantity. This difference of prominent fault at each quantity suggests the importance of understanding

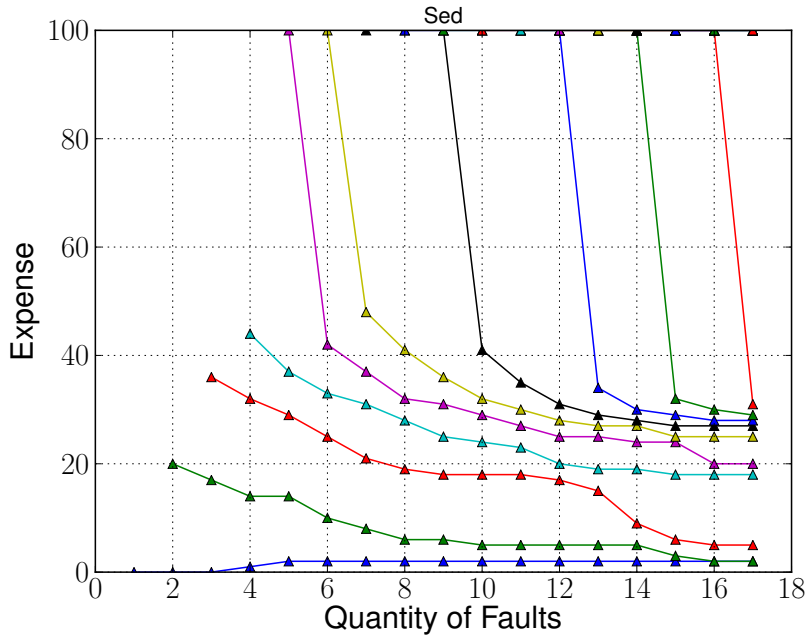

Fig. 4: Aggregated *expense* for each fault in Sed. The lowest plot line represents the fault with the least *expense*, and thus is localized first. Each of the other, higher lines represent next found faults. The other subject's results appear in the Appendix (see Section 7).

fault interaction, or how multiple faults interact to change the behavior of other faults (see [10] for a more thorough treatment). This is a somewhat surprising result as the *expense* at each quantity of faults is somewhat stable for the prominent fault. Indeed, this means that even though the fault(s) responsible for the failure are unique at each quantity of faults, they manage to score similarly in the context of being localized.

When examining the non-prominent faults (those which would not be localized first), in most cases, they contain values that suggest localization will be difficult. For example, at seventeen faults, the fourth-easiest-to-find fault has an *expense* that is more than five times that of the first-localized fault. Thus, it would likely be faster to fix a single fault, rerun the CFL technique, get new results, and find the next fault, than it would be to try and fix the second or third prominent fault. Another feature of interest is that the non-prominent faults change dramatically in *expense* at different quantities of faults. This is surprising given that the prominent fault is somewhat stable. An anecdotal investigation into this behavior suggests that fault localization is only able to isolate a single infection behavior at a time. In other words, the fault(s) causing failure mask the infections from the remaining faults such that the spectra

of non-prominent faults are obfuscated. We speculate that utilizing different spectra may be able to identify different fault(s) simultaneously or at least lower the *expense* for some of the non prominent faults.

The data points from these three sub-questions provide a few key insights: (1) the fault that is localizable may be unique to the combination of faults, (2) it is unlikely that more than a single fault can be localized at once, and (3) CFL best supports an iterative debugging approach targeting the prominent fault.

For RQ1: When considering the prominent fault, fault density has significant but modest impact of increasing CFL *expense* (∼2%), and for the majority of our subjects will narrow the *expense* variance at higher densities. When considering the non-prominent faults, fault density has a significant impact that makes most faults unlocalizable due to an increased *expense*.

4.2 Research Question 2: What is the impact of fault quantity on CFL *suspiciousness* scores?

RQ2 is motived by the important relationship between the *suspiciousness* of the the mean line of code, and the *suspiciousness* of the fault. A *suspiciousness* value that is too close to the mean *suspiciousness* will result in a technique that provides no meaningful assistance, even if the *expense* would otherwise be acceptable.

To answer RQ2, two data-types are investigated: the actual *suspiciousness* values of the most easily localized fault, which provides insights on expected values across different programs, and the relationship between non-faulty code and faulty code's *suspiciousness* values, which improves understanding regarding the simultaneous identification of multiple faults. Note that hereafter references to all-non-faulty code and the latent-code are interchangeable. Further, by examining the code's latent *suspiciousness* score, concerns presented by previous researchers can be investigated (e.g., multiple faults increase the *suspiciousness* of all instructions).

4.2.1 Non-Fault and Fault Code Suspiciousness

Figure 5 displays this data for Sed, which is a sample from our six subjects that allows for a more thorough discussion of the patterns observed. For completeness, the remaining five *suspiciousness* figures (corresponding to the remaining five subjects) can be found in the Appendix (see Section 7) as Figure 13.

The horizontal axis represents the quantity of faults in the program and the vertical axis represents the *Suspiciousness* (as a percentage). The points

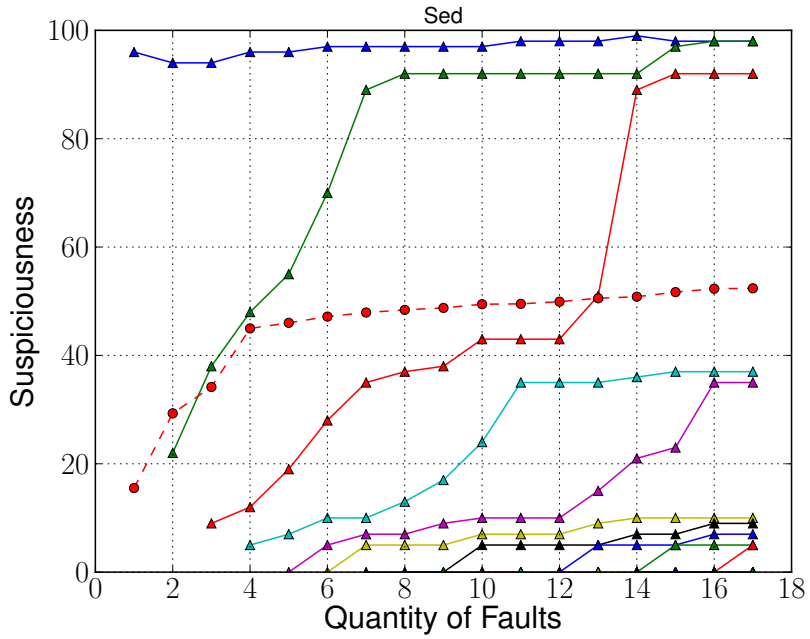

Fig. 5: Mean *suspiciousness* values for each fault of Sed, evaluated with a ranked list. The dotted line represents the mean latent *suspiciousness*.

connected by solid lines represent the median *suspiciousness* value assigned to the n-th placed fault in a sorted list of instructions, from most suspicious to least, while the dotted line represents the latent suspiciousness for a given quantity of faults. The top-most plot line represents the prominent fault, and thus the most suspicious; the second plot line from the top represents the second fault to be found by the technique, and so forth. However, there can only be a second fault when there are two faults in the program, meaning the second-highest plot line has no value at 1-fault on the horizontal axis — each line makes its introduction at one point further along this axis. The median, latent suspiciousness value assigned to *all* instructions in the program (faulty and non-faulty) is represented by the dotted line. Lastly, all plot points correspond with multiple different faults — points are classified by their position in the sorted *suspiciousness* measure, not by a fault identifier.

Figure 5 suggests a few common trends: (1) the latent *suspiciousness* initially increases, and then slows its rate of increase with six or more faults, (2) fault *suspiciousness* is positively correlated with fault quantity, and (3) a majority of faults are less suspicious than the latent *suspiciousness*. The interest of these trends will each be discussed in turn.

For half of our subjects, the latent *suspiciousness* spikes early and then has a very small rate of increase. Although the spike is non-trivial (a 10–

25% jump between quantity of faults 1 and 4), the latent *suspiciousness* only changes 4% between 9 and 18 on the horizontal axis. For the remainder of our subjects, the latent suspiciousness changes dramatically, with Replace and Space experiencing more than a three times increase in latent *suspiciousness*, and Schedule experiencing an increase than decrease in *suspiciousness*.

For all our subjects but Schedule, all prominent faults, many non-prominent faults, and the latent code continues to increase in *suspiciousness* as the quantity of faults increases.

The relationship between fault *suspiciousness* and latent *suspiciousness* is important. When a fault's *suspiciousness* score is below the latent *suspiciousness*, it becomes indistinguishable (from a CFL user's perspective) from non-faulty code. Even faults with a similar *suspiciousness* to the latent are likely to be ignored by a CFL user. Consider fault-quantity 9 where the latent *suspiciousness* is 48%, and the third prominent fault is 53%. This fault's *suspiciousness* is so similar to the general code-base that it is likely to be ignored. Only scores that differ substantially from the latent are likely to stand out or be inspected. Thus, although the prominent fault often scores at least 30% higher than the latent *suspiciousness*, Figure 5 exhibits that at the majority of fault quantities, all but a few faults are less suspicious than the latent code.

These results suggest that the increase of latent-code *suspiciousness* is somewhat program-specific. Although almost all our subjects experience an increase, the rate of increase appears to be different for different programs. Although some programs may have latent *suspiciousness* so high that intuition is that CFL techniques provide ineffective results, even for our subjects with this behavior (e.g., Replace) the prominent fault is statistically different (p-value less than 0.05) than the non-prominent faults, with a difference of their medians of roughly 5%.

One other important observation is that for all our subjects, at least two faults were more *suspicious* than the latent code (a p-value less than 0.05), and close in their *suspiciousness*. This result suggests that CFL techniques may be able to identify at least two faults simultaneously. We speculate that as the fault density increases, it is increasingly likely that two faults that are related in terms of infection implications contribute to the failure, allowing CFL techniques to identify them both as a single entity that is responsible for the failure. However, the vast majority of faults are far less *suspicious* than the latent code, meaning that some manner of iterative debugging is likely required for CFL users.

These results suggest that CFL techniques are not limited in their useful by fault quantity. In other words, although an increase in the *suspiciousness* value of the latent code should be expected, this increase will likely remain below that of the prominent fault by enough of a margin to enable the localization of at least one, potentially two faults.

For RQ2: The prominent fault's *suspiciousness* (and most of the non-prominent faults) increase as fault quantity increases until they near 100% and stabilizes. The latent *suspiciousness* also tends to increase, after an initial spike. Lastly, at high fault densities the majority of faults exhibit a lower suspiciousness than the latent suspiciousness, making them indistinguishable from normal code.

4.3 Research Question 3: Is fault type a predictor of CFL effectiveness?

To investigate whether fault type is an accurate predictor of CFL effectiveness, this experiment evaluates fault type against *expense*. This evaluation measures fault-type correlation with *expense*; signifying a predictability between type and CFL behavior. This experiment used four taxonomies in this experiment as described in Section 2.2: `Smith92`, `Firesmith92`, `Hayes94` and `Hayes11`.

Figure 6 characterizes each fault, and displays the *expense* for the Sed program. Note that due to the long descriptions of each fault type, numerical values are used to represent them — the index relating numerical value to fault-type description is given in the Appendix (see Section 7). Only Sed's figure is displayed in this section (which sufficiently represents all our subjects), but the remaining five fault-type figures (corresponding to the remaining five subjects) can be found in the Appendix (see Section 7) as Figure 14.

Note that each of these four subfigures represents Sed and contains the same values (i.e., the same boxes), but they are ordered and labeled differently based upon the four taxonomies. The horizontal axis enumerates the fault identifiers, labeled according to each taxonomy (in other words, each horizontal point represents a different fault) grouped according to fault type. The vertical axis represents *expense*. Each box represents the *expense* range of a particular fault for all versions of the program and fault quantities. Each box is generated using all instances of *expense* for a particular fault across all versions. Like our previous box plot, the center red line represents the median, the box represents the first and third quartiles, and the "whiskers" represent the min and the max values. Note that in circumstances where there appears to be no box, the entire box is a single value at 100%.

Each subfigure demonstrates that fault types appear to share no significant correlation within a fault type. In other words, this figure demonstrates that these samples cannot reject the null hypothesis, meaning that, statistically speaking, we cannot validate that these samples are from different populations. We confirmed this by performing a Kendall Tau test for each program, across each fault taxonomy, and found that *in all cases, there was no statistical correlation* between the *expense* range of any fault types for any of our six subjects (every result was near 0). 4b) Further, we performed a Mann Whitney U test and found that the results could not reject the null hypothesis (i.e. p-values greater than 0.05).

In an effort to better classify faults that have similar *expense*, we performed

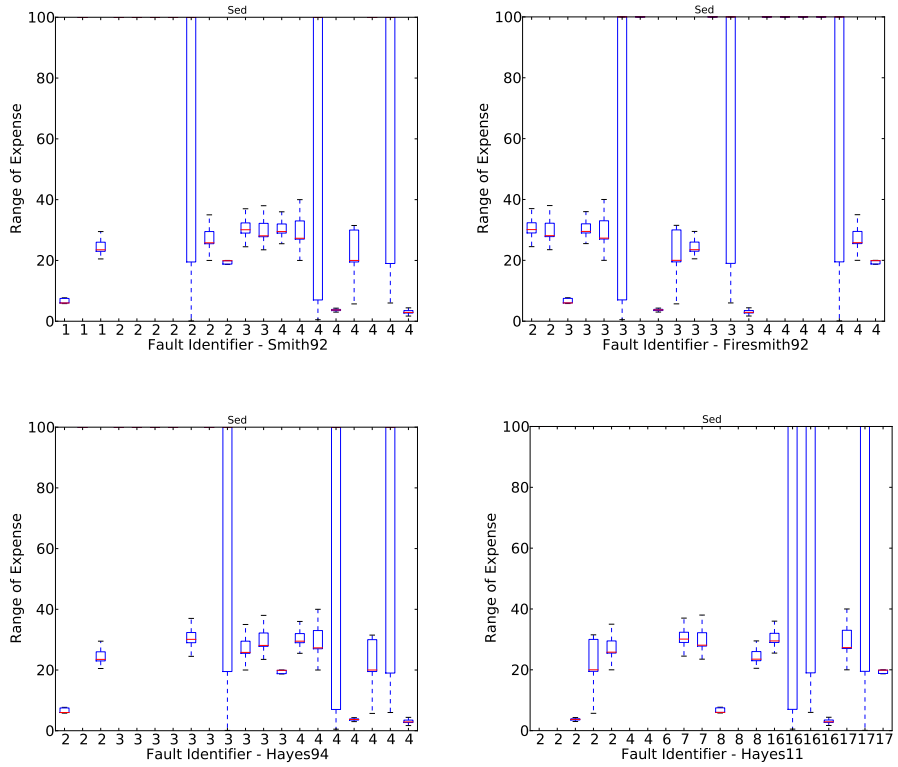

Fig. 6: The range of *expense* for every fault across each taxonomy with the Sed subject.

a correlation analysis of fault type to *expense*; unfortunately, no substantial commonalities within faults types were found. In addition, across subjects no meaningful correlations were found between fault types and *expense*. For example, when considering `Smith92`, we compared all bug ranges from classification 1, with those from classification 2, 3 and 4. In this way, each classification receives its own group of data.

Consider our fault types when categorized according to `Smith92`, which results in four identified fault types; some have large variances with high means, and others have low variances with low means. Additionally, faults with similar *expense* values are spread throughout different fault types. For example, types 1, 2 and 4 all have faults with low variance and low medians, and other faults completely at 100%. This behavior demonstrates a lack of consistency within a specific fault type, and in many cases, more similarity between different fault types. This behavior also exists between programs. All four taxonomies demonstrate that no single fault type has a representative *expense*.

Additionally, each fault across all six subjects was examined by hand for any qualitative similarities among the faults. Our analysis consistently revealed no qualitative similarities; fault-type similarities were a poor indicator of additional similarities. This anecdotal analysis revealed no patterns among fault types, or consistency within a fault type.

Our quantitative results and our qualitative investigation suggest that fault type has little to no correlation with *expense* values, and that *expense* is more influenced by nuanced circumstances (e.g., the exact variable in question, the method calling the faulty code, the fault's location during execution).

Unfortunately, these results indicate that these type of general taxonomies are insufficient to identify the way a fault might behave. This suggests the need for either, more domain/language-specific fault taxonomies or for different, more context-sensitive identifiers to enable the automatic classification of fault behaviors.

For RQ3: Usage of these taxonomies suggests that these fault types have little to no correlation with CFL *expense*, and that these fault types are not a determining factor in localizability.

4.4 Research Question 4: What are the practical ramifications of fault-localization interference on CFL?

To illustrate the practical ramifications of fault localization interference on CFL, this section illustrates with an example from the Gzip. This example allows a more fine-grained understanding of how FLI impacts a system, and how CFL results can change as more faults are introduced.

Results presented thus far have been aggregated across many versions — up to 1000 for every fault quantity of each subject. To facilitate a more complete understanding of the practical impact of FLI, Figure 7 presents a unique sequence of faults from the Gzip program as the fault quantity is altered. More simply, this section presents a small, focused study from real software to illustrate the functional challenges imposed by FLI. The vertical axis represents *expense* while the horizontal axis is the quantity of faults in the program. Each line represents a unique fault in the program and because faults are introduced one-at-a-time, a new line is added at each horizontal point. For the sake of replication, the sequence of faults used, in the order of introduction are ⟨19,3,20,8,10,5,12,1,13,15,4,16,11,17,9,14,18⟩.

With only a single fault the *expense* is 11%. Upon the introduction of the fourth fault, the *expense* for the second introduced fault jumps to 100%. When the eighth fault is introduced, two of the existing faults' expenses jump drastically from roughly 10% to above 20% and two drop drastically from around 30% to near 5%.

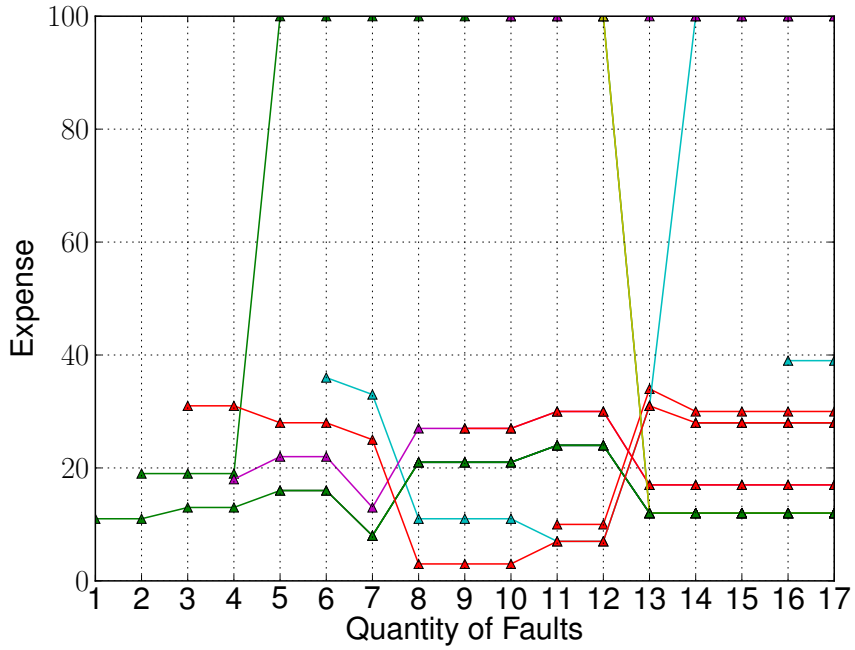

Fig. 7: The *expense* for individual faults following a particular sequence of 1–17 faults in Gzip. The *expense* for a particular fault can drastically change as a new fault is introduced or removed.

In these cases, additional faults interfere with the localizability of previous faults — that is, the presence of one fault limits or heightens the ability of CFL to effectively localize the other faults (i.e., FLI). One of the most pronounced examples of FLI is observed when one fault causes another fault's *expense* to reach 100% — this occurs twice, first at five faults, and again at fourteen faults. In these cases, obscured faults are no longer executed by any failing test cases due to control flow changes.

This study presents results for a particular 1–17 sequence of fault introductions. This particular scenario gives the reader a "microscopic" view of the effects that were observable throughout nearly all of the inspected versions — these effects cannot be adequately presented in all other "macroscopic" plots that aggregate results from over 70,000 multi-fault versions. However, in our anecdotal experience, these interference effects are typical across the other versions.

One insight here is that a single introduction or removal of a fault can drastically alter which faults are currently visible. This suggests that developers ought to be thorough and repetitive about running their test suites after changes are made or faults are fixed. Indeed, this example clearly demonstrates the need for efficient and thorough regression testing, as the removal of a single

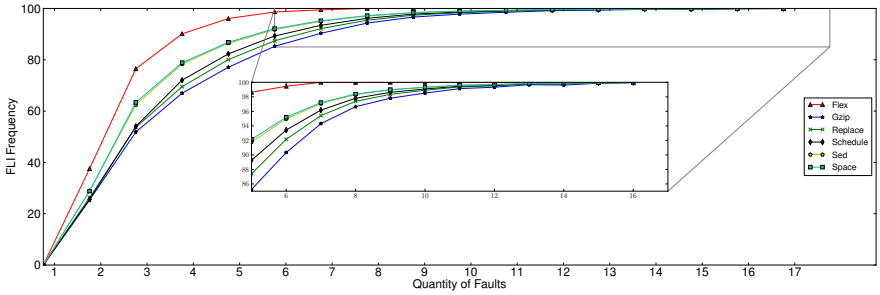

Fig. 8: The frequency of FLI occurrences as the quantity of faults increases. All subjects quickly rise to and then slowly approach 100%.

fault may enable the identification of others that were previously concealed.

An additional insight, is that it is currently unclear which faults tend to cause FLI and which are interfered with. In other words, this example highlights that some faults when interfered with are completely obfuscated, resulting in an *expense* of 100%, whereas others are only slightly impacted raising their *expense* by 10%. Additional investigations should investigate whether there is some correlation between those faults which are completely obfuscated, and those which are only slightly affected.

For RQ4: FLI can cause faults to become unlocalizable, only slightly obscure them, or improve their ability to be localized. In other words, the impact of multiple faults on a *specific* fault's localizability is unpredictable in occurrence and intensity.

4.5 Research Question 5: How often does fault-localization interference occur?

First, recall that FLI is defined as occurring when a fault's *expense* increases (i.e., the fault becomes more difficult to localize) due to the presence of at least one other fault. Also, recall that to discuss how often FLI occurs at varying fault densities the term FLI *frequency* means the rate at which FLI occurs over a given sample.

The FLI frequency is displayed as a percentage on the vertical axis of Figure 8, and the horizontal axis represents the quantity of faults. Note that for this evaluation, only instances of interference are measured: if one fault experiences 80% interference and another fault experiences 13%, both are categorized as experiencing interference. Additionally, lines are only drawn to highlight the trends of each subject, there are no actual values between discrete points.

For this evaluation, FLI is classified as an *expense* increase of at least an *order of magnitude*. In other words, if any fault in a version has an *expense* increase of at least an order of magnitude, then it is classified as experiencing FLI. An order of magnitude is used because it represents a substantial and likely noticeable loss of effectiveness.

By six faults, all subjects experience interference at least 80% of the time. Figure 8 demonstrates a positive correlation between fault density and frequency of interference. To confirm this, we performed a Kendall Tau test between adjacent x-coordinates (e.g., quantity of faults 1,2; 5,6; 11,12...) and found that in each case, the values was greater than 0.93, indicating a strong positive correlation between quantity of faults, and frequency of FLI occurance. By 15 faults, 100% of versions in 100% of our subjects experience FLI. These results demonstrate that interference among faults occurs frequently.

Additionally, by ten faults, when performing a Mann Whitney considering the FLI frequency experienced by all subjects, we were unable to reject the null hypothesis (i.e. p-values were greater than 0.05) meaning we cannot draw statistical conclusions regarding these populations. This means that regardless of subject, after ten faults, the frequency of FLI experienced will be identical for all subjects.

These results suggest potential impact on developers looking for a specific fault. Due to the high level of interference, if the fault does not cause the failure, it will be obfuscated, meaning that CFL might be unable to locate the fault a developer is looking for. We speculate that without techniques that enable the separation of these failure causes (e.g., failure clustering) or improvements to CFL, existing techniques will be unreliably successful at identifying any arbitrary specific fault.

Additionally, these results suggest the importance of understanding fault interaction. This figure clarifies that at larger quantities of faults, the interaction causes obfuscation among a majority of faults. Although there is some existing work that examines this type of interaction explicitly (e.g., [6, 10]), these are not sufficient to completely explain this type of behavior. Indeed, this result highlights the needs for further studies that examine the potential and realized complexities that exist when multiple faults interact to change program behavior, and the spectra by which it can be analyzed.

For RQ5: FLI is prevalent, occurring in over 80% of all multi-fault versions containing at least six faults. Further, by ten faults, our programs were not statistically different in terms of the frequency in which FLI was experienced.

4.6 Research Question 6: What is the typical magnitude of fault-localization interference?

Although research question RQ5 addresses how often FLI occurs, research question RQ6 addresses the impact of an FLI occurrence on CFL. In other words, the evaluation of magnitude is used as an approximation of FLI's influence on debugging. In this discussion, recall that magnitude is defined in Section 3.1.

To investigate the magnitude of FLI, this experiment measures the *expense increase* against fault quantity. For each fault that experiences FLI, the difference in *expense* between the single-fault version and the multi-fault version at varying levels of fault density is measured. In other words, to calculate the *expense increase*, we leverage the version containing *only* the fault in question, and another containing multiple faults (including the fault in question) and compare the increase in *expense* for each individual fault against all multiple fault version.

Figure 9 displays the magnitude of FLI on fault *expense* as fault density increases. The vertical axis represents the *expense* increase due to FLI, and the horizontal axis represents the fault quantity. Like the previous box plots, the red line is the median, the box is the first and third quartile, and the "whiskers" are the max and min.

There is an initial increase in the median, the first and third quartile. Along with the initial jump, there is a monotonic increase in the third quartile and the median as they approach 100% interference (N.B., at 100%, faults are no longer executed): the third quartile reaches 100% at seven faults and the median reaches it at 15 faults. The third quartile and median increase are contrasted by the first quartile that remains mostly constant after six faults.

This graph illustrates that if a fault experiences interference when there are at least 15 faults in the program, it will likely be entirely obfuscated and unlocalizable by CFL techniques — in fact, it is not even executed. Additionally, if a fault experiences FLI at lower fault quantities, its localizability will still be reduced enough to make it likely unlocalizable (as other faults will be found first). This is substantial impairment — significant enough to make CFL results unusable for developers attempting to localize the specific obfuscated fault (i.e., a developer specifically targeting a fault that is experiencing interference); however, note that in such a case, such fault is not causing any failures and thus not likely to be even attempted to be debugged.

These interference patterns suggest that as fault quantity increases, the maximum impact increases (towards 100%) while the minimum remains unchanged (around 0.5%). Further, a Mann Whitney U test found that each adjacent quantity of faults (e.g., 1,2; 5,6; 11,12...) was statistically different from one another (with p-values < 0.05). Also, we performed a Kendall Tau test for each adjacent pairwise quantity of faults and found that each value was above 0.75 (with most being above 0.9).

One insight from these results is that debugging is likely to be an iterative process. Consider that at higher fault quantities, if a fault experiences FLI

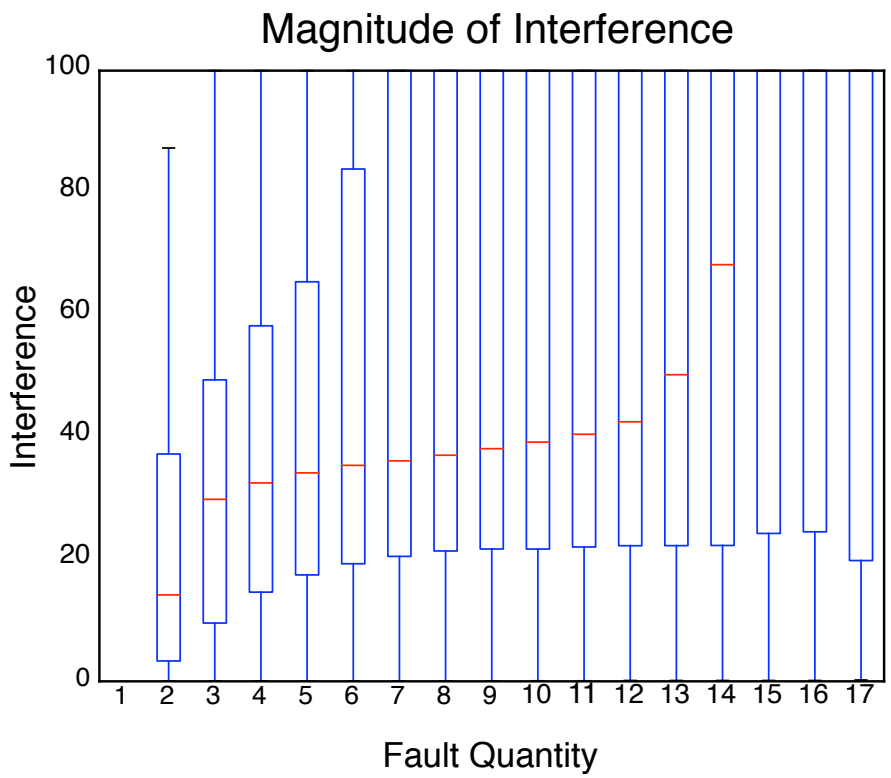

Fig. 9: The percentage of interference that an individual fault experiences as the quantity of faults increases. There is a monotonic increase of the median and the third quartile until 100% interference is reached.

(which RQ5 found to be over 80%), on average that fault will be completely obfuscated, meaning that it is not even executed. Thus, even if a developer were to fix every fault that caused a failure and that is localizable (*expense* less than 100%), they would be required to rerun their tests again to identify the obfuscated faults.

Further, this presents complications for regression testing, wherein developers might become confused and believe they introduced new bugs. Consider that a developer has two test cases and only one fails. Upon *fixing* a fault, they find that while the failing test case now passes, the passing test case now fails. Intuition would suggest that new faults were introduced, but these results suggest that such behavior is likely.

For RQ6: There is a positive correlation between fault quantity and FLI magnitude. Higher fault densities correlate with an increased interference and an increased probability of an entirely obfuscated fault. In other words, FLI can cause the localization of arbitrary pre-specified faults to be difficult at higher fault quantities.

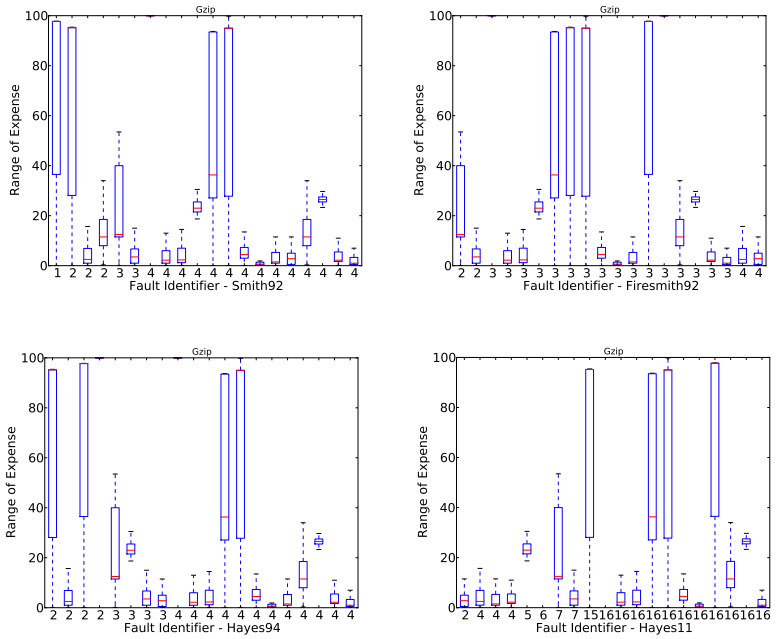

Fig. 10: The range of interference that was observed for every fault for each taxonomy with Gzip.

4.7 Research Question 7: Does fault type affect the likelihood of causing interference or being subject to interference?

This section investigates potential correlations between fault type and FLI by comparing fault types against the interference they experience.

To classify each fault type, this experiment uses the same four fault taxonomies discussed in Section 2.2: `Smith92`, `Firesmith92`, `Hayes94`, and `Hayes11`. Note that due to the long descriptions of each fault type, numerical values are used to represent them, though the index relating numerical value to fault-type description is given in the Appendix (see Section 7). For simplicity of analysis and discussion, only Gzip's figures is shown in this section (which sufficiently represents all our subjects), but the remaining five fault-type figures

(corresponding to the remaining five subjects) can be found in the Appendix (see Section 7) as Figure 15.

In Figure 10 the horizontal axis represents the identifier for each fault type, and is labeled with the taxonomy used, and the vertical axis is the interference that the fault experienced across all multi-fault versions. Like previous box plots, the center red line represents the median, the box represents the first and third quartiles, and the "whiskers" represent the min and max. Note that each subfigure represents the same subject and data (i.e., the bars are the same in all four graphs), the difference lies in the taxonomy used, and the ordering of the bars within each subfigure.

Each box represents a single unique fault, across all multi-fault versions of Gzip. In every instance where the *expense* of a fault is increased (i.e., the fault experiences interference), the *amount of increase* is stored so that each box represents the increase experienced.

Some faults have a large interference variance (large boxes) — i.e., they experienced many different interference values — and others have a small interference variance (small boxes) — i.e., they consistently experienced the same degree of FLI. Note that when there appears to be no box, the entirety of the box is a single line at 100%, which means that when the fault is obfuscated, it becomes completely unexecuted.

As in the previous study leveraging fault taxonomies, we performed an Mann Whitney U test for each subject, utilizing each fault taxonomy, and found that regardless of classification, the null hypothesis could not be rejected for any fault type (p-values greater than 0.05), meaning no conclusion could be drawn regarding the statistical difference of these sample populations Further, when performing a Kendall Tau test between a fault type and its interference, we found scores near 0. For example, when considering `Smith92`, we compared all bug ranges from classification 1, with those from classification 2, 3 and 4. In this way, each classification receives its own group of data.

Across all four taxonomies there is no correlation between fault type and interference. For our subjects, faults with different types experience the same degree of interference. For example, looking at `Firesmith92`, considering type-3 faults, some instances have large variances with high means, and other instances have small variance with low means. Additionally, faults with similar degrees of interference spread throughout each fault type. For example, type-2, -3 and -4 all have faults with low variance and low medians.

Further, there is significant diversity within a fault type in regards to the degree of interference that is experienced. Looking at `Hayes94`, fault types 2, 3 and 4 have some low and tight bars, and some wide bars. This diversity within a fault type demonstrates a lack of consistency, meaning that there is no "typical" value for a fault type.

We also analyzed fault-type behavior between programs, and again found no correlation to interference. All four of the taxonomies demonstrate that no single fault type has a representative interference range.

Lastly, a manually inspection was performed for each fault across all six subjects to identify any qualitative similarities. This manual analysis consis-

tently revealed that fault-type similarities were a poor indicator of interference similarities. The quantitative results and our qualitative analysis revealed no patterns among fault types, or consistency within a fault type.

One meaningful impact of this study is the understanding that identifying faults likely require investigating more context specific behavior than "fault type", or that general fault types (like those used in our study) are potentially insufficient to categorize faults. Indeed, these results suggest the need for more mature fault taxonomies, or for spectra that allow for more context sensitive classification.

For RQ7: Our data suggests that fault types have little to no correlation with FLI. Fault type is not a determining factor in causing interference, or experiencing interference.

4.8 Results Summary

Due to the multi-faceted nature of the results presented in Section 4, this section presents a brief overview of the findings grouped by CFL, FLI, and fault type.

CFL Results. Fault density only has a significant impact upon CFL effectiveness: the *expense* of the prominent fault was statistically likely to degrade, with an impact on the median of 2% between a program containing only a single fault and a program containing as many as 17 faults. Further, regarding the *suspiciousness* value, at least two faults were always statistically higher than the *suspiciousness* of the non-faulty code. However, although at least one fault was localizable, the majority of the remaining faults (at higher fault densities) had *suspiciousness* and *expense* scores too poor to enable reasonable use of CFL techniques. In other words, even in the presence of multiple faults, CFL techniques can still perform comparable to its effectiveness for at least one fault, although which fault will be found is unknown, and the majority of faults cannot be simultaneously localized. However, our studies showed that typically the non-prominent (and unlocalizable) faults were often not even executed, and as such would not have been the cause of witnessed failures, thus being unlikely to be the target for debugging.

FLI Results. FLI is prevalent and has a substantial impact upon the *expense* of non-prominent faults. Indeed, there is a positive correlation with fault density and both FLI frequency and magnitude. At higher fault densities, 100% of our multi-fault versions experienced FLI with a mean approaching 100% obfuscation. This means that CFL techniques are likely to perform poorly when developers are looking for arbitrary faults in multi-fault programs, if they were known to exist.

Fault Type Results. The fault types examined are not a predictor of either CFL *expense* or FLI. These four fault taxonomies displayed no correlation between fault type and both FLI frequency or FLI magnitude. Additionally, our data exhibited no correlation between fault type and CFL effectiveness.

5 Threats to Validity

One difficulty in creating external validity for this work pertains to generalization. Our evaluation measures from one to seventeen faults, thus any generalizations for programs with more than seventeen faults are only projections. However, because mean line representing all our subjects in Figure 2 manifests patterns that are fairly smooth and consistent, it appears unlikely the behavior will drastically change in the presence of more faults.

An additional concern with generalizability is that our evaluation only contains six subjects. With the large diversity of existing software (in size and complexity), it is difficult to examine a sufficiently large sample to ensure consistent results for all software. However, because four of the six programs in our experiments are real-world software, and more than 72,000 versions were created, executing over 1,600 million test cases, we expect the results from our study to at least be representative of programs with similar size or complexity.

One final concern in generalizability is that our evaluation utilizes mutants. The nature of this study requires more faults than are delivered by SIR, imposing a need for mutation [15]. Although it is true that mutants are not "real" faults, Ali *et al.* found that simple, single-line mutants behave similarly to real faults [2]. Further, Ali *et al.* assert that, "single-mutant-line mutants...behaved very similarly to the real faults...with respect to Tarantula," and that their results, "showed no reason to believe that mutants are unsuitable as candidates for faulty versions for the purpose of studying FL [fault localization] algorithms," [2]. Although only a single study, these findings suggest the suitability of using mutants for this type of evaluation. Further, a study by Offutt *et al.* indicates that properly created single-line mutants can be both "effective and efficient" in their ability to test a program [26]. Thus, to minimize this concern, all of our mutants were made in accordance with Offutt *et al.*' suggestions.

One threat to construct validity is that we measure expense as the percentage of lines a developer must examine to find the fault. For suitably large programs, 1% could easily be 1000 LoC or larger, leading to information overload. However, although 1% of a large program is still sizable, if the required search space for a developer to examine is reduced by 99%, then CFL has successfully reduced potential information overload. This search-space reduction has the potential to greatly reduce the time spent localizing the fault. Moreover, this work investigates the impact of fault interactions on fault-localization effectiveness — and does not target the evaluation of other downstream usability effects. Moreover, a recent study by Parnin and Orso [27] found that developers were limited in their ability to utilize ranked lists of statements for debugging

when presented with such lines without the context of their original positions in source-code files. Such findings are unsurprising and highlight the need for visualizations or other user interfaces that allow developers to understand the context of the fault-localization results (such as TARANTULA [21]). Nevertheless, the experiments described in this paper are designed to study the ways in which multiple faults can interact to affect spectra-based fault-localization effectiveness, outside the scope of user interfaces or developer behavior, and hence, such issues are out of scope. Regardless, rigorous experimentation that specifically target such usability issues remain an issue for future study.

6 Conclusions

This paper presents an experiment that examines how fault quantity affects CFL effectiveness, fault-localization interference, and whether fault-types are determining factors in either a CFL effectiveness, or FLI values.

This work provides evidence that is contrary to a commonly held belief that CFL techniques cannot perform effectively in the presence of multiple faults. On one hand, beliefs and intuitions of faults creating "noise" or interference that inhibits the effectiveness of CFL are well founded — this interference exists and is prevalent. On the other hand, this interference has limited impact upon the localizability of at least one, prominent fault. Our results demonstrate that CFL techniques may continue to be effective in the presence of multiple faults in spite of interference. Unless the goal is the simultaneous debugging of multiple faults, or the localization of an arbitrary fault, CFL tools can still be expected to perform well, regardless of fault quantity, with only a small loss in effectiveness.

This work formally defines fault-localization interference, gives examples of it in real-world software, and provides data that suggests it to be a relevant concern for CFL. This study found a positive correlation between fault density and FLI frequency, meaning that as fault density increases the probability that an obscuring fault is active increases until it is nearly inevitable. Additionally, our results show a positive correlation between the magnitude of interference observed for a fault and the fault quantity.

Two investigations into how the *types* of the faults affects their localizability in multi-fault programs and their interference among those faults were performed. Our data showed no significant correlation or connection — the fault type appears independent of localizability and interference. Our data instead suggests that the determining factors for localizability and interference may be more execution-context-specific than fault-type alone (e.g., fault-execution order, data-flow relationships, value sensitivity), which suggests open areas for future study.

Acknowledgements This material is based upon work supported by the National Science Foundation, through Award CCF-1116943 and through Graduate Research Fellowship under Grant No. DGE-0808392.

References

1. Abreu, R., Zoeteweij, P., Van Gemund, A.J.: On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007, pp. 89–98. IEEE (2007)
2. Ali, S., Andrews, J.H., Dhandapani, T., Wang, W.: Evaluating the accuracy of fault localization techniques. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, pp. 76–87. IEEE Computer Society (2009)
3. Arumuga Nainar, P., Liblit, B.: Adaptive bug isolation. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, pp. 255–264. ACM (2010)
4. Clark, S., Cobb, J., Kapfhammer, G.M., Jones, J.A., Harrold, M.J.: Localizing sql faults in database applications. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 213–222 (2011)
5. Cleve, H., Zeller, A.: Locating causes of program failures. In: Proceedings of the 27th international conference on Software engineering, pp. 342–351. ACM (2005)
6. Debroy, V., Wong, W.E.: Insights on fault interference for programs with multiple bugs. In: Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on, pp. 165–174. IEEE (2009)
7. Denmat, T., Ducassé, M., Ridoux, O.: Data mining and cross-checking of execution traces: a re-interpretation of jones, harrold and stasko test information. Tech. rep. (2005)
8. Dickinson, W., Leon, D., Podgurski, A.: Finding failures by cluster analysis of execution profiles. In: Proceedings of the International Conference on Software Engineering (2001). URL `http://portal.acm.org/citation.cfm?id=381473.381509`
9. Dickinson, W., Leon, D., Podgurski, A.: Pursuing failure: the distribution of program failures in a profile space. In: Proceedings of the International Symposium on Foundations of Software Engineering (2001). DOI http://doi.acm.org/10.1145/503209.503243
10. DiGiuseppe, N., Jones, J.A.: Fault interaction and its repercussions. In: Software Maintenance (ICSM), 2011 27th IEEE International Conference on, pp. 3–12. IEEE (2011)
11. DiGiuseppe, N., Jones, J.A.: Fault interaction and its repercussions. In: Software Maintenance (ICSM), 2011 27th IEEE International Conference on, pp. 3–12. IEEE (2011)
12. DiGiuseppe, N., Jones, J.A.: On the influence of multiple faults on coverage-based fault localization. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, pp. 210–220. ACM (2011)
13. DiGiuseppe, N., Jones, J.A.: Concept-based failure clustering. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, p. 29. ACM (2012)
14. DiGiuseppe, N., Jones, J.A.: Software behavior and failure clustering: An empirical study of fault causality. In: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on, pp. 191–200. IEEE (2012)
15. Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering **10**(4), 405–435 (2005)
16. Hayes, J.H.: Testing of object-oriented programming systems (oops): A fault-based approach. In: Object-Oriented Methodologies and Systems, pp. 205–220. Springer (1994)
17. Hayes, J.H., Chemannoor, I.R., Holbrook, E.A.: Improved code defect detection with fault links. Software Testing, Verification and Reliability **21**(4), 299–325 (2011)
18. Jones, J.A.: Semi-automatic fault localization. Ph.D. thesis, Georgia Institute of Technology (2008)
19. Jones, J.A., Bowring, J.F., Harrold, M.J.: Debugging in parallel. In: Proceedings of the 2007 international symposium on Software testing and analysis, pp. 16–26. ACM (2007)
20. Jones, J.A., Harrold, M.J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pp. 273–282. ACM (2005)
21. Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of test information to assist fault localization. In: Proceedings of the 24th international conference on Software engineering, pp. 467–477. ACM (2002)

22. Kung, D.C., Gao, J., Kung, C.H.: Testing object-oriented software. Tech. rep. (1998)
23. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. pp. 15–26. ACM (2005)
24. Liu, C., Han, J.: Failure proximity: a fault localization-based approach. In: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 46–56. ACM (2006)
25. Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.P.: Sober: statistical model-based bug localization. pp. 286–295. ACM (2005)
26. Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C.: An experimental determination of sufficient mutant operators. ACM Transactions on Software Engineering and Methodology (TOSEM) **5**(2), 99–118 (1996)
27. Parnin, C., Orso, A.: Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, pp. 199–209. ACM (2011)
28. Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., Wang, B.: Automated support for classifying software failure reports. In: Software Engineering, 2003. Proceedings. 25th International Conference on, pp. 465–475. IEEE (2003)
29. Renieres, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on, pp. 30–39. IEEE (2003)
30. Santelices, R., Jones, J.A., Yu, Y., Harrold, M.J.: Lightweight fault-localization using multiple coverage types. In: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pp. 56–66 (2009)
31. Smith, M., Robson, D.: A framework for testing object-oriented programs. Journal of Object-Oriented Programming **5**(3), 45–53 (1992)
32. Srivastav, M., Singh, Y., Gupta, C., Chauhan, D.S.: Complexity estimation approach for debugging in parallel. In: Computer Research and Development, 2010 Second International Conference on, pp. 223–227. IEEE (2010)
33. Vessey, I.: Expertise in debugging computer programs: A process analysis. International Journal of Man-Machine Studies **23**(5), 459–494 (1985)
34. Voas, J.: Pie: a dynamic failure-based technique. Software Engineering, IEEE Transactions on **18**(8), 717–727 (1992). DOI 10.1109/32.153381
35. Wong, W.E., Debroy, V.: A survey of software fault localization. University of Texas at Dallas, Tech. Rep. UTDCS-45-09 (2009)
36. Yu, Y., Jones, J.A., Harrold, M.J.: An empirical study of the effects of test-suite reduction on fault localization. In: Proceedings of the 30th international conference on Software engineering, pp. 201–210. ACM (2008)
37. Zeller, A.: Isolating cause-effect chains from computer programs. In: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering, pp. 1–10. ACM (2002)
38. Zeller, A.: In: Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann (2009)
39. Zheng, A.X., Jordan, M.I., Liblit, B., Naik, M., Aiken, A.: Statistical debugging: simultaneous identification of multiple bugs. In: Proceedings of the 23rd international conference on Machine learning, pp. 1105–1112. ACM (2006)

7 Appendix

7.1 Appendix: Fault-Type to Number Index

`Smith92`

1. Inter-routine conceptual
2. Inter-routine actual
3. Intra-routine conceptual
4. Intra-routine actual

`Firesmith92`

1. Incorrect visibility
2. Missing component
3. Inconsistent component
4. Incorrect allocation/deallocation of resources
5. Does not meet requirements

`Hayes94`

1. Abstraction
2. Encapsulation
3. Modularity
4. Hierarchy

`Hayes11`

1. Data Declaration
2. Data Initialization
3. Data representation
4. Data accessing
5. Incorrect equation
6. Wrong manipulation
7. Incorrect/missing processing
8. Unnecessary processing
9. Rampaging go to
10. Incorrect labels
11. Dead-end code
12. Duplicate logic
13. Unachievable path
14. Incorrect initial value
15. Incorrect terminal value
16. Incorrect control value processing
17. Incorrect exception exit processing
18. Illogical conditions or impossible cases
19. Incorrect module interaction
20. Incorrect module-external data structure
21. Incorrect input parameters
22. Large response time
23. Lack of naturalness
24. Inconsistency
25. Redundancy
26. Complexity
27. Lack of flexibility
28. Non-supportiveness
29. Unpredictable flows
30. Visual stimulation
31. Platform
32. Wrong file included
33. Incorrect environment variable setting
34. Documentation

7.2 Appendix: Expense for Prominent Fault

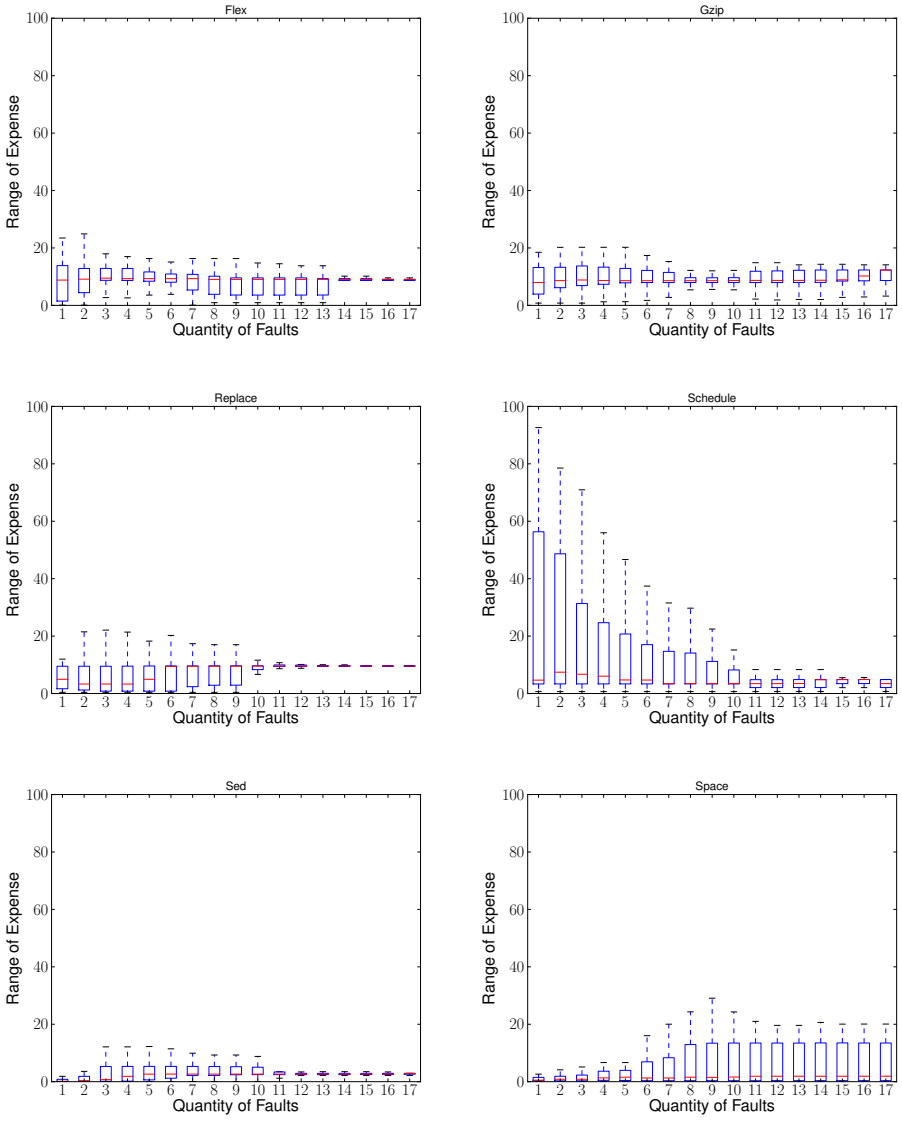

Fig. 11: The range of *expense* that was observed for the most localizable fault at each discrete quantity of faults.

7.3 Appendix: Median Expense for All Faults

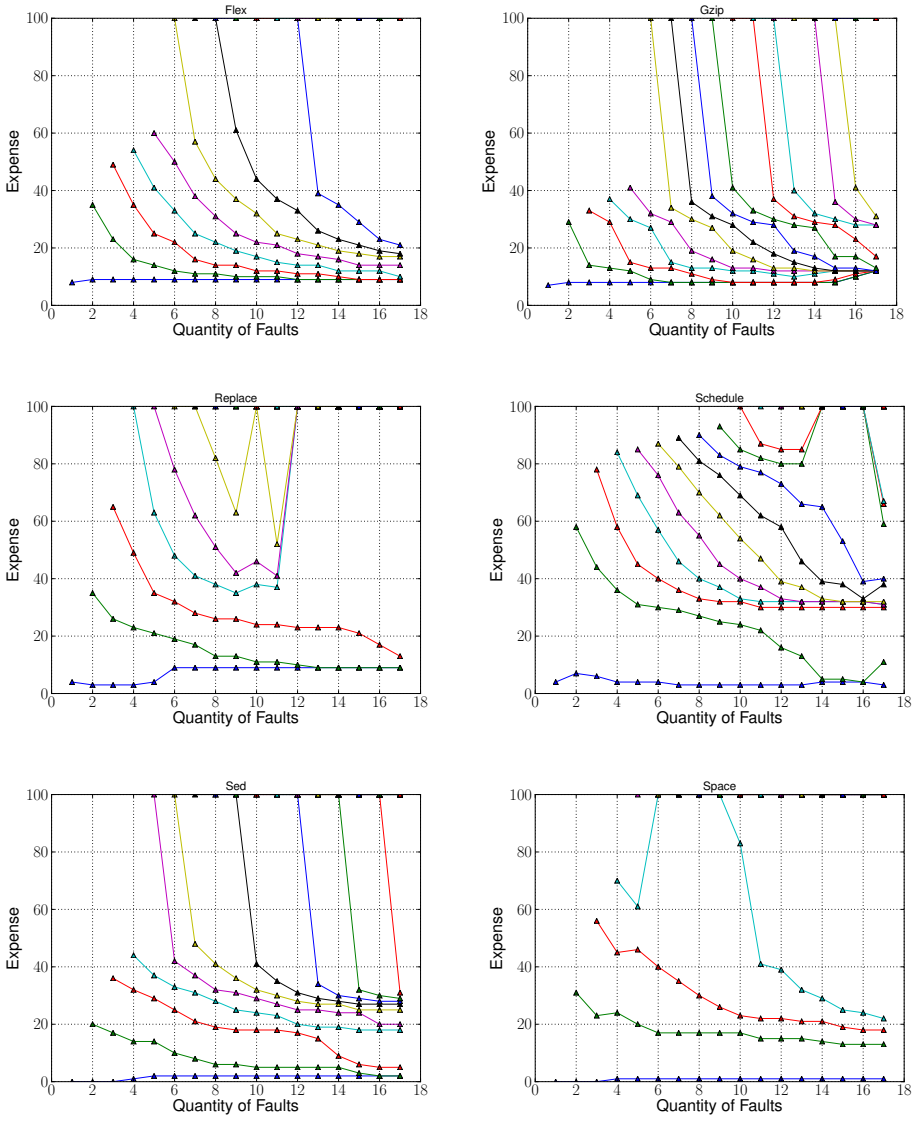

Fig. 12: The aggregated *expense* for each fault, evaluated with a ranked list, in all our programs. The lowest plot line represents the fault with the least *expense* (i.e., the first localized), and each higher line represents the next found faults..

7.4 Appendix: Median Suspiciousness for All Faults

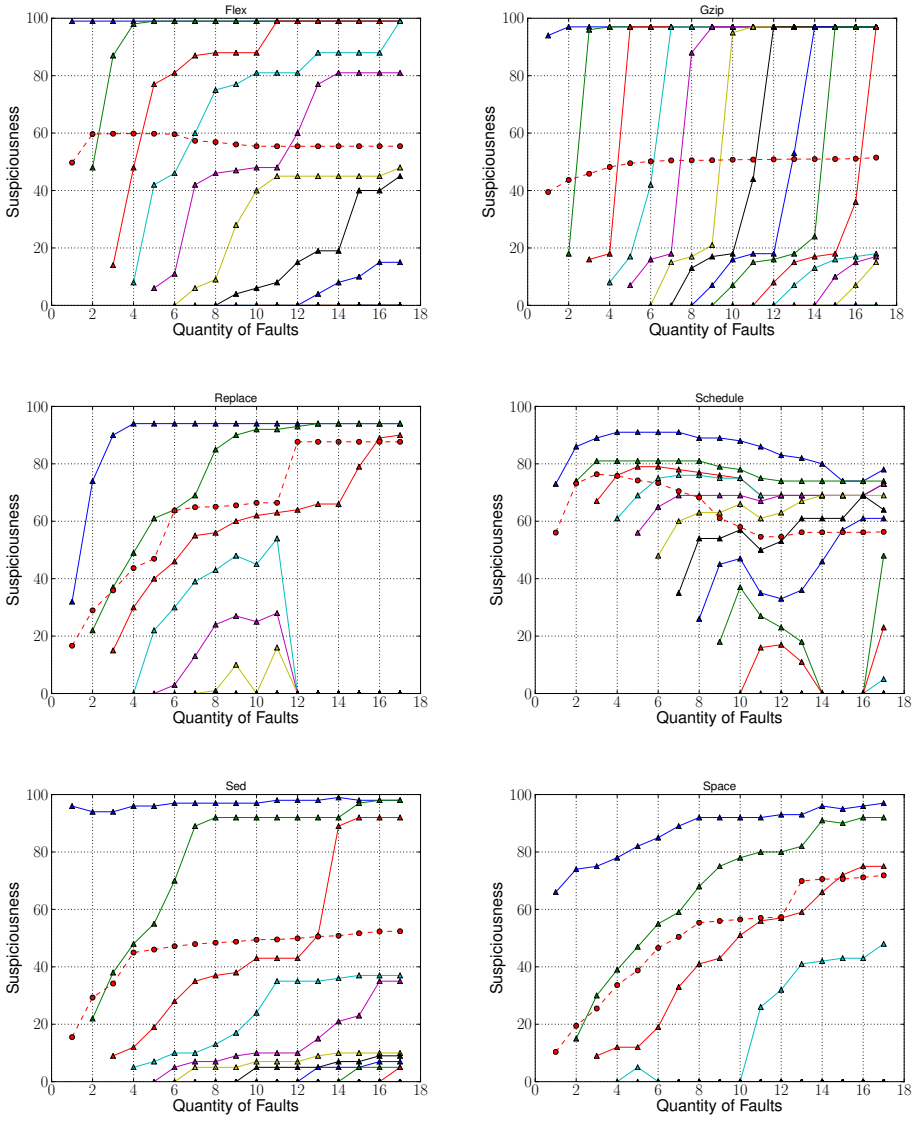

Fig. 13: Mean *suspiciousness* values for each fault, evaluated with a ranked list, across all our programs. The dotted line represents the mean *suspiciousness* of all instructions in the program.

7.5 Appendix: Expense Distribution per Fault Type

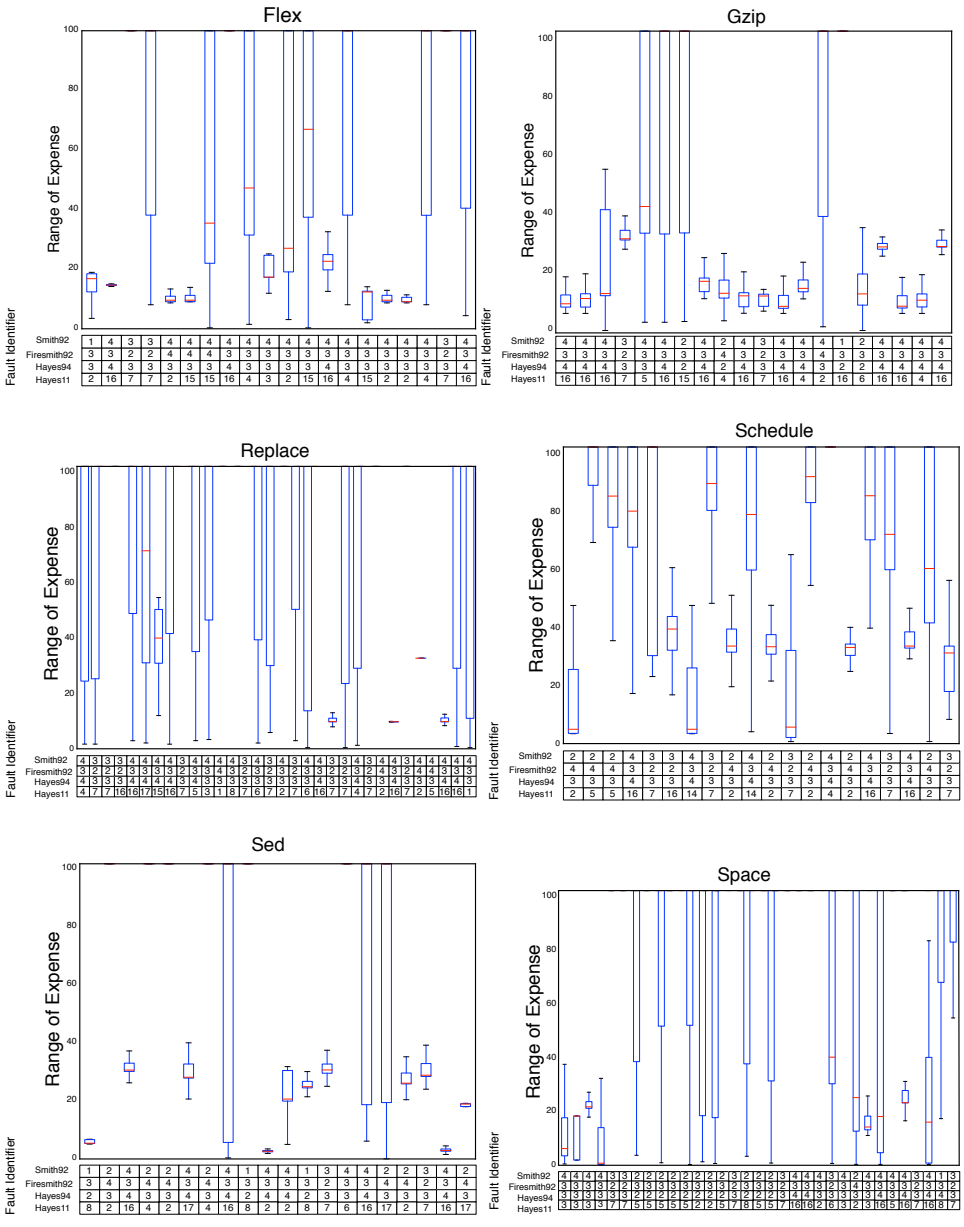

Fig. 14: The range of *expense* that was observed for every fault with each of the four taxonomies used, for all our programs.

7.6 Appendix: Fault Interference Distribution per Fault Type

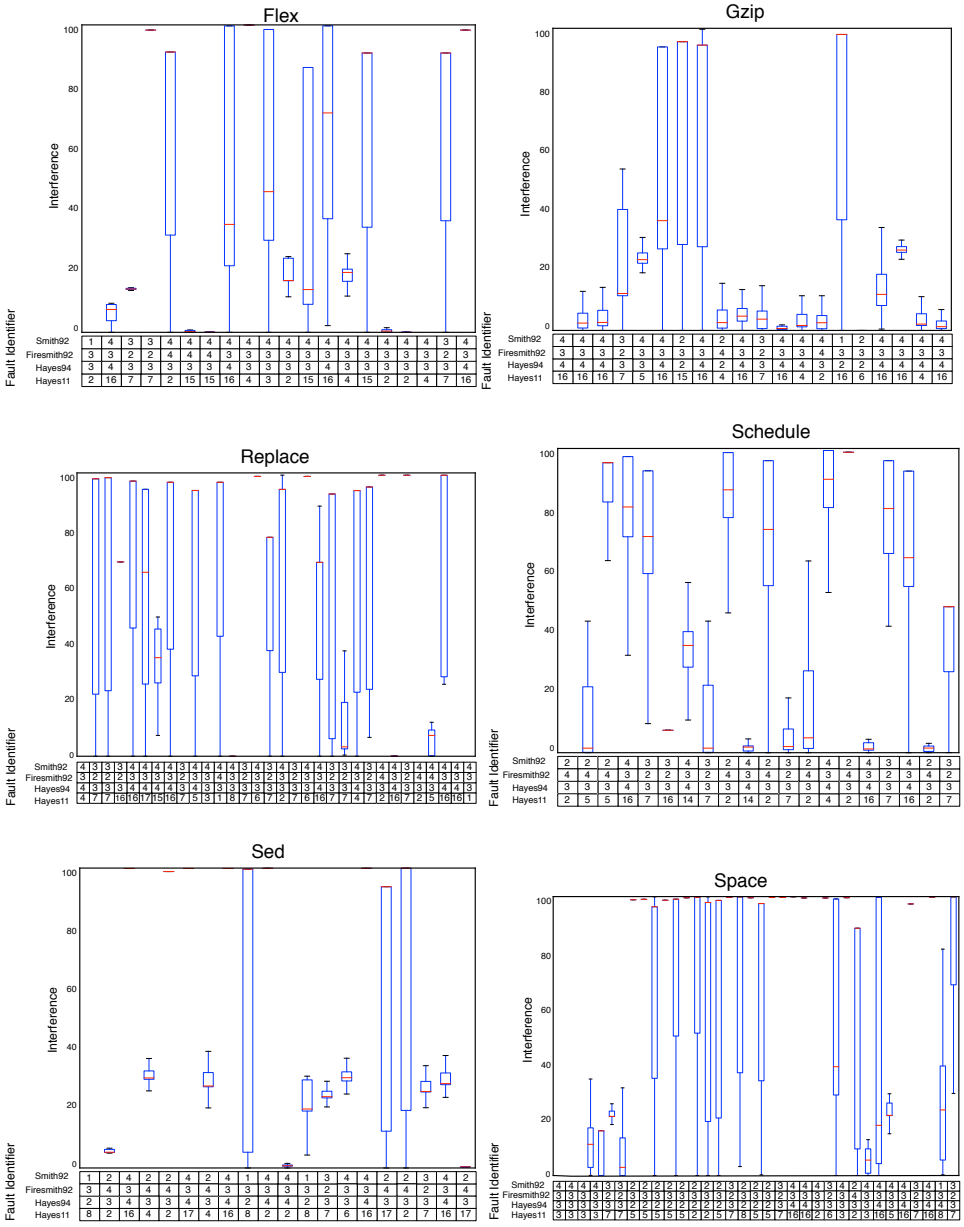

Fig. 15: The range of interference that was observed for every fault for each of the four taxonomies used.