Localizing Failure-Inducing Program Edits Based on Spectrum Information

Lingming Zhang, Miryung Kim, Sarfraz Khurshid
Electrical and Computer Engineering
The University of Texas at Austin
Email: zhanglm10@gmail.com, {miryung, khurshid}@ece.utexas.edu

***Abstract*—Keeping evolving systems fault free is hard. Change impact analysis is a well-studied methodology for finding faults in evolving systems. For example, in order to help developers identify failure-inducing edits, Chianti extracts program edits as atomic changes between different program versions, selects affected tests, and determines a subset of those changes that might induce test failures. However, identifying real regression faults is challenging for developers since the number of affecting changes related to each test failure may still be too large for manual inspection. This paper presents a novel approach FAULTTRACER which ranks program edits in order to reduce developers' effort in manually inspecting all affecting changes. FAULTTRACER adapts spectrum-based fault localization techniques and applies them in tandem with an enhanced change impact analysis that uses Extended Call Graphs to identify failure-inducing edits more precisely. We evaluate FAULTTRACER using 23 versions of 4 real-world Java programs from the Software Infrastructure Repository. The experimental results show that FAULTTRACER outperforms Chianti in selecting affected tests (slightly better, but handles safety problems of Chianti) as well as in determining affecting changes (with an improvement of approximately 20%). By ranking the affecting changes using spectrum-based test behavior profile, for 14 out of 22 studied failures, FAULTTRACER places a real regression fault within top 3 atomic changes, significantly reducing developers' effort in inspecting potential failure-inducing edits.**

I. INTRODUCTION

While fixing bugs or adding features, developers must keep such modifications from introducing new faults. Regression test suites are often used to validate such program edits. However, regression suites can be large and time consuming to run, and localizing and fixing faults can be tedious and error-prone.

Change impact analysis is a well-known methodology for selecting *affected tests*, i.e., a subset of the regression suite that may be influenced by the edits, and determining *affecting changes*, which are a subset of edits that may cause test failures. While change impact analysis techniques, such as Chianti [19], can effectively reduce potential failure-inducing edits as affecting changes, the number of affecting changes for each failed test may still be too large for manual inspection. For example, even for a medium size Java program such as ant 6.0 and ant 7.0, the number of atomic changes for each failed test can be substantial, ranging from 22 to 182 atomic changes, with an average of approximately 100 changes.

Recent years have seen much progress in *automated debugging* [24] and *fault localization* [3], [4], [9], [11], [23]. A state-of-the-art approach to identify faulty statements in source code is based on *test spectra*—code coverage for each passed or failed test—and produces a ranking of potential faulty statements based on suspiciousness scores [3], [4], [11], [23]. For large programs, however, the number of suspicious statements could be too large for manual inspection. For example, according to a recent study [21], Tarantula localizes faults to 14.77% of code on average for *xml-security1.0*, in other words, 3192 statements. Moreover, in the context of evolution, a vast majority of the identified statements may not correspond to program edits, as these techniques do not consider information about program changes.

While change impact analysis and fault localization techniques are closely related, the strengths and limitations of these techniques are complementary. Change impact analysis focuses on potential failure-inducing edits (i.e., affecting changes), yet it does not leverage spectrum information to rank those edits. On the other hand, spectrum-based fault localization ranks likely faults within a single program version as opposed to program edits. Our insight is that we can combine the strengths of change impact analysis and fault localization by applying them in tandem to localize failure-inducing program edits more precisely.

This paper presents the FAULTTRACER framework, which embodies our insight. FAULTTRACER adapts four state-of-the-art spectrum-based fault localization techniques [3], [4], [11], [23] to Chianti-style change impact analysis to achieve more precise fault localization for evolving software systems. Instead of computing test spectra using all statements in the new program version, FAULTTRACER's spectrum computation focuses on potential failure-inducing program edits (i.e., affecting changes) between the old and new version. To capture more accurate spectrum information for program edits, FAULTTRACER introduces *extended call graphs* (ECGs), which extend traditional method call graphs with field access information. The use of ECGs enables more accurate selection of affected tests and determination of affecting changes, thus enabling more precise spectrum-based fault localization.

This paper makes the following contributions:

- **Fault-localization for evolving systems.** We define spectrum-based fault localization in the context of program edits. Moreover, we adapt four existing fault localization techniques and show how to apply them to rank potential failure-inducing program edits.

- **Extended call graphs for change impact analysis.** We define an enhanced program representation to capture program edits more precisely. We also present methods for improving the precision of affected test selection and affecting change determination.
- **Experimental evaluation.** We implemented FAULTTRACER as an Eclipse IDE plug-in and performed an experimental evaluation using 23 versions of 4 real-world Java programs from the Software Infrastructure Repository [7]. The results show that for change impact analysis, FAULTTRACER outperforms Chianti in selecting affected tests (slightly better, but handles safety problems of Chianti) as well as in determining affecting changes (with an improvement of approximately 20%). Furthermore, by ranking potential failure-inducing edits based on test spectra, FAULTTRACER localizes real regression faults with a surprising accuracy: for 14 of the 22 studied failures, FAULTTRACER localizes real regression faults within top 3 affecting changes.

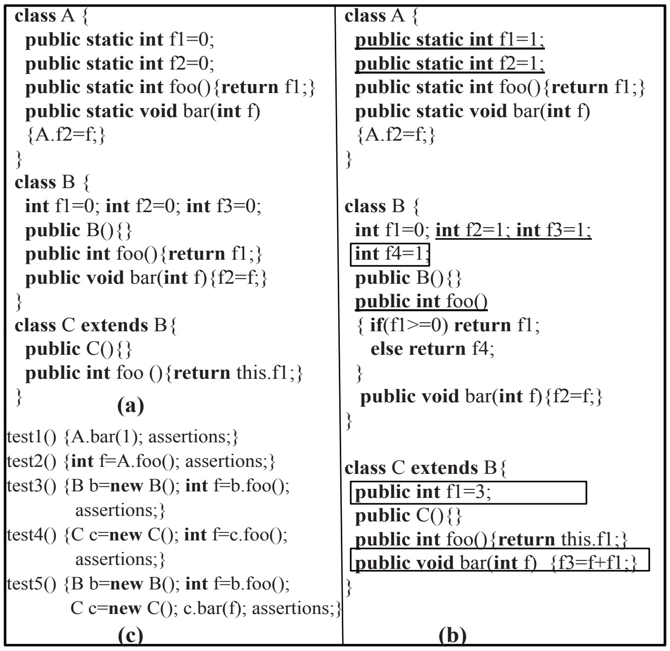
II. RELATED WORK

A. *Change Impact Analysis*

Change Impact Analysis [5], [12], [14], [15], [17], [19], [20] aims to determine the impact of source code changes on other parts of a program. For example, Chianti [17], [19], [20] selects a subset of regression tests whose behavior might have changed and then identifies affecting program edits responsible for test failures. For each failed test, the number of affecting changes produced by Chianti could be still large. Chesley et al. [6] propose Crisp, which uses four pre-defined rules to group relevant edits based on compilation dependences such as a new method `A.m()` that accesses a newly declared field `A.f`. This technique still requires manual debugging to localize failure-inducing changes. Stoerzer et al. [22] use change classification techniques to find failure-inducing changes. However, this technique does not rank changes, and the classified changes might still be large in number. Ren et al. [18] propose a heuristic ranking algorithm for method-level edits based on their numbers of ancestors, descendants, callers and callees on call graphs of tests. Different from this heuristic algorithm, FAULTTRACER is based on spectrum information of program edits. Furthermore, while Ren et al.'s ranking algorithm is confined to method-level edits, FAULTTRACER's ranking algorithm uses test profile at the level of *extended call graphs* to consider both method calls and field accesses and can rank all types of program edits including addition, modification, and deletion of fields.

B. *Fault Localization*

Fault localization techniques [3], [4], [8], [9], [13], [23] such as Tarantula [11] statistically analyze the execution traces of passed and failed test cases, and rank potential faulty statements based on suspiciousness scores, which are derived from test spectra. However, such spectrum-based fault localization is not applicable to large evolving software systems, as they compute spectra on all statements in each program version and do not leverage information about program edits between the old and new versions.



```
class A {
  public static int f1=0;
  public static int f2=0;
  public static int foo(){return f1;}
  public static void bar(int f)
  {A.f2=f;}
}
class B {
  int f1=0; int f2=0; int f3=0;
  public B(){}
  public int foo(){return f1;}
  public void bar(int f){f2=f;}
}
class C extends B{
  public C(){}
  public int foo (){return this.f1;}
}
```

(a)

```
test1() {A.bar(1); assertions;}
test2() {int f=A.foo(); assertions;}
test3() {B b=new B(); int f=b.foo();
         assertions;}
test4() {C c=new C(); int f=c.foo();
         assertions;}
test5() {B b=new B(); int f=b.foo();
         C c=new C(); c.bar(f); assertions;}
```

(c)

```
class A {
  public static int f1=1;
  public static int f2=1;
  public static int foo(){return f1;}
  public static void bar(int f)
  {A.f2=f;}
}

class B {
  int f1=0; int f2=1; int f3=1;
  int f4=1;
  public B(){}
  public int foo()
  { if(f1>=0) return f1;
    else return f4;
  }
   public void bar(int f){f2=f;}
}

class C extends B{
  public int f1=3;
  public C(){}
  public int foo(){return this.f1;}
  public void bar(int f)  {f3=f+f1;}
}
```

(b)

Fig. 1. (a) An example program before edits. (b) The example program after edits. Note that code in box are newly added, and code with underlines are changed. (c) Tests for the program.

Delta Debugging [24] applies a subset of all changes to construct intermediate versions to find a minimum set of changes that lead to a test failure. However, Delta Debugging considers all changes between the old and new program version as a candidate set without considering method call and field access level dependences among those changes. Furthermore, Delta Debugging determines only a set of edits that may lead to a test failure but does not rank these edits according to their test spectra, leaving it to a programmer to sort out a real culprit of a regression test failure among a likely large set of potential failure-inducing changes.

Our approach, FAULTTRACER, combines the strengths of Chianti-style change impact analysis and Tarantula-style fault localization. To present a ranked list of potential failure-inducing edits, we apply a set of spectrum-based ranking techniques to the affecting changes determined by Chianti-style change impact analysis. This hybrid approach also requires us to develop a new enhanced call graph representation to measure test spectrum information directly for field-level edits and to improve upon existing Chianti algorithm. These extensions are described in the next section.

III. FAULTTRACER

This section first provides an overview of FAULTTRACER. Given the old and new program versions, P and P', FAULTTRACER computes all atomic changes and their dependences, denoted as Δ. Second, it selects a subset of regression tests whose behaviors could be affected by the atomic changes Δ by analyzing extended call graphs (ECGs) of tests. Third, when

TABLE I
CHANGE TYPES SUPPORTED BY FAULTTRACER.

No.	Change Types	Description
1	CM	change method
2	AM	add method
3	DM	delete method
4	AF	add field
5	DF	delete field
6	CFI	change instance field initializer
7	CSFI	change static filed initializer
8	$\texttt{LC}_m$	look up change due to method changes
9	$\texttt{LC}_f$	look up change due to field changes

a test t fails, FAULTTRACER determines the subset of changes, i.e., δ_t, that may have influenced the failed test based on its ECG. Fourth, it uses spectrum information extracted from the ECGs of selected tests, T', to compute suspiciousness scores for each atomic change in Δ. It then ranks the changes in δ_t according to their suspiciousness scores for each failed test t. Using FAULTTRACER, developers may inspect a ranked list of atomic changes, i.e., δ'_t, for each failed test t, enabling more efficient inspection of failure-inducing edits. In the following subsections, we use the example program in Figure 1 to illustrate the approach of FAULTTRACER.

A. ECG-Based Change Impact Analysis

Atomic Change Types. Similar to Chianti, FAULTTRACER extracts atomic changes such as method additions and deletions by comparing the abstract syntax tree of two program versions. While this step is almost identical to Chianti's atomic change extraction, as shown in Table I, FAULTTRACER uses nine atomic change types, where the first eight types are inherited from Chianti and the ninth type is added. FAULTTRACER introduces look-up changes ($\texttt{LC}_f$) that arise due to field hiding hierarchy changes. Consider method `C.foo` in Figure 1 that illustrates the importance of field look-up changes. Although there is no change to the method and the field accessed by the method, the hiding hierarchy of `C.f1` has been changed, making `C.foo` access a new field with a different value. Also note that FAULTTRACER splits all higher-level changes (e.g., class and package changes) into these nine basic changes.

Atomic Change Dependences. The dependences between atomic changes are useful for understanding and debugging programs. While Chianti considers dependences among atomic changes, it does not model dependences between changes precisely when involving method overriding or field hiding, which leads to false positive dependences. FAULTTRACER makes refinements in these situations to make derived dependences among atomic changes more precise. The details are summarized in Table II, where the shadowed parts denote the constraints introduced by FAULTTRACER to define more precise dependence inference. `Callee(m)` denotes a set of methods statically called by `m`, and `Access(m)` denotes a set of fields statically accessed by m. Row 1 denotes that, for every $\texttt{LC}_m$ change, there is method additions (AM) or method deletions (DM) which caused it, and the look-up change $\texttt{LC}_m$ is dependent on those changes, represented by the $\preceq$. Similarly, Row 2 denotes that every $\texttt{LC}_f$ change also depends on corresponding AF or DF changes. According to Row 3, for every AM change, if a method called by the added method is new and all methods overridden by it are also new, the caller should be dependent on the added callee. If a field accessed by an added method is new and all fields hidden by it are also new, the added method should be dependent on the added field. Compared to Chianti, FAULTTRACER adds the shadowed constraints that all overridden methods and all hidden fields should also be newly added. Otherwise, any overridden method (or hidden field) can make the new version compile without the overriding method (or the hiding field). Therefore, FAULTTRACER avoids false positive dependences among atomic changes.

In Row 4, for every method deletion (DM), if a method called within the deleted method is newly deleted and all the method overridden by it are also newly deleted, the callee should be dependent on the caller. Otherwise the caller would be calling methods without declarations. Similarly, if a field accessed by the deleted method DM is a deleted field (DF) and if all fields hidden by it also belong to DF, it should also be dependent on the caller DM. In Row 5, we use one-to-one mappings to denote changed methods (CM): <m_1, m_2>, where m_1 denotes a method before edits, while m_2 denotes that method after edits. For any deleted fields or deleted methods that are accessed or called by the method before edits (i.e., m_1), if their overriding methods or hiding fields are all newly deleted, they should be dependent on <m_1, m_2>; for any added fields or added methods that are accessed or called by the method after edits (i.e., m_2), if their overriding methods or hiding fields are all newly added, <m_1, m_2> should be dependent on them. These refined rules allow FAULTTRACER to model dependences among atomic changes more accurately.

Extended Call Graph. Chianti records dynamic call graphs of tests to select affected tests and determine affecting changes. In contrast, FAULTTRACER uses *extended call graph* (ECG) representation which enhances a dynamic call graph to additionally include field access information.

Definition 1: A graph G is an ECG of a root method, if and only if G= <V, E>, $V = M \cup F$, $E = \bigcup_{m \in M}(\{m \rightarrow m' | m' \in \texttt{Callee(m)} \cap \texttt{IsStatic(m')}\} \cup \{m \rightarrow m'' | m'' \in \texttt{Callee}(m) \cap \texttt{IsVirtual}(m'')\} \cup \{m \dashrightarrow f | f \in \texttt{Access}(m)\})$.

In the definition, M and F denote a subset of methods and fields reachable from the root method in the program under test. Also, we use "$\rightarrow$" to denote call edges between methods, and "$\dashrightarrow$" to represent access edges from methods to fields. To detect method look-up changes, $\texttt{LC}_m$, for each virtual call edge, we also retain information about the receiver object's runtime type and the static target method in the form of <T, $C.m$>. In addition, we augment each field access edge with (<Op, $C.f$>), in which $C.f$ denotes the static target field used to detect $\texttt{LC}_f$ changes, and Op represents the type of an operation performed:

$$Op := FW | FR | SFW | SFR$$

where FW means an instance field write, FR means an

TABLE II
RULES FOR DEFINING DEPENDENCES AMONG ATOMIC CHANGES.

$\forall lc \in LC_m : \forall c \in AM \cup DM, c \text{ caused } lc \Rightarrow (c \preceq lc) \in Dependences$
$\forall lc \in LC_f : \forall f \in AF \cup DF, f \text{ caused } lc \Rightarrow (f \preceq lc) \in Dependences$
$\forall m \in AM : \forall m' \in Callee(m), (m' \in AM) \wedge (Overriding(m') \subseteq AM) \Rightarrow (m' \preceq m) \in Dependences;$ $\forall f \in Access(m), (f \in AF) \wedge (Hiding(f) \subseteq AF) \Rightarrow (f \preceq m) \in Dependences$
$\forall m \in DM : \forall m' \in Callee(m), (m' \in DM) \wedge (Overriding(m') \subseteq DM) \Rightarrow (m \preceq m') \in Dependences;$ $\forall f \in Access(m), (f \in DF) \wedge (Hiding(f) \subseteq DF) \Rightarrow (m \preceq f) \in Dependences$
$\forall < m_1, m_2 > \in CM : \forall m'_1 \in Callee(m_1), (m'_1 \in DM) \wedge (Overriding(m'_1) \subseteq DM) \Rightarrow (< m_1, m_2 > \preceq m'_1) \in Dependences;$ $\forall f \in Access(m_1), (f \in DF) \wedge (Hiding(f) \subseteq DF) \Rightarrow (< m_1, m_2 > \preceq f) \in Dependences;$ $\forall m'_2 \in Callee(m_2), (m'_2 \in AM) \wedge (Overriding(m'_2) \subseteq AM) \Rightarrow (m'_2 \preceq < m_1, m_2 >) \in Dependences;$ $\forall f \in Access(m_2), (f \in AF) \wedge (Hiding(f) \subseteq AF) \Rightarrow (f \preceq < m_1, m_2 >) \in Dependences$

TABLE III
AFFECTED TESTS AND AFFECTING CHANGES INFERENCE.

$$\begin{aligned}
&AffectedTests(T, \Delta) = \\
&\quad \{t | t \in T, (V(P, t) \setminus (FW \cup SFW)) \cap (\Delta \setminus (LC_m \cup LC_f)) \neq \emptyset\} \\
&\quad \cup \{t | t \in T, n, A.m \in V(P, t), B <^* X, LC_m(< B, X.m >) \in LC_m, \\
&\qquad n \rightarrow_{<B,X.m>} A.m \in E(P, t)\} \\
&\quad \cup \{t | t \in T, n, A.f \in V(P, t), X <^* A, LC_f(X.f) \in LC_f, \\
&\qquad n \dashrightarrow_{<*,X.f>} A.f \in E(P, t)\}
\end{aligned}$$

$$\begin{aligned}
&AffectingChanges(t, \Delta) = \\
&\quad \{c | c \in \Delta \setminus (LC_m \cup LC_f), (V(P', t) \setminus (FW \cup SFW)) \cap c \neq \emptyset\} \\
&\quad \cup \{c | LC_m(< B, X.m >) \in LC_m, n, A.m \in V(P', t), \\
&\qquad n \rightarrow_{<B,X.m>} A.m \in E(P', t), c \preceq LC_m(< B, X.m >)\} \\
&\quad \cup \{c | LC_f(X.f) \in LC_f, n, A.f \in V(P', t), \\
&\qquad n \dashrightarrow_{<*,X.f>} A.f \in E(P', t), c \preceq LC_f(X.f)\}
\end{aligned}$$

instance field read, SFW means a static field write, and SFR means a static field read. Field accesses are determined statically, so we do not need to affiliate runtime type information with a field access edge. Instead, we associate Op to indicate types of access operations on fields and apply different matching strategies on field-write and field-read nodes (illustrated in the following two paragraphs). Figure 2 shows ECGs for the tests from the code example. In the figure, methods are represented by rounded rectangles; field are represented by rounded rectangles in dashed line; method call edges are indicated by solid lines; field accesses are indicated by dashed lines; and associated labels are presented in rectangles.

Selecting Affected Tests. Given a program P, its modified version P', and a test suite T, FAULTTRACER uses the first rule shown in Table III to determine a subset of tests whose behaviors could change and thus must be rerun. Δ denotes all atomic changes between P and P', $<^*$ denotes transitive inheritance relations between two classes, $V(P, t)$ and $E(P, t)$ denote the set of vertices and the set of edges in t's ECG when run on P. The rule selects three categories of tests. First, it selects tests that have common method invocation nodes with method changes, or common field-read nodes with field changes. Here, we do not consider the field-write nodes, because field changes will be reset by field write nodes, thus cannot influence corresponding tests. Second, it selects tests that have method invocation edges corresponding to method look-up changes. Third, it also selects tests that have field-access edges corresponding to field look-up changes.

The use of ECG has three benefits over the use of regular method call graphs: first, the use of ECG enables FAULTTRACER not to explicitly transform field changes (e.g., `CFI` and `CSFI`) to `CM` changes as Chianti does, because the field operations can be directly mapped to vertices representing fields in ECG (e.g., `CFI` changes will map to `FW` or `FR` nodes in ECG). Second, FAULTTRACER avoids selecting unnecessary tests due to `CSFI` changes. Third, ECG helps to avoid false negative selections made by Chianti due to `CSFI` changes and absence of $\texttt{LC}_f$ changes.

Consider the example shown in Figure 1. The following table shows test selection results for Chianti, FAULTTRACER, and an optimal solution that selects exactly real affected tests:

Tech	test1	test2	test3	test4	test5
Chianti	✓		✓		✓
FAULTTRACER		✓	✓	✓	✓
Optimal		✓	✓	✓	✓

Chianti produces one false positive and two false negatives, while FAULTTRACER does not produce any false negative or false positive selection. For `test1` and `test2`, Chianti detects changes `CSFI(A.f1)` and `CSFI(A.f2)` and maps them to a change on class `A`'s static initializer: `CM(A.Clinit)`. Then, Chianti detects that `test1` is influenced because `test1` executes `CM(A.Clinit)` (false positive), and misses `test2` that must have been included (false negative)—the static class initializer is only executed once for all the tests, and does not appear in `test2`'s call graph. On the other hand, in Figure 2 (a), by with the use of ECG, FAULTTRACER successfully detects `test2` and does not select `test1`, since change `CSFI(A.f2)` appears as a `SFW` node in `test1`'s ECG (field-write node resets field value and mutes influence of field change). For `test4`, Chianti commits a false negative on `test4` because it cannot detect $\texttt{LC}_f$ changes.

Selecting Affecting Changes. Given a failed test t, FAULTTRACER uses the second rule of Table III to determine affecting changes. $V(P', t)$ and $E(P', t)$ denote the set of vertices and edges in the ECG of t when running on the new version P'. For each test, FAULTTRACER select all non-look-up changes appearing in its ECG as affecting changes. In addition, FAULTTRACER also selects the causing changes of look-up changes as affecting changes (excluding look-up changes since look-up changes are not directly committed by developers). The use of ECG here allows FAULTTRACER to

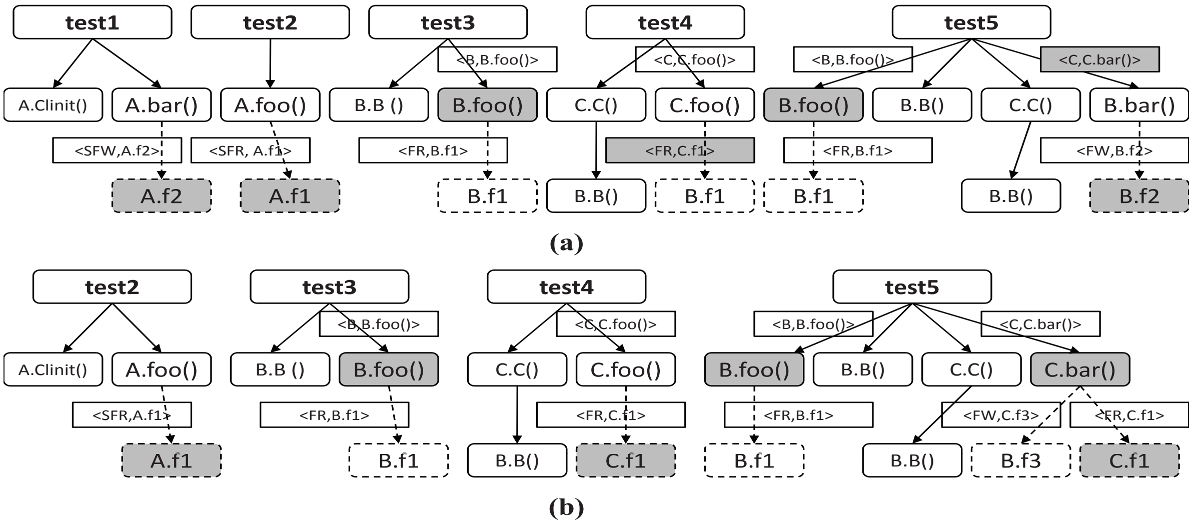


Fig. 2. (a) ECG graphs of the tests before edits, where gray nodes denote the atomic changes mapping to the ECGs. (b) ECG graphs of the affected tests after edits, where gray nodes denote affecting changes.

TABLE IV
SPECTRUM AND SUSPICIOUSNESS SCORES.

	Affected Tests				Suspiciousness Score				
Edits	test2	test3	test4	test5	Tarantula	SBI	Jaccard	Ochiai	BreakTie
`CSFI(A.f1)`	✓				0.00	0.00	0.00	0.00	-
`CM(B.foo)`		✓		✓	0.75	0.50	0.50	0.71	1
`AF(C.f1)`			✓	✓	0.75	0.50	0.50	0.71	0
`AM(C.bar)`				✓	1.00	1.00	1.00	1.00	-
Out	P	P	P	F					

find affecting changes by directly mapping atomic changes to corresponding ECG nodes. In contrast, Chianti transforms field changes to method changes when determining affecting changes, and thus it reports not only the vertices in the call graph but also all the changes that the vertices transitively depend on. In fact, many of them might not be executed in the real execution. For `test3` of the example, first, Chianti finds changes `CFI(B.f2)`, `CFI(B.f3)`, `AF(B.f4)`, and `CM(B.foo)`. Second, Chianti detects `CM(B.foo)` is dependent on `AF(B.f4)`. By transforming `CFI(B.f2)` and `CFI(B.f3)` into `CM(B.B)`, Chianti also detects that `CM(B.B)` is dependent on `CFI(B.f2)` and `CFI(B.f3)`. Third, by matching `CM` or `AM` changes on call graphs, Chianti selects `CM(B.B)` and `CM(B.foo)` as affecting changes. In addition, Chianti also selects all the changes that these two changes transitively depend on, i.e., `CFI(B.f2)`, `CFI(B.f3)`, `AF(B.f4)`. As a result, Chianti reports five atomic changes. On the other hand, FAULTTRACER selects only `CM(B.foo)` as affecting changes, since all changed or added fields are not even accessed by `test3`, as depicted in Figure 2 (b). This improvement not only enables FAULTTRACER to select affecting changes more precisely, but also allows collecting accurate spectrum information to rank affecting changes.

B. Spectrum-based Ranking of Program Edits

For each test failure, FAULTTRACER identifies a subset of changes that might induce test failures as described in Section III-A. Since the likelihood of each affecting change to induce a test failure is not equal, FAULTTRACER further ranks the set of affecting changes to assist developers in inspecting the results of FAULTTRACER. Inspired by fault localization techniques based on test spectra, FAULTTRACER adapts four different spectrum-based fault localization techniques to calculate the suspiciousness scores for affecting changes. These techniques are selected because they were widely used in comparative studies of fault localization techniques [3], [10], [23]. Then, given a test failure and a corresponding set of affecting changes related to the test failure, FAULTTRACER uses the suspiciousness scores to rank the affecting changes for the test failure. We next give formal definitions on how to calculate the suspiciousness score for each atomic change a. We illustrate our ranking techniques using the example program. Assume the real culprit that causes `test5` to fail is the `AM` change on `C.bar`, e.g., the code should have been `f3=f-f1`.

Given two versions of a tested program, P and P', and a regression test suite T, FAULTTRACER extracts an atomic change set Δ. Based on rules in Table III, FAULTTRACER selects affected tests, $T' = AffectedTests(T, \Delta)$, and for each affected test, $t' \in T'$, it identifies a set of affecting changes, $\delta_{t'} = AffectingChanges(t', \Delta)$. Then for each atomic change $a \in \Delta$, we denote a set of affected tests that exercise a as $C(a, T') = \{t' | t' \in T', a \in \delta_{t'}\}$. We denote the set of failed tests in T' as T'_f, and denote the set of passed tests in T' as T'_p. To illustrate, Table IV shows an application of FAULTTRACER on localizing failure-inducing edits for our example program in Figure 1. FAULTTRACER computes T' = $\{t2$, $t3$, $t4$, and $t5\}$. Check marks (✓) show the correspondence between affected tests and affecting changes. Then, $C(a, T')$ is represented by the row corresponding to each edit a in the table. Test outcomes are represented with **P** (i.e., passed) and **F** (i.e., failed).

Fault Localization Techniques. We adapt the following four state-of-the-art spectrum-based fault localization techniques to rank affecting changes:

Tarantula. Jones et al. [11] first propose Tarantula, which assigns higher suspiciousness scores to statements primarily executed by failed tests than statements primarily executed by passed tests. Similarly, we define a suspiciousness score for an atomic change a as:

$$S_t(a) = \frac{\overbrace{|C(a,T') \cap T'_f|/|T'_f|}^{\%failed(a)}}{\underbrace{|C(a,T') \cap T'_p|/|T'_p|}_{\%passed(a)} + \underbrace{|C(a,T') \cap T'_f|/|T'_f|}_{\%failed(a)}}$$

in which $\%failed(a)$ denotes the ratio of failed affected tests executing edit a to all failed affected tests, while $\%passed(a)$ denotes the ratio of passed affected tests executing edit a to all passed affected tests.

Statistical Bug Isolation. Liblit et al. [13] first propose Statistical Bug Isolation (SBI) to compute a suspiciousness score of a predicate. Yu et al. [23] then adapt SBI to rank potential faulty statements. Similarly, we define a suspiciousness score for an atomic change a as:

$$S_s(a) = \frac{\overbrace{|C(a,T') \cap T'_f|}^{failed(a)}}{\underbrace{|C(a,T') \cap T'_p|}_{passed(a)} + \underbrace{|C(a,T') \cap T'_f|}_{failed(a)}}$$

in which $failed(a)$ denotes the number of failed affected tests executing edit a and $passed(a)$ denotes the number of passed affected tests executing edit a.

Jaccard. Abreu et al. [3] first use the Jaccard similarity coefficient, which is used for measuring the statistical similarity and diversity between sample sets, in ranking faulty statements. Similarly, we define a suspiciousness score for an atomic change a as:

$$S_j(a) = \frac{\overbrace{|C(a,T') \cap T'_f|}^{failed(a)}}{\underbrace{|T'_f|}_{all\ failed} + \underbrace{|C(a,T') \cap T'_p|}_{passed(a)}}$$

in which $all\ failed$ denotes the number of all failed affected tests.

Ochiai. Yu et al. and Abreu et al. [3], [23] use Ochiai coefficient, which originated from the molecular biology domain, as a metric for ranking faulty statements. Similarly, we define a suspiciousness score for an atomic change a as:

$$S_o(a) = \frac{\overbrace{|C(a,T') \cap T'_f|}^{failed(a)}}{\sqrt{\underbrace{|T'_f|}_{all\ failed} * (\underbrace{|C(a,T') \cap T'_p|}_{passed(a)} + \underbrace{|C(a,T') \cap T'_f|}_{failed(a)})}}$$

Breaking Ties. When using the above spectrum-based ranking techniques, there could be more than one atomic changes with the same suspiciousness score. FAULTTRACER computes tie-breaking scores based on a hypothesis that changes that transitively depend on more changes are more likely to contribute to regression faults. Formally, we define a tie-breaking score as:

$$B(a) = |\{a' | a' \preceq^* a \wedge a' \neq a\}|$$

where the value $B(a)$ is the number of atomic changes that transitively depend on a.

Consider our running example again. The suspiciousness scores calculated for the affecting changes are shown in the right side of Table IV. Affecting changes for `test5` are δ_{test5}={`CM(B.foo)`, `AM(C.bar)`, `AF(C.f1)`}. When ranking them according to their suspiciousness scores, we find the score for `CM(B.foo)` is the same as the one for `AF(C.f1)` for all spectrum-based strategies. Thus, FAULTTRACER further uses tie-breaking scores to distinguish them (see the last column in Table IV). Finally, FAULTTRACER outputs the following result: `AM(C.bar)` (rank 1), `CM(B.foo)` (rank 2), and `AF(C.f1)` (rank 3), placing the real fault-inducing change, `AM(C.bar)` as the number 1 in the list.

C. Implementation

This section describes the FAULTTRACER Eclipse plug-in.

ECG Construction. FAULTTRACER constructs dynamic enhanced call graphs through code instrumentation. It uses the *ASM byte-code manipulation and analysis framework* [1]. We extend *visitor* classes of the ASM framework and override *visit* methods to trace method invocation relations, field access relations, and associated attributes (e.g., receiver object types, static target methods for virtual method invocations, specific types of field accesses, and so on).

Change Extraction. FAULTTRACER extracts atomic changes between two program using the abstract syntax tree analysis provided by the *Eclipse JDT toolkit* [2]. It traverses the abstract syntax tree of programs to compare fields and methods by their fully qualified names and find atomic changes. For each pair of compared atomic elements, we filter out all comments and white-spaces before comparison. We also find call and access dependences between atomic changes by tracing the definition and reference of each atomic element.

IV. EXPERIMENTAL STUDY

A. Research Questions

In our experimental study, we investigate the following research questions.

- **RQ1:** How effective is FAULTTRACER in ranking potential failure-inducing edits (i.e., affecting changes)?
- **RQ2:** How does FAULTTRACER compare to Chianti in terms of selecting affected tests and determining affecting changes?

To answer *RQ1*, we evaluate the performance of FAULTTRACER in ranking all affecting changes by measuring how many atomic changes a developer must inspect before finding a regression fault. We also compared FAULTTRACER with Ren

TABLE V
STATISTICS FOR SUBJECTS

S.	#Class	#SM	#IM	#SF	#IF	#Test	LoC
j0	17	0	267	31	58	95	1,831
j1	19	1	284	33	59	126	1,897
j2	21	2	302	35	59	128	2,031
j3	50	6	748	84	162	209	5,361
x0	193	292	1256	343	140	106	17,435
x1	179	315	1312	370	147	92	18,323
x2	180	313	1316	381	144	94	18,985
x3	145	303	1095	406	129	84	16,878
m0	302	166	2595	373	830	70	31,005
m1	334	174	2926	502	894	78	33,670
m2	319	171	2675	564	893	80	33,097
m3	373	202	3453	653	938	78	37,271
m4	380	235	3540	703	935	78	38,357
m5	389	239	3617	717	970	97	41,052
a0	172	55	1581	161	748	112	17,201
a1	228	84	2427	275	1145	137	25,846
a2	342	146	3690	440	1714	219	39,733
a3	342	149	3696	440	1718	219	38,810
a4	532	254	5430	671	2520	521	61,877
a5	536	256	5546	679	2546	557	63,510
a6	536	256	5808	681	2550	559	63,578
a7	649	334	7186	850	3212	877	80,381
a8	650	334	7190	850	3212	878	80,444

et al.'s ranking algorithm [18], which handles only method-level edits (e.g., CM and AM). In this comparison, we degrade FAULTTRACER to rank only method-level edits and compare their effectiveness for a fair comparison.

To answer *RQ2*, we compare FAULTTRACER with Chianti. As Chianti is not publicly available, to enable a fair comparison, we re-implemented Chianti based on their 2004 OOPSLA paper [19]. Note that FAULTTRACER and Chianti have different attitudes towards look-up changes in determining affecting changes, thus we do not take into account look-up changes in determining affecting changes to enable fair comparison.

B. Subjects

We use four Java programs from Software Infrastructure Repository (SIR) [7] as our experimental subjects. *jtopas* is a java library used for parsing text data. *xml-security* is a Java component library which implements XML signature and encryption standards. *jmeter* is implemented for load testing and performance measurement. *ant* is a Java-based build tool, similar to the Unix tool, make. All these programs are written in the Java language and have successive versions, which have been developed during software evolution. Our experimental study uses all versions of these programs in SIR, and the detailed statistics for the subjects are shown in Table V. In the table, Column 1 lists the names of the subjects, e.g., we denote 4 versions of *jtopas* as j_0, j_1, j_2, and j_3, respectively. Columns 2 to 8 show the number of classes, static methods, instance methods, static fields, instance fields, test cases, and lines of code for each subject.

C. Experimental Setup

We use successive versions of each program as version pair to perform the experimental study. For example, we have 4 successive versions of *jtopas*, i.e., we can construct 3 consecutive version pairs. For each program s, we use the tuple <s_i, s_{i+1}> to denote the version pair between s_i and s_{i+1}. For each version pair, we first run tests on the old version, and use Chianti and FAULTTRACER to select affected tests. Then, we run the selected tests on the new version and determine affecting changes. FAULTTRACER then ranks program edits with test spectra. We first show the statistics for all atomic changes extracted by FAULTTRACER on all the 19 version pairs in Figure 3. Note that the *Horizontal-axis* denotes all the 19 version pairs, the *Z-axis* denotes the change types for each version pair, and the *Vertical-axis* reports the number of changes for each change type and each version pair. The number of all atomic changes varies significantly across the studied version pairs, ranging from 35 (<a_5, a_6>) to 5752 (<a_3, a_4>).

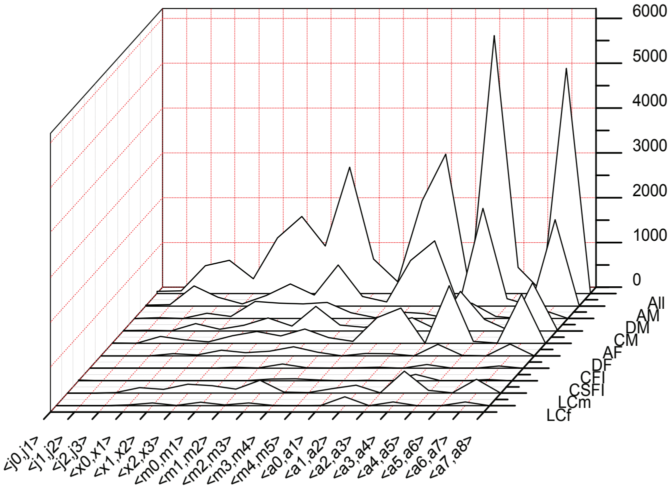


Fig. 3. Statistics for all atomic changes over 19 version pairs

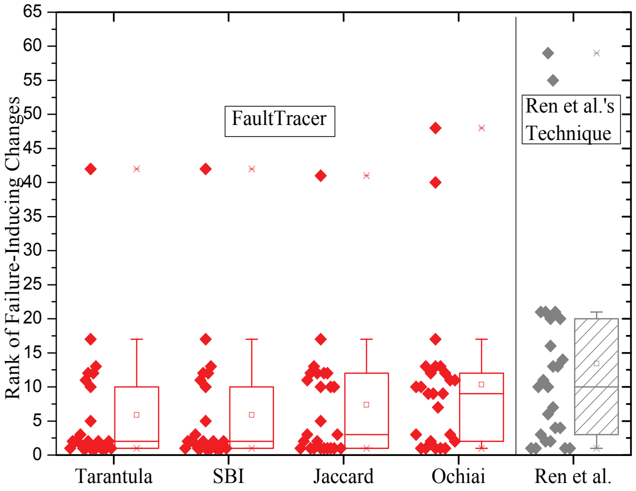


Fig. 4. Comparison of FAULTTRACER with Ren et al. in ranking method-level failure inducing edits.

D. RQ1: Ranking of Program Edits.

For each studied version pair, we treat the tests that succeed in the old version but fail in the new version as regression test failures. For each test failure, we treat the minimum set of program edits that cause the corresponding failure as failure-

TABLE VI
FAULT LOCALIZATION RESULTS OF FAULTTRACER

				Tarantula		SBI		Jaccard		Ochiai	
Pair	Failed Test	Δ	δ_t	Susp	Rank	Susp	Rank	Susp	Rank	Susp	Rank
<x_1, x_2>	XMLSignatureInputTest.testSetNodeSetGetOctetStream1	329	10	0.25	2	0.03	2	0.03	2	0.08	2
<a_0, a_1>	EchoTest.test1	2071	13	1.00	1	1.00	1	0.60	1	0.77	1
	EchoTest.test2	2071	14	1.00	1	1.00	1	0.60	1	0.77	1
	EchoTest.test3	2071	15	1.00	1	1.00	1	0.60	1	0.77	1
	MkdirTest.test2	2071	11	0.92	2	0.40	2	0.36	2	0.56	2
	CommandlineJavaTest.testGetCommandline	2071	5	1.00	1	1.00	1	0.20	1	0.44	1
<a_3, a_4>	GzipTest.test5	5752	107	0.96	3	0.33	3	0.20	2	0.33	67
	GUnzipTest.test3	5752	105	0.91	8	0.15	8	0.15	8	0.38	3
	TaskdefTest.test5	5752	114	0.89	52	0.12	52	0.12	51	0.34	48
<a_4, a_5>	TypedefTest.testLocal	586	13	0.84	10	0.02	10	0.02	10	0.15	7
	ZipTest.test4	586	30	0.94	3	0.07	3	0.06	3	0.18	3
<a_6, a_7>	XMLCatalogTest.testNestedCatalog	5019	84	0.97	20	0.45	20	0.27	20	0.43	21
	XMLCatalogTest.testClasspath	5019	71	0.96	29	0.38	29	0.25	28	0.40	28
	XMLCatalogTest.testSimpleEntry	5019	77	0.97	19	0.45	19	0.27	20	0.43	20
	EchoPropertiesTest.testEchoToBadFileNoFail	5019	181	0.94	1	0.30	1	0.15	1	0.27	10
	XMLCatalogTest.testEmptyCatalog	5019	62	0.98	1	0.60	1	0.21	14	0.38	14
		5019	62	0.98	2	0.60	2	0.21	15	0.38	15
	XMLCatalogTest.testEmptyElementIfIsReference	5019	22	0.96	2	0.40	2	0.13	3	0.25	11
	XMLCatalogTest.testNonExistentEntry	5019	66	0.98	1	0.60	1	0.21	18	0.38	18
		5019	66	0.98	2	0.60	2	0.21	19	0.38	19
	XMLCatalogTest.testResolverBase	5019	79	0.97	21	0.45	21	0.27	18	0.43	19
	EchoPropertiesTest.testEchoToBadFileFail	5019	182	0.94	1	0.30	1	0.15	1	0.27	10
	EchoPropertiesTest.testEchoToBadFile	5019	182	0.94	1	0.30	1	0.15	1	0.27	10
	XMLCatalogTest.testEntryReference	5019	81	0.97	20	0.45	20	0.27	20	0.43	21
Average	-	3932	68.83	-	8.50	-	8.50	-	10.83	-	14.66

inducing edits. For all the studied version pairs, 27 regression test failures are caused by 17 failure-inducing edits, which were identified manually by the first author. All of the faults were method-level changes (i.e., `AM` and `CM` changes). We then evaluate the ranking techniques of FAULTTRACER on the 22 failed tests with more than five affecting changes, since it is trivial to localize faults for failed tests with less than 5 affecting changes.

Ranking all affecting changes. In Table VI, Columns 1 and 2 present the version pair and the test where each regression test failure occurs. Columns 3 and 4 present the number of all atomic changes (i.e., Δ) and the number of affecting changes (i.e., δ_t) for each failed test. Columns 5 to 12 present the fault localization results by the four fault localization techniques of FAULTTRACER, where Column **Susp** presents the suspiciousness score for the each fault (i.e., failure-inducing change) and Column **Rank** presents a respective ranking of each fault among affecting changes of failed tests caused by the fault. For test failures caused by more than one atomic changes, we present fault localization results in order, e.g., for faults in `testEmptyCatalog` and `testNonExistentEntry`. On average, among 3932 atomic changes, the change impact analysis technique of FAULTTRACER is able to identify 68.83 affecting changes for failed tests. Furthermore, the fault localization techniques of FAULTTRACER are even more precise: Tarantula and SBI are able to localize failure-inducing changes within 8.50 changes (i.e., 12.35% of affecting changes), while Jaccard and Ochiai are able localize failure-inducing changes within 10.83 and 14.66 changes (i.e., 15.86% and 21.30% of affecting changes), respectively. Moreover, the Tarantula and SBI techniques of FAULTTRACER are able to localize failure-inducing changes within top 3 changes for 14 of all the 22 regression test failures, even including the two regression test failures caused by multiple inducing changes. For example, for the failed test `EchoPropertiesTest.testEchoToBadFile`, the change impact analysis technique of FAULTTRACER determines 182 affecting changes out of 5019 changes, then the Tarantula and SBI techniques of FAULTTRACER further localize the fault-inducing `CM(EchoProperties.execute)` as the number 1 failure-inducing change! Even for the failed test `TaskdefTest.test5`, where FAULTTRACER ranks the real failure-inducing change as 52th among affecting changes, FAULTTRACER is still able to localize failure-inducing change within 50% of affecting changes.

Ranking method-level affecting changes. Ren et al.'s heuristic algorithm [18] ranks affecting method changes based on the calling structure of failed tests. To compare FAULTTRACER and Ren et al.'s technique fairly, we degrade FAULTTRACER to rank only method-level changes. The detailed results for this comparison are shown by the box plots in Figure 4. In the figure, the *X-axis* denotes the techniques compared (the first four techniques correspond to the four techniques of FAULTTRACER, and "Ren et al." denotes the heuristic algorithm proposed by Ren et al. [18]). The *Y-axis* denotes the rank of failure-inducing changes, i.e., the number of changes developers must inspect to discover a real fault. Generally speaking, Tarantula and SBI ranking techniques of FAULTTRACER show the best performance among the compared techniques, and all the four techniques of FAULTTRACER outperform Ren et al.'s heuristic algorithm. Furthermore, the average performance of Ren et al.'s heuristic algorithm on all the test failures places the failure-inducing change within top 5.88 affecting method changes, while the average performance of our adaptation

TABLE VII
COMPARISON OF FAULTTRACER WITH CHIANTI IN SELECTING AFFECTED TESTS.

Pair	#AffTest		%AffTest		#Diff_Sel		#Unsel_Fail	
	Ch	FT	Ch	FT	Ch	FT	Ch	FT
< j_0, j_1 >	0	0	0.00	0.00	0	0	0	0
< j_1, j_2 >	15	15	11.90	11.90	0	0	0	0
< j_2, j_3 >	56	56	43.75	43.75	0	0	0	0
< x_0, x_1 >	92	92	86.79	86.79	0	0	0	0
< x_1, x_2 >	61	53	66.30	57.60	8	0	0	0
< x_2, x_3 >	67	67	71.27	71.27	0	0	0	0
< m_0, m_1 >	57	57	81.42	81.42	0	0	0	0
< m_1, m_2 >	67	68	87.17	87.17	0	0	0	0
< m_2, m_3 >	72	72	90.00	90.00	0	0	0	0
< m_3, m_4 >	58	50	74.35	64.10	11	4	0	0
< m_4, m_5 >	57	57	73.07	73.07	0	0	0	0
< a_0, a_1 >	101	101	90.17	90.17	0	0	0	0
< a_1, a_2 >	123	123	89.78	89.78	0	0	0	0
< a_2, a_3 >	47	47	21.46	21.46	0	0	0	0
< a_3, a_4 >	201	203	91.78	92.69	0	2	0	0
< a_4, a_5 >	446	446	85.60	85.60	0	0	0	0
< a_5, a_6 >	45	45	8.07	8.07	0	0	0	0
< a_6, a_7 >	516	511	92.30	91.41	2	0	0	0
< a_7, a_8 >	369	369	42.07	42.07	0	0	0	0
average	136	135	63.54	62.54	-	-	-	-

of Tarantula and SBI techniques places the failure-inducing change within top 13.44 affecting method changes, indicating an improvement of more than 50% in accuracy.

In summary, FAULTTRACER is able to produce a high accuracy in localizing all types of affecting changes. Even when localizing only method related changes, FAULTTRACER is able to outperform the existing technique by more than 50%.

E. RQ2: Comparison of Chianti and FAULTTRACER *in terms of Change Impact Analysis Accuracy*

In this section, we compare FAULTTRACER against Chianti in terms of both selecting affected tests and determining affecting changes.

Selecting affected tests. For each version pair, we record the number of tests (Columns 2 and 3), and the ratio of selected tests (Columns 4 and 5) by Chianti and FAULTTRACER. In addition, we also record the number of tests selected by Chianti but not by FAULTTRACER (Column 6), the number of tests selected by FAULTTRACER but not by Chianti (Column 7), and the number of failed tests which are unselected by Chianti and FAULTTRACER (Columns 8 and 9). First, both Chianti and FAULTTRACER are able to select tests effectively: on average, Chianti selects 63.54% of all the tests, while FAULTTRACER's test selection is slightly better: 62.54%. Second, as Columns 8 and 9 show, both Chianti and FAULTTRACER select all tests that must re-run to guarantee safety. Indeed, both of the compared techniques are safe to the extent of this study.

We further examine the tests that are selected by Chianti but not FAULTTRACER. For < x_1, x_2 >, all 8 tests exclusively selected by Chianti execute no changes but the constructor change on class `signature.XMLSignatureInput`. Similarly, for < m_3, m_4 >, all 11 tests exclusively selected by Chianti only execute the constructor change on class `control.GenericController`, and for < a_6, a_7 >, the two tests exclusively selected by Chianti only execute the constructor change on class `ant.ProjectComponent`. By comparing these constructors before and after edits, we find that they did not change, and are all transformed from field changes not executed by these tests. Therefore, the tests exclusively selected by Chianti are not actually required to re-run.

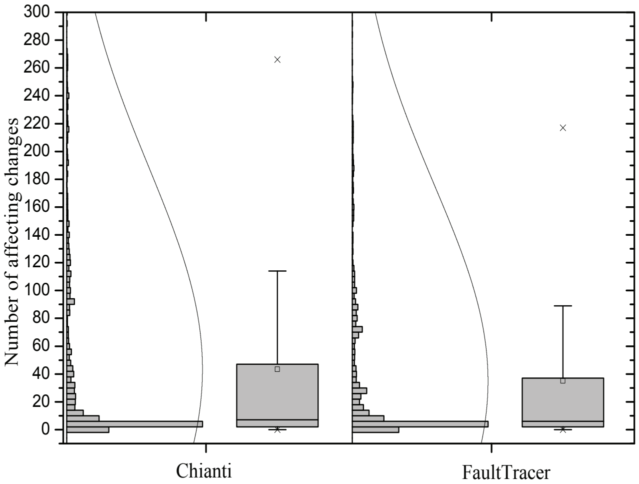


Fig. 5. Comparison of FAULTTRACER with Chianti in determining affecting changes.

We also examine the tests selected by FAULTTRACER but not Chianti. For < a_3, a_4 >, both of the two tests selected only by FAULTTRACER (i.e.,`testExtraFields` of test class `zip.ZipEntryTest` and `testMessage` of test class `zip.ExtraFieldUtilsTest`) read static field `HEADER_ID` of class `zip.AsiExtraField`, which is the only changed element they executed. Chianti does not select these two tests, because the class `zip.AsiExtraField` has been visited by other tests, making constructor `AsiExtraField.Clinit` not appear in the call graphs of `testExtraFields` and `testMessage`. These two tests do not fail after edits because the developers changed only the modifier of `HEADER_ID`. If developers changed the value of `HEADER_ID`, then these two tests might fail, making traditional change impact analysis unsafe in test selection. Similarly, for < m_3, m_4 >, each of the four test cases exclusively selected by FAULTTRACER executes only one `CSFI` atomic change, e.g., `testTestElements` of test class `junit.JMeterTest` only executes static field `log` of class `reflect.ClassFinder`. However, all the changes executed by the four tests are log object changes, which do not influence the functionality of tested programs. If the changes happen to functional fields of tested programs, Chianti would have been unsafe in test selection.

Selecting affecting changes. For all the affected tests of the 19 version pairs, Figure 5 displays the distribution histograms for the number of affecting changes determined by FAULTTRACER and Chianti along with standard quantile box plot to visualize the distribution's main descriptive statistics. The box plot shows the median, average, various quantiles (e.g., 1%, 25%, 75%, 99%), and the minimum and maximum. From the distribution histograms, we can find that the number of affecting changes for each affected test determined by FAULTTRACER tends to spread at lower ranges than that determined by Chianti, indicating FAULTTRACER's effectiveness in avoid selecting unnecessary changes. Furthmore, shown by the quantile box, on average, Chianti successfully determines 43.36 affecting changes, while FAULTTRACER successfully determines 34.96 affecting changes for all affected tests, indicating an improvement of approximately 20% on average. More precisely, while Chianti has to report matching changes on call graphs as well as changes which the matching changes transitively depend on, FAULTTRACER directly determines affecting changes through the use of ECGs and identify affecting changes to smaller ranges in 17 out of 19 version pairs with improvements ranging from 0.55% to 40.20%. The only exceptions are fault localization for <j_0, j_1> and <j_2, j_3>. In version pair <j_0, j_1>, both Chianti and FAULTTRACER successfully detect that no tests are influenced, and there are no affecting changes. In version pair <j_2, j_3>, all atomic changes are AM or AF changes except five CM changes and one DM change. The DM change is not affecting change, while the five CM changes are mainly refactoring changes, and does not invoke newly added methods or access newly added fields. Thus, they are not dependent on other changes. Therefore, Chianti and FAULTTRACER report the same number of affecting changes for all affected tests of this version pair.

In summary, in terms of regression test selection, although FAULTTRACER only performs slightly better than Chianti in test selection, FAULTTRACER avoids the safety problem of Chianti. In terms of affecting change determination, FAULTTRACER outperforms Chianti by approximately 20% on average.

V. CONCLUSION AND FUTURE WORK

To conclude, this paper presents a novel fault localization framework, FAULTTRACER, which adapts spectrum-based ranking techniques to change impact analysis to reduce developer effort in manually inspecting potential failure-inducing edits. In addition, to collect spectrum information for program edits, we improve upon Chianti-style change impact analysis to associate tests with program edits more accurately. Using the version histories of real world software projects, we conducted an empirical study to evaluate the performance of FAULTTRACER in terms of both fault localization and change impact analysis. The experimental results show that, based on test spectra, FAULTTRACER produces a high accuracy in ranking all types of affecting changes. Also, even when ranking only method related changes, FAULTTRACER reduces the number of changes to be manually inspected before discovering the real fault by more than 50% compared to Ren et al.'s heuristic algorithm [18]. Furthermore, our ECG-based change impact analysis technique improves upon the precision of Chianti [19] in terms of both reducing the number of selected tests and reducing the number of affecting changes.

In terms of future work, we plan to integrate FAULTTRACER with a refactoring reconstruction technique such as RefFinder [16] to further filter out non-semantic program edits. A rigorous comparison of different ranking techniques in the context of change impact analysis with a large set of subject programs is also planed.

ACKNOWLEDGMENTS

The work was funded in part by the NSF under Grant, CCF-1043810, CCF-1117902, IIS-0438967 and CCF-0845628, AFOSR grant FA9550-09-1-0351, and the Microsoft SEIF Award.

REFERENCES

[1] http://asm.ow2.org/.

[2] http://www.eclipse.org/jdt/.

[3] R. Abreu, P. Zoeteweij, and A. Van Gemund. On the accuracy of spectrum-based fault localization. 2007.

[4] B. Baudry, F. Fleurey, and Y. Le Traon. Improving test suites for efficient fault localization. In *Proc. of ICSE*, pages 82–91. ACM, 2006.

[5] S. Bohner and R. Arnold. *Software change impact analysis*. Wiley-IEEE Computer Society Pr, 1996.

[6] O. Chesley, X. Ren, and B. Ryder. Crisp: A debugging tool for Java programs. 2005.

[7] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[8] D. Hao, Y. Pan, L. Zhang, W. Zhao, H. Mei, and J. Sun. A similarity-aware approach to testing based fault localization. In *Proc. of ASE*, pages 291–294. ACM, 2005.

[9] D. Hao, L. Zhang, T. Xie, H. Mei, and J.-S. Sun. Interactive fault localization using test information. *Journal of Computer Science and Technology*, 24(5), 2009.

[10] B. Jiang, Z. Zhang, T. Tse, and T. Chen. How well do test case prioritization techniques support statistical fault localization. In *Proc. of COMPSAC*, pages 99–106, 2009.

[11] J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proc. of ICSE*, page 477. ACM, 2002.

[12] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proc. of ICSE*, pages 308–318, 2003.

[13] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *Proc. of PLDI*, pages 15–26. ACM, 2005.

[14] A. Orso, T. Apiwattanapong, and M. Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. of FSE*, pages 128–137. ACM, 2003.

[15] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proc. of ICSE*, pages 491–500. IEEE Computer Society, 2004.

[16] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based Reconstruction of Complex Refactorings. In *Proc. of ICSM*, 2010.

[17] X. Ren, O. Chesley, and B. Ryder. Identifying failure causes in java programs: An application of change impact analysis. *IEEE TSE*, pages 718–732, 2006.

[18] X. Ren and B. Ryder. Heuristic Ranking of Java Program Edits for Fault Localization. In *Proc. of ISSTA*, 2007.

[19] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of java programs. In *Proc. of OOPSLA*. Citeseer, 2004.

[20] B. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proc. of PASTE*, pages 46–53. ACM, 2001.

[21] R. Santelices, J. Jones, Y. Yu, and M. Harrold. Lightweight fault-localization using multiple coverage types. In *Proc. of ICSE*, pages 56–66, 2009.

[22] M. Stoerzer, B. Ryder, X. Ren, and F. Tip. Finding failure-inducing changes in java programs using change classification. In *Proc. of FSE*, pages 57–68. ACM, 2006.

[23] Y. Yu, J. Jones, and M. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proc. of ICSE*, pages 201–210. ACM, 2008.

[24] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proc. of FSE*, pages 253–267. Springer, 1999.