FaultTracer: A Change Impact and Regression Fault Analysis Tool for Evolving Java Programs

Lingming Zhang, Miryung Kim, Sarfraz Khurshid

The University of Texas at Austin
Austin TX, USA
zhanglm@utexas.edu, {miryung, khurshid}@ece.utexas.edu

ABSTRACT

Keeping evolving software fault-free is hard. In our previous work, we proposed FaultTracer, a change impact and regression fault analysis tool for evolving programs. It takes the old and new versions of a program and a regression test suite as inputs, and then identifies affected tests—a subset of tests relevant to the program differences between the two versions and affecting changes—a subset of atomic changes relevant to each affected test. It adapts spectrum-based fault localization techniques and applies them in tandem with an enhanced change impact analysis to identify and rank failure-inducing program edits. We have shown that FaultTracer, compared to existing techniques (e.g., Chianti), achieves improvement in selecting influenced tests, determining suspicious failure-inducing edits, and ranking failure-inducing program edits. In this paper, we show the design, implementation, and demonstration of our FaultTracer approach as a publicly available toolkit for testing and debugging Java programs.

Categories and Subject Descriptors

D2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Design, Experimentation

Keywords

Regression Testing, Fault Localization, Software Evolution

1. INTRODUCTION

During software evolution, regression test suites have been utilized to validate the program edits. A large body of research is dedicated to executing regression test suites efficiently and localizing faults effectively when regression tests fail. For example, *Chianti-style change impact analysis* [5] selects affected tests—a subset of tests whose behaviors have been influenced by program edits, and determines affecting changes—suspicious program edits that might have caused test failures. In our previous work, we proposed FaultTracer [6], which extends Chianti-style change impact analysis and applies spectrum-based fault localization [1–3] in tandem to rank affecting changes.

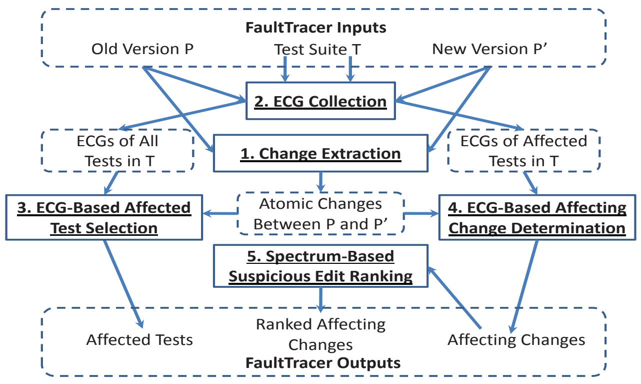
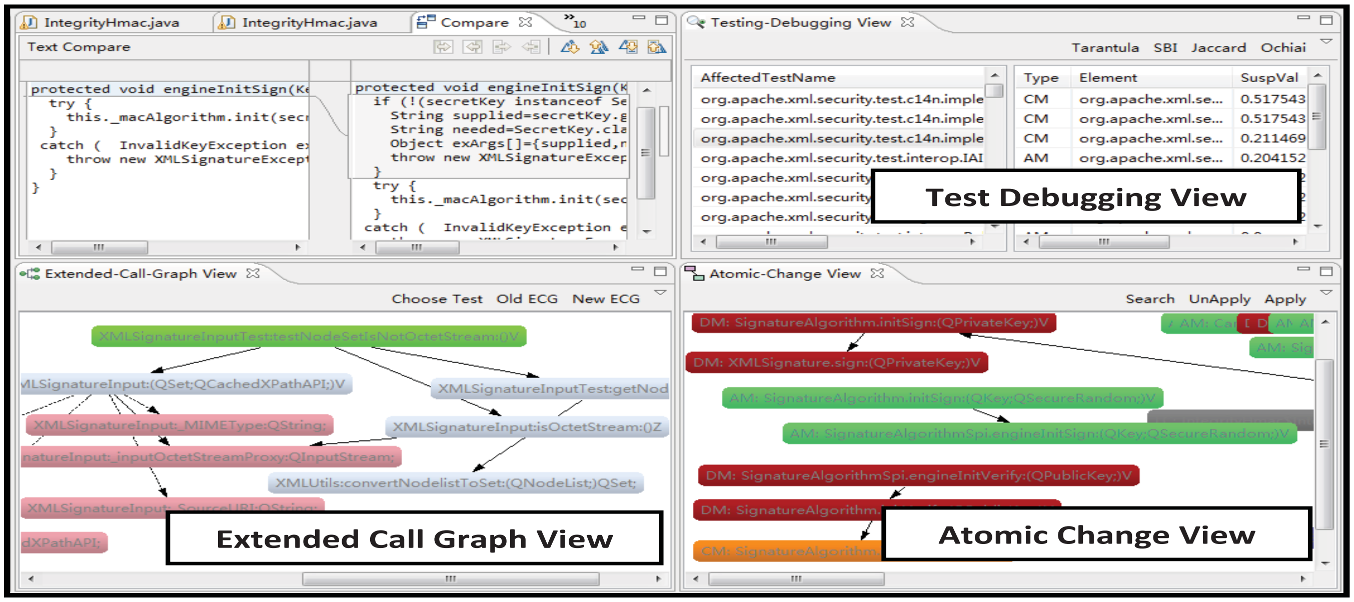


Figure 1: FaultTracer architecture.

Figure 1 gives an overview of the design of FaultTracer [6]. It takes two program versions P and P' and a regression test suite T as inputs. In Step 1, FaultTracer identifies atomic changes between P and P' and determines dependences among the changes based on a set of prescribed atomic change dependence rules [5,6]. In Step 2, FaultTracer constructs the extended call graphs (ECGs) for all the tests in T on the old version P. Compared with traditional call graphs used by Chianti [5], ECGs model field reads and writes directly in addition to calls between methods. In Step 3, based on refined selection using ECGs, FaultTracer selects T', a subset of tests whose behaviors might have been influenced by the program edits—*affected tests*. In Step 4, based on refined determination using ECGs, it runs the selected tests on the new version P' and determines suspicious edits that might have caused each test failure—*affecting changes*. Finally, to localize failure-inducing edits more precisely, FaultTracer further ranks the affecting changes for each failed test based on the aggregated execution profile of passed and failed tests (i.e., spectrum information). In this paper, we describe the design and implementation of our FaultTracer tool, which is publicly available, and give a demonstration of how to use it.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA.
Copyright 2012 ACM 978-1-4503-1614-9/12/11 ...$15.00.


IntegrityHmac.java
IntegrityHmac.java
Compare
Text Compare
Testing-Debugging View
Tarantula
SBI
Jaccard
Ochiai
Test Debugging View
Extended-Call-Graph View
Choose Test
Old ECG
New ECG
XMLSignatureInputTest:testNodeSetIsNotOctetStream:()V
XMLSignatureInput:(QSet;QCachedXPathAPI;)V
XMLSignatureInputTest:getNod
XMLSignatureInput:_MIMEType:QString;
XMLSignatureInput:isOctetStream:()Z
natureInput:_inputOctetStreamProxy:QInputStream;
XMLUtils:convertNodelistToSet:(QNodeList;)QSet;
XMLSignatureInput:_SourceURI:QString;
dXPathAPI;
Extended Call Graph View
Atomic-Change View
Search
UnApply
Apply
DM: SignatureAlgorithm.initSign:(QPrivateKey;)V
AM: Ca
AM: Sig
DM: XMLSignature.sign:(QPrivateKey;)V
AM: SignatureAlgorithm.initSign:(QKey;QSecureRandom;)V
AM: SignatureAlgorithmSpi.engineInitSign:(QKey;QSecureRandom;)V
DM: SignatureAlgorithmSpi.engineInitVerify:(QPublicKey;)V
DM: SignatureAlgorithm
CM: SignatureAlgorithm
Atomic Change View

```
protected void engineInitSign(Ke
  try {
    this._macAlgorithm.init(secr
  }
 catch (  InvalidKeyException ex
    throw new XMLSignatureExcept
  }
}
```

```
protected void engineInitSign(K
  if (!(secretKey instanceof Se
    String supplied=secretKey.g
    String needed=SecretKey.cla
    Object exArgs[]={supplied,n
    throw new XMLSignatureExcep
  }
  try {
    this._macAlgorithm.init(sec
  }
 catch (  InvalidKeyException e
```

AffectedTestName
org.apache.xml.security.test.c14n.imple
org.apache.xml.security.test.c14n.imple
org.apache.xml.security.test.c14n.imple
org.apache.xml.security.test.interop.IAI
org.apache.xml.security
org.apache.xml.security
org.apache.xml.security

Type	Element	SuspVal
CM	org.apache.xml.se...	0.517543
CM	org.apache.xml.se...	0.517543
CM	org.apache.xml.se...	0.211469
AM	org.apache.xml.se...	0.204152

Figure 2: Overview of FaultTracer's front-end.

2. FAULTTRACER IMPLEMENTATION

FaultTracer is implemented as a toolkit, which includes the front-end plug-in, and the back-end library. The following subsections present the details of the toolkit.

2.1 FaultTracer Plug-in

The front-end of FaultTracer is implemented as an Eclipse IDE plugin. The plugin takes two program versions as input and extracts atomic program changes from the two versions based on the abstract syntax tree (AST) analysis provided by the Eclipse JDT toolkit.[1] It traverses the ASTs of two versions to compare fields and methods by their fully qualified names to find atomic changes. For each pair of compared methods, FaultTracer filters out all comments and white-spaces before comparison. FaultTracer also finds call and access dependencies between atomic changes by tracing the definition and reference of each used method and field.

The FaultTracer plugin also includes three views to visualize internal and final outputs of FaultTracer (Figure 2):

Atomic-change view is implemented using the Eclipse Zest Visualization Toolkit.[2] It visualizes the all atomic changes between program versions and their dependencies, and supports various user interaction (details shown in Appendix). Note that this view depends on the data produced by Step 1 in Figure 1.

Extended-call-graph view is also implemented using the Eclipse Zest Visualization Toolkit. It visualizes the extended call graphs for individual tests. This view can help the user to better understand the behaviors of individual tests. This view depends on the data produced by Step 2 in Figure 1.

Testing-debugging view lists the affected tests between two compared versions, the affecting changes for each affected test, and the ranked list of affecting changes for each failed test. This view visualizes all the final outputs of FaultTracer. When the user double-clicks an affected test in the view, the view immediately displays the affecting changes for the selected test. The view would also display the ranked list of affecting changes for the test along with their suspiciousness scores computed based on program spectra. When the user double-clicks any affecting change in the view, FaultTracer extracts corresponding changed code fragments in the Java Editor to facilitate manual inspection of relevant code. Note that this view uses the data produced by Steps 3, 4, and 5 respectively.

2.2 FaultTracer Library

The back-end of FaultTracer is implemented as Ant[3] tasks, which fully automate the process of constructing extended call graphs, selecting affected tests, determining affecting changes, and ranking affecting changes for failed tests. The back-end performs the ECG construction task on-the-fly through byte code instrumentation. It dynamically instruments classes loaded into the JVM through a Java agent without any modification of the target program. For instrumentation, FaultTracer uses the ASM[4] bytecode manipulation and analysis framework. We extend `visitor` classes in ASM 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, and types of field accesses).

The back-end of FaultTracer also performs all the core analysis tasks: selection of affected tests, determination of affecting changes, and spectrum-based ranking of affecting-changes. The final results are then visualized by the front-end plugin.

3. DEMONSTRATION

This section illustrates how to configure FaultTracer and how to perform the five key steps for regression test execu-

[1]`http://www.eclipse.org/jdt/`. Accessed in Aug. 2012.
[2]`http://www.eclipse.org/gef/zest/`. Accessed in Aug. 2012.

[3]`http://ant.apache.org/`. Accessed in Aug. 2012.
[4]`http://asm.ow2.org/`. Accessed in Aug. 2012.

```
<project name="FaultTracer-Configuration">
 <property name="subject" value="..."/>
 <property name="prefix" value="..."/>
 <property name="testsuite" value="..."/>
 <property name="newversion" value="..."/>
 <property name="faulttracer" value="..."/>
 <property name="cp" value="..."/>
 <property name="maxmemory" value="..."/>
  <import file="${faulttracer}\resources\
      faulttracer.xml"/>
</project>
```

Figure 3: FaultTracer's user config file.

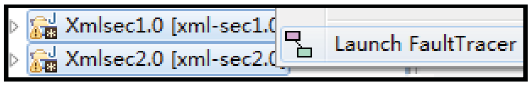


Figure 4: Launch FaultTracer and extract program edits.

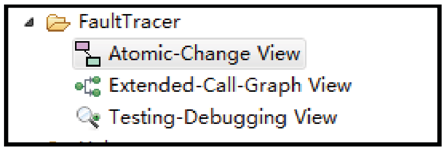


Figure 5: Open FaultTracer views.

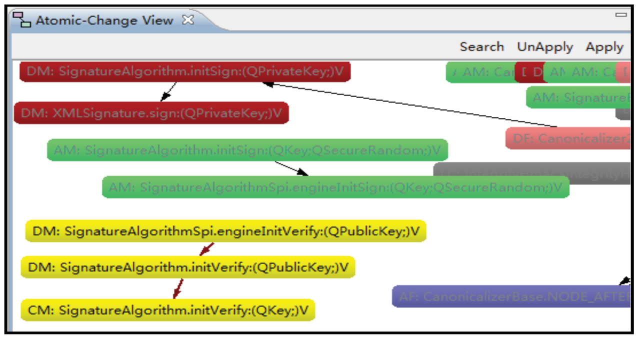
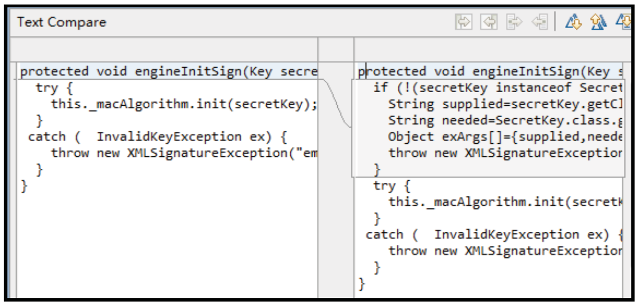

Atomic-Change View
Search
UnApply
Apply
DM: SignatureAlgorithm.initSign:(QPrivateKey;)V
DM: XMLSignature.sign:(QPrivateKey;)V
AM: SignatureAlgorithm.initSign:(QKey;QSecureRandom;)V
AM: SignatureAlgorithmSpi.engineInitSign:(QKey;QSecureRandom;)V
DM: SignatureAlgorithmSpi.engineInitVerify:(QPublicKey;)V
DM: SignatureAlgorithm.initVerify:(QPublicKey;)V
CM: SignatureAlgorithm.initVerify:(QKey;)V
DF: Canonicalizer
AF: CanonicalizerBase.NODE_AFTE

Figure 6: FaultTracer's atomic-change view.



```
protected void engineInitSign(Key secre
  try {
    this._macAlgorithm.init(secretKey);
  }
 catch (  InvalidKeyException ex) {
    throw new XMLSignatureException("em
  }
}
```

```
protected void engineInitSign(Key s
  if (!(secretKey instanceof Secret
    String supplied=secretKey.getCl
    String needed=SecretKey.class.g
    Object exArgs[]={supplied,neede
    throw new XMLSignatureExceptior
  }
  try {
    this._macAlgorithm.init(secretk
  }
 catch (  InvalidKeyException ex) {
    throw new XMLSignatureExceptior
  }
}
```

Figure 7: Detailed changes shown by FaultTracer.

tion and fault localization using the tool. Consider the two versions *xml-sec1.0* and *xml-sec2.0* of the open-source application XML-security (as available in the Software-artifact Infrastructure Repository[5]).

The user provides the configuration for each program version under test as an XML file called *config.xml*. Figure 3 shows a skeletal configuration: `subject` is the name of the version under test; `prefix` is the common package prefix (shared by the project); `testsuite` is the fully qualified name for the test suite class; `newversion` is the absolute path of the next version with respect to the current version; `faulttracer` is the absolute path of the FaultTracer back-end (i.e., the path to the unzipped FaultTracer library); `cp` is the class path needed for executing the program under test; and `maxmemory` is the maximum memory allowed for executing FaultTracer. In this demonstration we use the same setup as described in detail in our ICSM paper [6].

3.1 Extract Program Edits

Execution of FaultTracer begins by extracting program edits and collecting ECGs for regression tests. To extract program edits, the user right-clicks the two program versions and selects *"Launch FaultTracer"* (Figure 4). The user can view the visualized atomic changes in the *Atomic-Change View*. The complete list of FaultTracer views is accessible using the following Eclipse menu: *"Window"* → *"Show View"* → *"Other"* → *"FaultTracer"* (Figure 5). Figure 6 illustrates a subset of the atomic changes between *xml-sec1.0* and *xml-sec2.0*. As illustrated, the user can search a specific change node, apply and unapply a specific change, as well as change the graph layout by right-clicking it. When the user clicks one change node, FaultTracer highlights all the changes that the selected change transitively depends on (marked as yellow nodes in Figure 6). When the user double clicks a specific change node, FaultTracer shows change details in the Java Editor. For example, Figure 7 illustrates the Java Editor content after the user double clicks the change node for method `IntegrityHmac.engineInitSign(Key)`.

[5]`http://sir.unl.edu/`. Accessed in Aug. 2012.

3.2 Collect ECGs

Since the collection of ECGs is implemented as an Ant task, the user can construct ECGs of the program under test using the following command-line script:

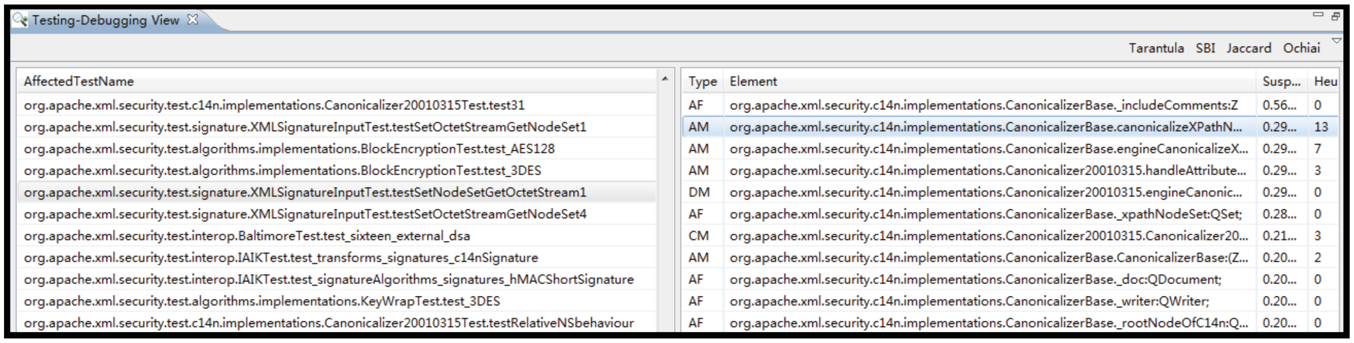
```
cd path-of-program-to-trace
ant -f config.xml collectECGCoverage
```

This script navigates to the base directory of the program under test, and then runs the Ant task `collectECGCoverage`. Then FaultTracer automatically runs the test suite for the program and records an ECG for each test. The user can choose to see the visualized ECG using *Extended-Call-Graph View*. Figure 8 illustrates the ECG for test `XMLSignatureInputTest.testNodeSetIsNotOctetStream()`, where the green nodes denote the test, blue nodes denote methods invoked, pink nodes denote fields accessed, solid arrows denote method invocations, and dashed arrows denote field accesses. When the user clicks a node, FaultTracer highlights all the methods and fields transitively invoked or accessed by the node (marked as yellow nodes in Figure 8). The user can select a test to display by clicking the *"Choose Test"* menu. The user can also choose to view the ECG on the old or the new version of the chosen test by clicking *"Old ECG"* or *"New ECG"*.

3.3 Select Affected Tests

Once the program edits and ECGs are computed, the user can select affected tests as follows:

```
cd path-of-xml-sec1.0
ant -f config.xml selectAffectedTests
```



AffectedTestName
org.apache.xml.security.test.c14n.implementations.Canonicalizer20010315Test.test31
org.apache.xml.security.test.signature.XMLSignatureInputTest.testSetOctetStreamGetNodeSet1
org.apache.xml.security.test.algorithms.implementations.BlockEncryptionTest.test_AES128
org.apache.xml.security.test.algorithms.implementations.BlockEncryptionTest.test_3DES
org.apache.xml.security.test.signature.XMLSignatureInputTest.testSetNodeSetGetOctetStream1
org.apache.xml.security.test.signature.XMLSignatureInputTest.testSetOctetStreamGetNodeSet4
org.apache.xml.security.test.interop.BaltimoreTest.test_sixteen_external_dsa
org.apache.xml.security.test.interop.IAIKTest.test_transforms_signatures_c14nSignature
org.apache.xml.security.test.interop.IAIKTest.test_signatureAlgorithms_signatures_hMACShortSignature
org.apache.xml.security.test.algorithms.implementations.KeyWrapTest.test_3DES
org.apache.xml.security.test.c14n.implementations.Canonicalizer20010315Test.testRelativeNSbehaviour

Type	Element	Susp...	Heu
AF	org.apache.xml.security.c14n.implementations.CanonicalizerBase._includeComments:Z	0.56...	0
AM	org.apache.xml.security.c14n.implementations.CanonicalizerBase.canonicalizeXPathN...	0.29...	13
AM	org.apache.xml.security.c14n.implementations.CanonicalizerBase.engineCanonicalizeX...	0.29...	7
AM	org.apache.xml.security.c14n.implementations.Canonicalizer20010315.handleAttribute...	0.29...	3
DM	org.apache.xml.security.c14n.implementations.Canonicalizer20010315.engineCanonic...	0.29...	0
AF	org.apache.xml.security.c14n.implementations.CanonicalizerBase._xpathNodeSet:QSet;	0.28...	0
CM	org.apache.xml.security.c14n.implementations.Canonicalizer20010315.Canonicalizer20...	0.21...	3
AM	org.apache.xml.security.c14n.implementations.CanonicalizerBase.CanonicalizerBase:(Z...	0.20...	2
AF	org.apache.xml.security.c14n.implementations.CanonicalizerBase._doc:QDocument;	0.20...	0
AF	org.apache.xml.security.c14n.implementations.CanonicalizerBase._writer:QWriter;	0.20...	0
AF	org.apache.xml.security.c14n.implementations.CanonicalizerBase._rootNodeOfC14n:Q...	0.20...	0

Figure 9: **FaultTracer's testing-debugging view.**

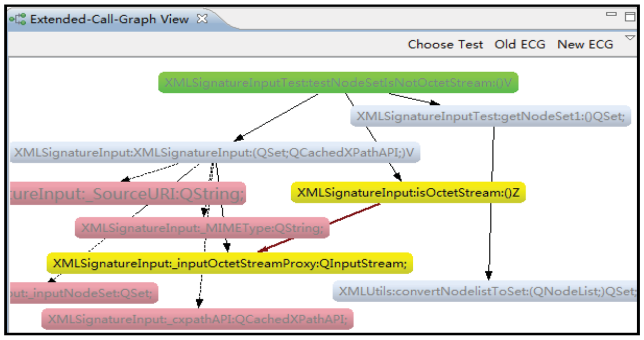

Extended-Call-Graph View
Choose Test
Old ECG
New ECG
XMLSignatureInputTest:testNodeSetIsNotOctetStream:()V
XMLSignatureInputTest:getNodeSet1:()QSet;
XMLSignatureInput:XMLSignatureInput:(QSet;QCachedXPathAPI;)V
tureInput:_SourceURI:QString;
XMLSignatureInput:isOctetStream:()Z
XMLSignatureInput:_MIMEType:QString;
XMLSignatureInput:_inputOctetStreamProxy:QInputStream;
put:_inputNodeSet:QSet;
XMLUtils:convertNodelistToSet:(QNodeList;)QSet;
XMLSignatureInput:_cxpathAPI:QCachedXPathAPI;

Figure 8: **FaultTracer's extended-call-graph view.**

This script will select affected tests due to the edits between *xml-sec1.0* and *xml-sec2.0*. Next, the user can selectively run the affected tests on the new version:

```
cd path-of-xml-sec2.0
ant -f config.xml runAffectedTests
```

Between *xml-sec1.0* and *xml-sec2.0*, 64 of all the 94 regression tests cases are selected by FaultTracer because their behaviors may be different after the edits.

3.4 Determine Affecting Changes

When some affected tests fail on the new version (i.e., *xml-sec2.0*), the user can determine the affecting changes using the following script:

```
cd path-of-xml-sec2.0
ant -f config.xml determineAffectingChanges
```

Once the affecting changes are determined, the user can view the affecting changes for each affected test in the *Testing-Debugging View* (Figure 9). The table on the left side of the view lists all the affected tests. When the user double-clicks an affected test in this table, the view displays all the affecting changes for the test in the table on the right side. The view displays all suspicious values as 0 and does not rank affecting changes.

3.5 Rank Affecting Changes

To localize failure-inducing edits more precisely, the user can rank affecting changes using the following script:

```
cd path-of-xml-sec2.0
ant -f config.xml rankAffectingChanges
```

The user can then view the ranked list of affecting changes for each affected test in the *Testing-Debugging View* (Figure 9). The view ranks affecting changes and displays suspicious values for affecting changes. As the figure shows, when the user double-clicks the failed test `XMLSignatureInputTest.testSetNodeSetGetOctetStream1`, the tool ranks `AM: CanonicalizerBase.canonicalizeXPathNodeSet`, which is the cause of the test failure, as the second most likely failure-inducing edit in the list of 11 edits that are determined as affecting changes (from the full set of 329 edits between *xml-sec1.0* and *xml-sec2.0*).

4. SUMMARY

In this paper, we present the design, implementation, and demonstration of our change impact and regression fault analysis approach, FaultTracer. Our tool is publicly available at `https://webspace.utexas.edu/~lz3548/ftracer.html`. Developers may use our tool for running regression tests and debugging regression faults. According to our previous evaluation [6], FaultTracer outperforms Chianti [5] in both selecting affected tests and determining affecting changes. In terms of ranking affecting changes, it also outperforms Ren et al.'s ranking heuristic [4] by more than 50%.

5. ACKNOWLEDGEMENT

This work was in part supported by National Science Foundation under the grants CCF-0845628, CCF-1117902, CCF-1149391, and CCF-1043810, AFOSR grant FA9550-09-1-0351, and the Microsoft SEIF award.

6. REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. Van Gemund. On the accuracy of spectrum-based fault localization. In *TAIC-PART*, 2007.
[2] J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, 2002.
[3] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.
[4] X. Ren and B. Ryder. Heuristic ranking of java program edits for fault localization. In *ISSTA*, 2007.
[5] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *OOPSLA*, 2004.
[6] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *ICSM*, 2011.