Collaborative Software Design & Development

Mining Software Repositories

Peter Kim and Birgi Tamersoy
The University of Texas at Austin
February 26th, 2008
Overview

→ Relevance to the course

→ An Overview of Mining Software Repositories

→ Instance 1: ROSE for change prediction
  ¦ Explanation
  ¦ Critique
  ¦ Analysis

→ Instance 2: Refining static analysis for return value checking
  ¦ Explanation
  ¦ Critique
  ¦ Analysis

→ State of the Art
Relevance to the course

→ So far, regarding OSS, we talked about...
  - Process and organization structure
  - Distributed nature
  - Case studies

→ We have the people and the software but...
  - Both the people and the software will change
  - Software evolution is difficult especially in OSS development
    - Developers do not have easy access to help
Guiding developers through software evolution

→ Static analysis
  ✴ Addition of a method call (e.g. register) requires addition of a matching method call (e.g. deregister)

→ Informal documents
  ✴ “To add an action to a toolbar, you must register it and then deregister it”.

→ Patterns
  ✴ “In frameworks, initialization must typically be followed by cleanup.”

→ ...

→ Mining software repositories
  ✴ In the past, when registration code was added, deregistration code was added as well
Overview

→ Relevance to the course

→ An Overview of Mining Software Repositories

→ Instance 1: ROSE for change prediction
  ⇨ Explanation
  ⇨ Critique
  ⇨ Analysis

→ Instance 2: Refining static analysis for return value checking
  ⇨ Explanation
  ⇨ Critique
  ⇨ Analysis

→ State of the Art
Mining Software Repositories

⇒ Purpose
  ⇨ To understand software development
  ⇨ To support predictions about software development
  ⇨ To plan future development

⇒ A hot research area
  ⇨ A working conference dedicated to this topic since 2004:
    (Professor Perry is on the PC)

⇒ We focus on two particular works in MSR
  ⇨ ROSE: Association rule mining to guide changes
  ⇨ Refining static analysis to improve bug fixes
Overview

→ Relevance to the course
→ An Overview of Mining Software Repositories
  → Instance 1: ROSE for change prediction
    - Explanation
    - Critique
    - Analysis
  → Instance 2: Refining static analysis for return value checking
    - Explanation
    - Critique
    - Analysis
→ State of the Art
According to CVS, Comp.java file changed with plugin.properties 20/40 times

It could be a coincidence

fKeys[] changed with plugin.properties 10/11 times and with initDefaults() 11/11 times

The next time fKeys[] changes, should plugin.properties and initDefaults() change too?
ROSE guides programmers based on evolutionary coupling

Customers Who Bought This Item Also Bought

- iPod Video / Classic Genuine Leather Case for 30GB, 60GB, 80GB...
  ★★★☆☆ (5) $12.83

- DLO HipCase Leather Folio for iPod classic (Black)
  ★★★★★ (21) $19.95

- DLO Jam Jacket with Cord Management for the 80 GB iPod Classic...
  ★★★☆☆ (12) $14.48

- iPod Video Silicone Case Package Includes 3Piece Screen Protec...
  ★★★★★ (29)

Programmers who changed “fKeys[]” also changed

<table>
<thead>
<tr>
<th>Action</th>
<th>Symbol</th>
<th>Support</th>
<th>Confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHG</td>
<td>~initDefaults(IPreferenceStore)</td>
<td>11</td>
<td>1.0</td>
</tr>
<tr>
<td>CHG</td>
<td>plugin.properties</td>
<td>10</td>
<td>0.9090</td>
</tr>
<tr>
<td>CHG</td>
<td>buildnotes_compare.html</td>
<td>8</td>
<td>0.7272</td>
</tr>
<tr>
<td>CHG</td>
<td>~createGeneralPage(Composite)</td>
<td>7</td>
<td>0.6363</td>
</tr>
<tr>
<td>ADD_TO</td>
<td>ComparePreferencePage.java</td>
<td>7</td>
<td>0.6363</td>
</tr>
<tr>
<td>CHG</td>
<td>~textCompareViewer(Composite, int)</td>
<td>6</td>
<td>0.5454</td>
</tr>
<tr>
<td>ADD_TO</td>
<td>TextCompareViewer.java</td>
<td>6</td>
<td>0.5454</td>
</tr>
<tr>
<td>CHG</td>
<td>~createTextComparePage(Compo</td>
<td>6</td>
<td>0.5454</td>
</tr>
</tbody>
</table>
ROSE aims to...

- Improve navigation
  - Guide programmers along related changes
  - E.g. `alter(fKeys[])` → `alter(initDefaults())` → `alter(plugin.properties)`

- Prevent errors
  - Detect incomplete changes
  - E.g. If `alter(fKeys[])`, `alter(initDefaults())`, and `alter(plugin.properties)` should occur in the same transaction, but one is missing, warn the user.
Examples of guidance

→ **Coupling in GCC**
  - Arrays define costs of different assembler operations
  - When costs type changes, its instances are changed too

→ **PYTHON and C files**
  - Inter-language connections
  - When C function changes, change PYTHON functions

→ **POSTGRES documentation**
  - Connections between documents
  - When documents about user changes, change documents about database
How ROSE works (1)

Preparation

CVS archive → Data collection and preparation → Data cleaning → Transactions

E.g. Past changes in a commit
- change(fKeys[])
- change(initDefaults())
- change(plugin.properties)

Editor

new changes

Suggestion Generator

Ranked Suggestions
How ROSE works (2)

Preparation

CVS archive

Data collection and preparation

Data cleaning

Transactions

Map the changes into a structured ROSE format

- alter(field, fKeys[], ...)
- alter(method, initDefaults(), ...)
- alter(file, plugin.properties, ...)

Editor

new changes

Suggestion Generator

Ranked Suggestions
How ROSE works (3)

Preparation

CVS archive → Data collection and preparation → Data cleaning → Transactions

Group the changes into a transaction using a sliding window algorithm.

\[ T_i = \{ \text{alter(field, fKeys[], ...), alter(method, initDefaults(), ...), alter(file, plugin.properties, ...)} \} \]
How ROSE works (4)

Preparation

CVS archive

Data collection and preparation

Data cleaning

Transactions

Editor

Suggestion Generator

Ranked Suggestions

The user makes a new change:

\texttt{alter(field, fKeys[], ...)}
How ROSE works (5)

Suggestion generator finds changes that were made with $L = \text{alter}(\text{field}, f\text{Keys}[], \ldots)$ in the same transaction.

For each change $R$ found, ROSE calculates:
1) Support Count = number of transactions with $L$ and $R$
2) Confidence = Support Count / number of transactions with $L$

Each $R$ is cast into a suggestion and ranked by confidence.

Thresholds can be specified on ROSE.
Predictive Power Based on Change Types and Granularity

- Change types
  - `alter`
  - `add_to, delete_from`
    - E.g. `add_to(file, Comp.java, ...)`
    - Abstracts away the element being added or deleted
      - ROSE learns additional, more general rules: if a particular element is specified, it probably will not match later (how many times will you add “FOO” constants to a file?)

- Granularity
  - **Fine-granular**
    - `alter` on fields, functions, subsections, and non-structured (e.g. image) files
    - `add_to, delete_from` on structured files
  - **Coarse-granular**
    - `alter` files
    - `add_to, delete_from` on directories
Evaluation

➔ How effective is ROSE for the main tasks?
  ✷ Navigation
    ➢ Does one change in a transaction imply the rest in it?
  ✷ Error Prevention
    ➢ Can a missing change in a transaction be implied?
  ✷ Closure
    ➢ Does ROSE suggest changes when no changes are needed?

➔ How is the effectiveness influenced by the following?
  ✷ Granularity
    ➢ Working on a file-level, rather than fields, method, and etc.
  ✷ Maintenance
    ➢ Limiting change types to “alter”
  ✷ Multiple dimensions
    ➢ Limiting change types to “add_to” and “delete_from”
  ✷ History
    ➢ Predictive power in day 1 vs. day 1000
  ✷ Recent changes
    ➢ Giving more weight to recent changes
Setup

→ 8 large-open source projects
  ✐ Eclipse, GCC, GIMP, JBOSS, JEDIT, JEDIT, KOFFICE, POSTGRES, PYTHON

→ Use ROSE to “retroactively predict” changes that were actually made

- For each expectation, get the suggestions.
  - E.g. \( e_1 \Rightarrow \{ e_2 \ [\text{Sup. Count}=5, \text{Confidence}=1.0], \)
    \( X \ [2, 0.4], \)
    \( Y \ [1, 0.2] \} \)

- Navigation
  - E.g. \( e_1 \Rightarrow \{ e_2, e_3 \} \)

- Error prevention
  - E.g. \( \{ e_1, e_2 \} \Rightarrow \{ e_3 \} \)

- Closure
  - E.g. \( \{ e_1, e_2, e_3 \} \Rightarrow \{ \} \)

Compare suggestions against expectations.
Comparing Suggestions Against Expectations

- **Precision** (relevant suggestions)
  - Matches / Suggestions
  - E.g. 1/3 = 33%

- **Recall** (expectation coverage)
  - Matches / Expectations
  - E.g. 1/2 = 50%

- **Likelihood** (k)
  - top k suggestions are relevant
  - E.g. L₃=100%.

- **Feedback**
  - Making at least one suggestion
  - E.g. If no suggestions can be made for e2 and e3, then 1/3 = 33%

For the change “e1”...
## CVS Data

### TABLE 1

History of Analyzed Projects (Txn = Coarse-Grained Transaction, “±” = Standard Deviation)

<table>
<thead>
<tr>
<th>Project, Description</th>
<th>Before data cleaning</th>
<th>After data cleaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>in CVS since</td>
<td># Files</td>
</tr>
<tr>
<td>ECLIPSE, integrated environment</td>
<td>2001-04-28</td>
<td>34,186</td>
</tr>
<tr>
<td>GCC, compiler collection</td>
<td>1997-08-11</td>
<td>23,467</td>
</tr>
<tr>
<td>GIMP, image manipulation tool</td>
<td>1997-01-01</td>
<td>3,834</td>
</tr>
<tr>
<td>JBOSS, application server</td>
<td>2000-04-22</td>
<td>4,730</td>
</tr>
<tr>
<td>JEDIT, text editor</td>
<td>2001-09-02</td>
<td>954</td>
</tr>
<tr>
<td>KOFFICE, office suite</td>
<td>1998-04-18</td>
<td>8,098</td>
</tr>
<tr>
<td>POSTGRES, database system</td>
<td>1996-07-09</td>
<td>2,684</td>
</tr>
<tr>
<td>PYTHON, language + library</td>
<td>1990-08-09</td>
<td>3,459</td>
</tr>
</tbody>
</table>

### TABLE 3

Evaluation for Fine Granularity (Txn = Transaction, “±” = Standard Deviation)

| Project     | Navigation / Prevention (|T| ≥ 2) | Closure (T of any size) |
|-------------|--------------------------|--------------------------|
|             | # Txns | # Items/Txn | # Queries | # Txns | # Items/Txn | # Queries |
| ECLIPSE     | 982    | 4.97±4.77 | 4,876     | 2,443  | 2.59±3.59 | 2,443     |
| GCC         | 473    | 4.17±3.51 | 1,972     | 598    | 3.51±3.38 | 598       |
| GIMP        | 1,090  | 6.18±6.12 | 6,733     | 1,201  | 5.70±6.02 | 1,201     |
| JBOSS       | 148    | 5.67±5.51 | 839       | 340    | 3.03±4.31 | 340       |
| JEDIT       | 288    | 10.21±7.41| 2,941     | 323    | 9.21±7.56 | 323       |
| KOFFICE     | 700    | 6.76±5.87 | 4,729     | 1,210  | 4.33±5.29 | 1,210     |
| POSTGRES    | 369    | 6.33±5.98 | 2,337     | 659    | 3.99±5.20 | 659       |
| PYTHON      | 402    | 4.22±3.73 | 1,696     | 904    | 2.43±2.96 | 904       |
Results: Precision vs. Feedback

- Hard to find a set of changes appearing together frequently
- Precise suggestions or many suggestions, but not both

Evaluation data lowered high confidence gained from training data
Results: Navigation, Prevention, Closure for Fine-Grained Changes

TABLE 4
Results for Fine Granularity (R = recall; P = precision; Fb = feedback; L = likelihood)

<table>
<thead>
<tr>
<th>Project</th>
<th>Navigation</th>
<th></th>
<th>Prevention</th>
<th></th>
<th>Closure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fb</td>
<td>R_M</td>
<td>P_M</td>
<td>L_3</td>
<td>Fb</td>
</tr>
<tr>
<td>ECLIPSE</td>
<td>0.64</td>
<td>0.34</td>
<td>0.30</td>
<td>0.57</td>
<td>0.03</td>
</tr>
<tr>
<td>GCC</td>
<td>0.63</td>
<td>0.45</td>
<td>0.31</td>
<td>0.91</td>
<td>0.08</td>
</tr>
<tr>
<td>GIMP</td>
<td>0.60</td>
<td>0.35</td>
<td>0.30</td>
<td>0.92</td>
<td>0.03</td>
</tr>
<tr>
<td>JBOSS</td>
<td>0.59</td>
<td>0.36</td>
<td>0.31</td>
<td>0.62</td>
<td>0.02</td>
</tr>
<tr>
<td>JEDIT</td>
<td>0.74</td>
<td>0.21</td>
<td>0.31</td>
<td>0.86</td>
<td>0.01</td>
</tr>
<tr>
<td>KOFFICE</td>
<td>0.65</td>
<td>0.24</td>
<td>0.23</td>
<td>0.54</td>
<td>0.01</td>
</tr>
<tr>
<td>POSTGRES</td>
<td>0.76</td>
<td>0.29</td>
<td>0.29</td>
<td>0.65</td>
<td>0.02</td>
</tr>
<tr>
<td>PYTHON</td>
<td>0.66</td>
<td>0.37</td>
<td>0.27</td>
<td>0.54</td>
<td>0.02</td>
</tr>
<tr>
<td>Average</td>
<td>0.66</td>
<td>0.33</td>
<td>0.29</td>
<td>0.70</td>
<td>0.03</td>
</tr>
</tbody>
</table>

- **Navigation**: moderate feedback, low recall and precision, but high likelihood
- **Prevention**: covers a majority of missing changes precisely
- **Closure**: does not suggest a change when nothing needs to be changed
- **Better results for mature systems**
Results: Granularity, Maintenance, Multiple Dimensions

→ Coarse granularity
  - Mine and suggest changes to files only
  - Suggestions more frequent, but less useful
  - +16% feedback, +10% recall
  - In the future, start out with vague suggestions and refine it through fine-grained changes

→ Maintenance
  - ROSE can better predict “alter” changes than “add_to” and “delete_from” changes
  - +8% feedback, +19% recall (25% to 44%)

→ Multiple Dimensions
  - Abstracting fine-grained element additions/deletions to “add_to” and “delete_from”
  - +6% recall
**Results: History**

- **Eclipse**: shortly before each release, features are frozen and bug fixes are performed. ROSE performs well here (maintenance changes).
- **GCC**: predictive power saturates around 2.95.3. History no longer useful.
- **Eclipse is a relative new project**
Results: Recent Changes

- Recent changes more likely to be useful
  - E.g. method renaming possibly makes previous changes irrelevant
- Modified rule calculation using 2 alternatives
  - 1. Mine from last 6 months only
  - 2. Assign weight to age of change (0 oldest, 1 newest)
- Eclipse
  - +3% recall, up to +11% possible with ideal ranking
- No effect for GCC (too mature)

Fig. 9. Different ranking techniques for ECLIPSE (2003-03-28 to 2004-09-14).
Overview

→ Relevance to the course
→ An Overview of Mining Software Repositories
→ Instance 1: ROSE for change prediction
   ⇨ Explanation
   ⇨ Critique
   ⇨ Analysis
→ Instance 2: Refining static analysis for return value checking
   ⇨ Explanation
   ⇨ Critique
   ⇨ Analysis
→ State of the Art
Critique of the ROSE paper: The Good

→ About the paper
  ➤ Well-structured
  ➤ Well-explained for most part

→ About the content of the paper
  ➤ A simple, elegant technique, based on counting, that scales
  ➤ A novel idea to use change history for change guidance
  ➤ A thorough evaluation achieved through automation
Critique of the ROSE paper: The Bad

- About the paper
  - Evaluation section could be more concise
  - Interest tapers off in the middle of the paper

- About the content of the paper
  - Lacks technical merit (or seems to, ioho)
  - Results aren’t as encouraging as we expected
  - Seems to brush off some very important issues
Overview

- Relevance to the course
- An Overview of Mining Software Repositories
- Instance 1: ROSE for change prediction
  - Explanation
  - Critique
  - Analysis
- Instance 2: Refining static analysis for return value checking
  - Explanation
  - Critique
  - Analysis
- State of the Art
Question 1.1: Strengths of ROSE

What is the killer application of ROSE? I.e. in what situations would you be compelled to use ROSE?
Answer 1.1: Strengths of ROSE

- For new bug fixers in OSS development with commit access
  - CVS repository widely accessible
  - ROSE is most effective for maintenance activities
  - Could reduce communication time with core developers
  - No learning curve to use ROSE

- For core and experienced bug fixers
  - Annoying
Question 1.2: Effectiveness

What are some of the problems you see with the effectiveness of the technique?

What are some threats to validity, i.e., influences limiting our ability to draw conclusions from the study’s data?
Answer 1.2: Effectiveness

➔ Paper says...
  ➢ Sample is not a representative set
  ➢ Order of changes within a transaction
    ➢ May affect navigation
  ➢ Quality of transactions
    ➢ More good than bad to learn from history
  ➢ Predictive power as a measure of usefulness
    ➢ Software development is a complex activity – there may be other ways of being guided along related changes (e.g. through compilation)

➔ We add...
  ➢ Quality of transactions do matter
  ➢ Isn't an extra step, i.e. comparing the results against alternative “change guidance” techniques, required?
    ➢ Measuring relative productivity to the best technique available
Question 1.3: Quality of Transactions

→ History is not always a good indicator of the past. E.g. Transactions may contain unrelated changes due to poor understanding at the time, poor committers, eager committers sneaking in changes, and etc.

How could ROSE be improved to remember the good times?
Answer 1.3: Quality of Transactions

➔ Assign weights to transactions at commit time
  ➸ Self-confidence ("Not sure if this is the best design, but I'll fix if later")
  ➸ Test results (especially compatible with XP)
  ➸ Code inspection (expert opinion likely to be correct)
  ➸ Developer status (changing someone else's file evaluated lower, lower-ranked and developers with lower-credibility evaluated lower)
  ➸ Preciseness of problem in commit report ("Bluescreen of death when running more than 10 processes", as opposed to "Initial checkin")

➔ Must have support for and willingness to create structured commit report
Question 1.4: Removing annoyingness

ROSE can suggest changes that are obvious. For example, when an interface is changed, changes to implementations are top-ranked suggestions.

How could ROSE be improved to eliminate this annoyingness?
Answer 1.4: Removing annoyingness

➔ Use static analysis to detect “obvious” couplings and skip these from the suggestions

◫ E.g. For alter(X) => alter(Y), if performing alter(X) breaks Y, then don’t suggest it
Question 1.5: ROSE and static analysis

How can static analysis improve ROSE? Conversely, how can ROSE improve static analysis?
Answer 1.5: ROSE and static analysis

→ Static analysis improving ROSE
  • Elevate confidence for elements “non-trivially” connected through program analysis

```java
public class FeatureNameDialog extends Dialog {
    protected static String[] columnNames =
            new String["id", "name", "check", "depends-on"];

public class Delta implements Visitable {
    // color prefix
    public static final String COLOR_PREFIX = "Feature";

    public class FeatureCellModifier implements ICellModifier {
        public void modify(Object element, 
            if (element instanceof Item) {
                element = ((Item) element);
        }
        feature f = (Feature) element;
        if (columnNames[1] == property)
            featureNames.setFeatureName;
        if (columnNames[2] == property)
            featureNames.setFeatureVis;
        if (columnNames[3] == property)
            featureNames.setRequiredFe;
    }
}
```

• Make “interesting” suggestions [Annie Ying et al. TSE 2004]

→ ROSE (or association rule mining) improving static analysis
  • Open question – maybe it can be used to restrict the search space for difficult program analysis (like inter-procedural reachability and alias analysis)
Question 1.6: Other evolution activities

ROSE uses association rule mining to provide change guidance. What other evolution activities can benefit from association rule mining?
Answer 1.6: Other evolution activities

- **Refactoring**: items related by changes are refactored into a module
  - *E.g.* A feature module is a collection of transformations across the system
  
  ![Diagram showing refactoring example]

- **Reverse engineering**: related changes are lifted into abstractions and relations (e.g. architecture diagram)

  ![Diagram showing reverse engineering example]
Overview

- Relevance to the course
- An Overview of Mining Software Repositories
- Instance 1: ROSE for change prediction
  - Explanation
  - Critique
  - Analysis
- Instance 2: Refining static analysis for return value checking
  - Explanation
  - Critique
  - Analysis
- State of the Art
Example

* Old Version:
  result2 = 3 + get_env()->value;
  ...

* Relatively Newer Version:
  [Same file, same function call]

  Env* e = get_env();
  if (e != NULL) {
    result2 = 3 + e->value;
  }
  ...

* Current Version:

  result1 = 3 + get_env()->value;
  ...

  Env* e = get_env();
  if (e != NULL) {
    result2 = 3 + e->value;
  }
  ...

  result3 = 3 + get_env()->value;
  ...

Static analysis would not give any warnings!
Paper Overview

→ Fact:
  - Source code repositories provide a detailed log of the source code evolution

→ Aim:
  - Refining the static analysis techniques using the data describing bug fixes

→ Claim:
  - Efforts to build bug-finding tools should start from an analysis of the occurrence of bugs in *real software*
Technique Overview

CVS Repository → Data Processing → MySQL Database

Preliminary Inspection of Historical Data

Static Analysis Tool
Function Return Value Checker
Mining the Source Code Repository
Ranking the Results

Current Version
Historical Data

Function Return Value Checker – Static Analysis

Results
Preliminary Inspection of Historical Data

- Review the historical data for *usable* information
- Categorize the types of bugs that had been fixed in the history

Results of the manual inspection:

- Automatic mining is possible
- Ignore commit messages and bug reports
- Good candidates for static analysis:
  - NULL pointer checks
  - Function return value checks (this kind of bug fixed many times in the past)
Function Return Value Checker

Determines if, when a function returns a value, that value is *tested* before being *used*.

- “using” a return value:
  - Passing it as an argument
  - Using it as part of a calculation
  - Dereferencing the value if it is a pointer
  - Overwriting the value before being tested

- “testing” a return value:
  - Some control flow decision relies on the value
Function Return Value Checker

→ The need for a return value checker:
  ¬ Return value of a function may be either valid data OR an error code

→ Dataflow analysis on the variable holding the return value until the variable is used or tested
  ¬ Identify the variable
  ¬ Determine the next use of that variable

→ The output of the return value checker is a list of warnings
Mining the Source Code Repository

- Storing Revision Histories in a Database
- Identifying CVS transactions (!= CVS commits)
- Obtaining Results
  - Try to determine a bug (concerned type) with its fix
  - Look for the source code changes that:
    - Takes a function return value (which previously not tested) &
      Adds a test of the return value
    - Use return value checker over both versions to find these changes
- Storing Results in a Database
Ranking the Results

- Two separate components to rank the output of the return value checker:
  - **Historical context information**
    - Data mined from the source code repository
    - List of functions that are involved in a potential bug fix
  - **Contemporary context information**
    - Data mined from the current version of the software
    - How often each function is being tested before its return value is used
Ranking the Results

- Functions are ranked rather than the warnings
- Ranking is done in two parts:
  - 1st step: group into two:
    - Those that are involved in a potential bug fix (ranks higher)
    - Those that are not
  - 2nd step: within each group:
    - How often the return value is checked before used? (very often ranks higher)
Computational Cost

- Almost 50,000 revisions stored in over 20,000 CVS transactions

- Checking one CVS transaction:
  - From extracting the source tree
  - To store the results in the database
  - Takes roughly 4 minutes

- 64 node Linux Cluster managed by the Portable Batch Scheduler (PBS) is used to mine all transactions
  - 20 processors at a time
Technique Summary

CVS Repository → Data Processing → MySQL Database

- Current Version
- Historical Data

Function Return Value Checker – Static Analysis → Results
Evaluation Overview

- Projects used: Apache Web Server & Wine
- Outputs of the static analysis tool are ranked in two ways in order to evaluate the technique
  - Naïve Ranking
    - Only contemporary context is used, purely static
    - Functions whose return value is tested before being used more than half of the time
  - HistoryAware Ranking
    - Result of the proposed technique
    - Two step ranking
Results

TABLE 1
Warnings and Likely Bugs for the Apache Web Server

<table>
<thead>
<tr>
<th></th>
<th>Warnings</th>
<th>Likely Bugs</th>
<th>False Positive Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVS bug fix flagged functions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function checked &gt; 50% of the time</td>
<td>121</td>
<td>38</td>
<td>68%</td>
</tr>
<tr>
<td>Function checked &lt;= 50% of the time</td>
<td>163</td>
<td>63</td>
<td>61%</td>
</tr>
<tr>
<td>Subtotal</td>
<td>284</td>
<td>101</td>
<td>64%</td>
</tr>
<tr>
<td>Non-CVS bug fix flagged functions</td>
<td>283</td>
<td>70</td>
<td>75%</td>
</tr>
<tr>
<td>Total</td>
<td>567</td>
<td>171</td>
<td>70%</td>
</tr>
</tbody>
</table>

TABLE 3
Warnings and Likely Bugs for Wine

<table>
<thead>
<tr>
<th></th>
<th>Warnings</th>
<th>Likely Bugs</th>
<th>False Positive Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVS bug fix flagged functions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function checked &gt; 50% of the time</td>
<td>329</td>
<td>106</td>
<td>68%</td>
</tr>
<tr>
<td>Function checked &lt;= 50% of the time</td>
<td>449</td>
<td>154</td>
<td>66%</td>
</tr>
<tr>
<td>Subtotal</td>
<td>778</td>
<td>260</td>
<td>67%</td>
</tr>
<tr>
<td>Non-CVS bug fix flagged functions</td>
<td>1537</td>
<td>285</td>
<td>81%</td>
</tr>
<tr>
<td>Total</td>
<td>2315</td>
<td>545</td>
<td>76%</td>
</tr>
</tbody>
</table>
Analysis of Ranked Functions

- Top three (Wine) are all system supplied string manipulation functions: strrchr(), strchr(), strstr()

- Functions returning a pointer to an already allocated data structure
- Allocating memory functions:
- Functions performing complex logic
- Functions manipulating a data structure

```c
if ((ap_server_pre_read_config->nelts || ap_server_post_read_config->nelts) && !strcmp(fname, ap_server_root_relative(p, SERVER_CONFIG_FILE))}{
```
Effectiveness of the Technique

Apache Web Server

Wine

WineApache Web Server

Functions checked > 50% of the time in contemporary context

CVS Bug Fix Flagged Functions

Warnings: 163
Bugs: 63
False Positive Rate: 0.61

Warnings: 121
Bugs: 38
False Positive Rate: 0.68

Warnings: 283
Bugs: 70
False Positive Rate: 0.75

Warnings: 449
Bugs: 103
False Positive Rate: 0.77

Warnings: 329
Bugs: 103
False Positive Rate: 0.68

Warnings: 1537
Bugs: 285
False Positive Rate: 0.82
Precision

* Relevant suggestions

Apache

Wine
Recall

* Expectation coverage

Apache

Wine
Cumulative False Positive Rate Overview

- **X-axis:**
  - Contemporary context information

- CVS flagged warnings have a lower false positive rate

- False positive rate for the HistoryAware ranking remains over 50 percent

- More dataflow analysis and a deeper understanding of the context of function calls may help to improve the false positive rate
Cumulative False Positive Rate

False Positive Rate Vs Contemporary Context

- Naive Ranking
- HistoryAware Ranking - CVS Flagged
- HistoryAware Ranking - NonCVS Flagged

Apache

Wine
Overview

→ Relevance to the course
→ An Overview of Mining Software Repositories
→ Instance 1: ROSE for change prediction
  ↗ Explanation
  ↗ Critique
  ↗ Analysis
→ Instance 2: Refining static analysis for return value checking
  ↗ Explanation
  ↗ Critique
  ↗ Analysis
→ State of the Art
Critique of the static analysis refinement paper: The Good

➔ About the paper
   ➔ Well-written
   ➔ Easy to understand

➔ About the content of the paper
   ➔ Technique is explained step-by-step, clearly
   ➔ Easy to grasp
Critique of the static analysis refinement paper: The Bad

➔ About the paper
   ✰ Calculation mistakes
   ✰ Not so well-organized

➔ About the content of the paper
   ✰ Manual inspections
   ✰ Minor improvements
   ✰ Computational cost
Overview

→ Relevance to the course
→ An Overview of Mining Software Repositories
→ Instance 1: ROSE for change prediction
   ⇧ Explanation
   ⇧ Critique
   ⇧ Analysis
→ Instance 2: Refining static analysis for return value checking
   ⇧ Explanation
   ⇧ Critique
   ⇧ Analysis
→ State of the Art
Question 2.1: Killer Application

What is the killer application of the static analysis refinement for function return value checking? When will you be compelled to use this?
Answer 2.1: Killer Application

→ **Systems**
   - Large, safety-critical systems

→ **People**
   - **System testers**
     - Job is to ensure correctness of the system
   - **Devoted developers**
     - The effort required to run the refined static analysis and manually check every result is not justified by others
     - Probably most knowledgeable about bug types
Question 2.2: Effectiveness

What are some of the problems you see with the effectiveness of the static analysis refinement approach?
Answer 2.2: Effectiveness

- Bug-specific: You have to choose a known (and frequent) bug to tackle and mine the repository and create a static analysis for that
  - OK to start off for a few bugs, but after, should try to generalize afterwards
- Too imprecise: 68% false positives
- Comparison to looking at just the snapshot
  - Combining snapshot and history: 9% less (Apache) and 17% less (Wine) false positives
  - Not a huge improvement...
- Computational cost
Question 2.3: Evaluation

- The evaluation involved manually checking for correctness of the suggestions. This was not necessary for ROSE, since the checking between expected and actual was done automatically.

  Couldn't the checking for the static analysis refinement be done automatically too?
Answer 2.3: Evaluation

To automate evaluation,

- The expected program must be “correct” - i.e. free of bugs of the chosen type
- There is no way to automatically know that a chosen program will be correct
  - No such tool - we’re trying to build one!
  - Someone must check that the program is correct

So (probably) not possible
Question 2.4: Alternatives

What is an alternative, and possibly better, way of addressing this particular bug, the function return bug?

Hint: some functions need to their values checked, but other functions do not.
Answer 2.4: Alternatives

➔ Use “obligations” (Perry, SIGSOFT)

➔ Annotate a function that requires an if-check with an obligation

```java
/**
 * @Obligation(Check-Before-Use)
 */
public Env *get_env()
{
    ...
}
```

➔ Editor/compiler checks fulfillment of the obligation

```java
void method1()
{
    result = 3 + get_env()->value; // ERROR
    printf(result); // OK
}

void method2()
{
    Env env = get_env(); // OK
    if(env != null)
    {
        result = 3 + env->value;
    }
```
Overview

- Relevance to the course
- An Overview of Mining Software Repositories
- Instance 1: ROSE for change prediction
  - Explanation
  - Critique
  - Analysis
- Instance 2: Refining static analysis for return value checking
  - Explanation
  - Critique
  - Analysis
- State of the Art
State of the Art in MSR

- Change prediction
  - Combining single-version (dependency analysis) vs. evolutionary dependencies (MSR) (Kagdi et al., MSR 2007)

- Refining static analysis
  - Multi-version program analyses (M. Kim et al., MSR 2006)
  - Prioritizing warning categories based on warning lifetime (S. Kim et al., MSR 2007)

- Collaboration/communication
  - Source code that talks (Ying et al., MSR 2005)

- Extracting CVS information
  - TA-RE: exchange language to describe software repository data (S. Kim et al., MSR 2006)
  - EvoOnt and iSPARQL: exchange language based on OWL and SPARQL (MSR 2007)