Identifying Bug Signatures Using Discriminative Graph Mining

Hong Cheng¹, David Lo², Yang Zhou¹, Xiaoyin Wang³, and Xifeng Yan⁴

¹Chinese University of Hong Kong
²Singapore Management University
³Peking University
⁴University of California at Santa Barbara
Automated Debugging

- Bugs part of day-to-day software development
- Bugs caused the loss of much resources
  - NIST report 2002
  - 59.5 billion dollars/annum
- Much time is spent on debugging
  - Need support for debugging activities
  - Automate debugging process
- Problem description
  - Given labeled correct and faulty execution traces
  - Make debugging an easier task to do
Bug Localization and Signature Identification

- **Bug localization**
  - Pinpointing a *single statement or location* which is likely to contain bugs
  - Does not produce the bug context

- **Bug signature mining** [Hsu et al., ASE’08]
  - Provides the *context* where a bug occurs
  - Does not assume “perfect bug understanding”
  - In the form of *sequences of program elements*
  - Occur when the bug is manifested
Outline

- Motivation: Bug Localization and Bug Signature
- Pioneer Work on Bug Signature Mining
- Identifying Bug Signatures Using Discriminative Graph Mining
- Experimental Study
- Related Work
- Conclusions and Future Work
Pioneer Work on Bug Signature Identification

- **RAPID** [Hsu et al., ASE’08]
  - Identify relevant suspicious program elements via Tarantula

\[
suspiciousness(s) = \frac{\frac{\text{failed}(s)}{\text{total failed}}}{\frac{\text{passed}(s)}{\text{total passed}}} + \frac{\text{failed}(s)}{\text{total failed}}
\]

- Compute the longest common subsequences that appear in all faulty executions with a sequence mining tool **BIDE** [Wang and Han, ICDE’04]
- Sort returned signatures by length
- Able to identify a bug involving path-dependent fault
Software Behavior Graphs

- Model software executions as behavior graphs
  - Node: method or basic block
  - Edge: call or transition (basic block/method) or return
  - Two levels of granularities: method and basic block

- Represent signatures as discriminating subgraphs

- Advantages of graph over sequence representation
  - Compactness: loops $\rightarrow$ mining scalability
  - Expressiveness: partial order and total order
Example: Software Behavior Graphs

Two executions from Mozilla Rhino with a bug of number 194364

Solid edge: function call
Dashed edge: function transition
Bug Signature: Discriminative Sub-Graph

- Given two sets of graphs: correct and failing
- Find the most discriminative subgraph

Information gain: \( IG(c|g) = H(c) - H(c|g) \)
- Commonly used in data mining/machine learning
- Capacity in distinguishing instances from different classes
- Correct vs. Failing

Meaning:
- As frequency difference of a subgraph \( g \) in faulty and correct executions increases
- The higher is the information gain of \( g \)

Let \( F \) be the objective function (i.e., information gain), compute:
\[
\arg \max_g F(g)
\]
Bug Signature: Discriminative Sub-Graph

- The discriminative subgraph mined from behavior graphs contrasts the program flow of correct and failing executions and provides context for understanding the bug.

- Differences with RAPID:
  - Not only element-level suspiciousness, signature-level suspiciousness/discriminative-ness
  - Does not restrict that the signature must hold across all failing executions
  - Sort by level of suspiciousness
System Framework

STEP 1
Build Behavior Graphs

STEP 2
Remove Non-Suspicious Edges

STEP 3
Mine Top-K Discriminative Graphs

Bug Signatures
System Framework (2)

- **Step 1**
  - Trace is “coiled” to form behavior graphs
  - Based on transitions, call, and return relationship
  - Granularity: method calls, basic blocks

- **Step 2**
  - Filter off non-suspicious edges
  - Similar to Tarantula suspiciousness
  - Focus on relationship between blocks/calls

\[
\text{suspe}_{\text{edg}} = \frac{\text{failed}(\text{edg})}{\text{passed}(\text{edg})} > \frac{\text{total failed}}{\text{total passed}}
\]

- **Step 3**
  - Mine top-k discriminating graphs
  - Distinguishes buggy from correct executions
An Example

1: void replaceFirstOccurrence (char arr [], int len, char cx, char cy, char cz) {

    int i;
2:        for (i=0;i<len;i++) {
3:            if (arr[i]==cx){
4:                arr[i] = cz;
5:                // a bug, should be a break;
6:            }
7:            if (arr[i]==cy){
8:                arr[i] = cz;
9:                // a bug, should be a break;
10:           }
11:       }
11:    }

<table>
<thead>
<tr>
<th>No</th>
<th>arr</th>
<th>sx</th>
<th>sy</th>
<th>sz</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{a, b}</td>
<td>a</td>
<td>g</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>{a, b}</td>
<td>g</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>{a, g}</td>
<td>a</td>
<td>g</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>{a, g}</td>
<td>g</td>
<td>a</td>
<td>1</td>
</tr>
</tbody>
</table>

Four test cases

<table>
<thead>
<tr>
<th>No</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>h1, 2, 3, 4, 7, 10, 2, 3, 7, 10, 11i</td>
</tr>
<tr>
<td>2</td>
<td>h1, 2, 3, 7, 10, 2, 3, 7, 8, 10, 11i</td>
</tr>
<tr>
<td>3</td>
<td>h1, 2, 3, [4], 7, 10, 2, 3, 7, [8], 10, 11i</td>
</tr>
<tr>
<td>4</td>
<td>h1, 2, 3, 7, [8], 10, 2, 3, [4], 7, 10, 11i</td>
</tr>
</tbody>
</table>

Generated traces
Behavior Graphs for Trace 1, 2, 3 & 4

Normal

Buggy
An Example (3)

Figure 4: Pre-processed graphs for the four execution traces. All edges are labeled as trans.

Figure 5: The discriminative subgraph. All edges are labeled as trans.
Challenges in Graph Mining: Search Space Explosion

- If a graph is frequent, all its subgraphs are frequent - the Apriori property

- An n-edge frequent graph may have up to $2^n$ subgraphs which are also frequent

- Among 423 chemical compounds which are confirmed to be active in an AIDS antiviral screen dataset, there are around 1,000,000 frequent subgraphs if the minimum support is 5%
Traditional Frequent Graph Mining Framework

1. Computational bottleneck: millions, even billions of patterns

2. No guarantee of quality
Leap Search for Discriminative Graph Mining

- Yan et al. proposed a new leap search mining paradigm in SIGMOD’08
  - Core idea: structural proximity for search space pruning

- Directly outputs the most discriminative subgraph, highly efficient!
Core Idea: Structural Similarity

Structural similarity $\rightarrow$ Significance similarity

$g \sim g' \Rightarrow F(g) \sim F(g')$

Mine one branch and skip the other similar branch!
Structural Leap Search Criterion

Skip $g'$ subtree if

$$\frac{2\Delta_+(g, g')}{\sup_+(g) + \sup_+(g')} \leq \sigma$$

$$\frac{2\Delta_-(g, g')}{\sup_-(g) + \sup_-(g')} \leq \sigma$$

$\sigma$: tolerance of frequency dissimilarity

g : a discovered graph

g' : a sibling of g
Extending LEAP to Top-K LEAP

- LEAP returns the single most discriminative subgraph from the dataset

- A ranked list of k most discriminative subgraphs is more informative than the single best one

Top-K LEAP idea
- The LEAP procedure is called for k times
- Checking partial result in the process
- Producing k most discriminative subgraphs
Experimental Evaluation

- **Datasets**
  - Siemens datasets: All 7 programs, all versions

- **Methods**
  - RAPID [Hsu et al., ASE'08]
  - Top-K LEAP: our method

- **Metrics**
  - Recall and Precision from top-k returned signatures
  - Recall = proportion of the bugs that could be found by the bug signatures
  - Precision = proportion of the returned results that highlight the bug
  - Distance-based metric to exact bug location penalize the bug context
## Experimental Results (Top 5)

<table>
<thead>
<tr>
<th>Prog.</th>
<th>RAPID</th>
<th></th>
<th></th>
<th></th>
<th>Top-K LEAP</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tcas</td>
<td>82.9</td>
<td>82.9</td>
<td>8.0</td>
<td></td>
<td>85.9</td>
<td>95.1</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>ptok</td>
<td>71.4</td>
<td>71.4</td>
<td>4.0</td>
<td></td>
<td>85.7</td>
<td>100</td>
<td>4.3</td>
<td></td>
</tr>
<tr>
<td>ptok2</td>
<td>20.0</td>
<td>20.0</td>
<td>2.7</td>
<td></td>
<td>36.0</td>
<td>60.0</td>
<td>2.9</td>
<td></td>
</tr>
<tr>
<td>sched</td>
<td>33.3</td>
<td>33.3</td>
<td>2.3</td>
<td></td>
<td>54.1</td>
<td>66.7</td>
<td>3.6</td>
<td></td>
</tr>
<tr>
<td>sched2</td>
<td>0.0</td>
<td>0.0</td>
<td>N/A</td>
<td></td>
<td>24.2</td>
<td>30.0</td>
<td>2.2</td>
<td></td>
</tr>
<tr>
<td>tinfo</td>
<td>21.7</td>
<td>21.7</td>
<td>2.5</td>
<td></td>
<td>69.6</td>
<td>78.3</td>
<td>2.4</td>
<td></td>
</tr>
<tr>
<td>rep</td>
<td>53.1</td>
<td>53.1</td>
<td>5.1</td>
<td></td>
<td>54.4</td>
<td>81.3</td>
<td>2.9</td>
<td></td>
</tr>
<tr>
<td>Avg.</td>
<td>40.4</td>
<td>40.4</td>
<td>4.1</td>
<td></td>
<td>58.5</td>
<td>73.0</td>
<td>3.3</td>
<td></td>
</tr>
</tbody>
</table>

Result - Method Level
## Experimental Results (Top 5)

<table>
<thead>
<tr>
<th>Prog.</th>
<th>RAPID Pre.</th>
<th>RAPID Rec.</th>
<th>RAPID Size</th>
<th>Top-K LEAP Pre.</th>
<th>Top-K LEAP Rec.</th>
<th>Top-K LEAP Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcas</td>
<td>90.2</td>
<td>90.2</td>
<td>11.3</td>
<td>88.3</td>
<td>100</td>
<td>3.8</td>
</tr>
<tr>
<td>ptok</td>
<td>100</td>
<td>100</td>
<td>9.7</td>
<td>85.7</td>
<td>100</td>
<td>4.8</td>
</tr>
<tr>
<td>ptok2</td>
<td>65.0</td>
<td>70.0</td>
<td>7.2</td>
<td>74.0</td>
<td>100</td>
<td>3.4</td>
</tr>
<tr>
<td>sched</td>
<td>75.0</td>
<td>77.8</td>
<td>5.1</td>
<td>86.7</td>
<td>88.9</td>
<td>3.2</td>
</tr>
<tr>
<td>sched2</td>
<td>40.0</td>
<td>40.0</td>
<td>2.6</td>
<td>52.0</td>
<td>80.0</td>
<td>2.8</td>
</tr>
<tr>
<td>tinfo</td>
<td>56.5</td>
<td>56.5</td>
<td>15.4</td>
<td>55.0</td>
<td>87.0</td>
<td>3.6</td>
</tr>
<tr>
<td>rep</td>
<td>80.5</td>
<td>81.3</td>
<td>20.7</td>
<td>78.1</td>
<td>81.3</td>
<td>4.9</td>
</tr>
<tr>
<td>Avg.</td>
<td>72.5</td>
<td>73.7</td>
<td>10.3</td>
<td>74.3</td>
<td>91.0</td>
<td>3.8</td>
</tr>
</tbody>
</table>

Result – Basic Block Level
Experimental Results (2) - Schedule

Precision Recall

Top-K Signatures Analyzed

Precision

Recall
Efficiency Test

- Top-K LEAP finishes mining on every dataset between 1 and 258 seconds

- RAPID cannot finish running on several datasets in hours
  - Version 6 of replace dataset, basic block level
  - Version 10 of print_tokens2, basic block level
Experience (1)

```c
1 upgrade_process_prio(prio, ratio){
   ...
2       n = (int) (count*ratio+1);
3       if(ratio == 1.0) n--; //added code
4       proc = find_nth(src_queue, n);
   ...
}

5 unblock_process(prio, ratio){
   ...
6       n = (int) (count*ratio +1);
7       if(ratio == 1.0) n--; // added code
8       proc = find_nth(src_queue, n);
   ...
}
```

Version 7 of schedule
Top-K LEAP finds the bug, while RAPID fails
Experience (2)

```c
1 InfoTbl( r, c, f, pdf ){
2     rdf = r-1; cdf = c-1;  //->basic block 1
3     if ( rdf == 0 || cdf == 0 )  //bug
4         info = -3.0;  //->basic block 2
5         goto ret3;
6     }
7     N = 0.0;  //->basic block 3
8     for ( i = 0; i < r; ++i ){
9         //->basic block 4,
10     }
11     //->basic block 5
12     if ( N <= 0.0 ){...}
13 }
```

if ( rdf <=0 || cdf <= 0)

For rdf<0, cdf<0

bb1→bb3→bb5

Our method finds a graph connecting block 3 with block 5 with a transition edge

Version 18 of tot_info
Threat to Validity

- Human error during the labeling process
  - Human is the best judge to decide whether a signature is relevant or not.

- Only small programs
  - Scalability on larger programs

- Only c programs
  - Concept of control flow is universal
Related Work

- **Bug Signature Mining**: RAPID [Hsu et al., ASE’08]
- **Bug Predictors to Faulty CF Path** [Jiang et al., ASE’07]
  - Clustering similar bug predictors and inferring approximate path connecting similar predictors in CFG.
  - Our work: finding combination of bug predictors that are discriminative. Result guaranteed to be feasible paths.

- **Bug Localization Methods**
  - Tarantula [Jones and Harrold, ASE’05], WHITHER [Renieris and Reiss, ASE’03], Delta Debugging [Zeller and Hildebrandt, TSE’02], AskIgor [Cleve and Zeller, ICSE’05], Predicate evaluation [Liblit et al., PLDI’03, PLDI’05], Sober [Liu et al., FSE’05], etc.
Related Work on Graph Mining

- Early work
  - SUBDUE [Holder et al., KDD'94], WARMR [Dehaspe et al., KDD'98]

- Apriori-based approach
  - AGM [Inokuchi et al., PKDD'00]
  - FSG [Kuramochi and Karypis, ICDM'01]

- Pattern-growth approach - state-of-the-art
  - gSpan [Yan and Han, ICDM'02]
  - MoFa [Borgelt and Berthold, ICDM'02]
  - FFSM [Huan et al., ICDM'03]
  - Gaston [Nijssen and Kok, KDD'04]
Conclusions

- A discriminative graph mining approach to identify bug signatures
  - Compactness, Expressiveness, Efficiency

- Experimental results on Siemens datasets
  - On average, 18.1% higher precision, 32.6% higher recall (method level)
  - On average, 1.8% higher precision, 17.3% higher recall (basic block level)
  - Average signature size of 3.3 nodes (vs. 4.1) (method level) or 3.8 nodes (vs 10.3) (basic block level)
  - Mining at basic block level is more accurate than method level - (74.3%, 91%) vs (58.5%, 73%)
Future Extensions

- **Mine minimal subgraph patterns**
  - Current patterns may contain irrelevant nodes and edges for the bug

- **Enrich software behavior graph representation**
  - Currently only captures program flow semantics
  - May attach additional information to nodes and edges such as program parameters and return values
Thank you for your attention

Questions? Comments? Advice?

hcheng@se.cuhk.edu.hk  davidlo@smu.edu.sg
Bug Signature: Discriminative Sub-Graph

- Given graphs labeled as correct or failing
- Find the most discriminative subgraph
- Information gain: \( IG(c|g) = H(c) - H(c|g) \)

\[
H(c) = \sum_{i \in \{0,1\}} p(c_i) \log p(c_i)
\]

\[
H(c|g) = \sum_{i \in \{0,1\}} p(g_i) \sum_{j \in \{0,1\}} p(c_{j|g_i}) \log p(c_{j|g_i})
\]

- \( c \) - class label, \( g \) - subgraph
- \( p(c_1) \) - proportion of faulty traces
- \( p(g_1) \) - prop. of traces containing the sub-graph
- \( p(c_1|g_1) \) - proportion of the traces that are faulty given that the graph is exhibited in the trace.
Other Related Work

- **Chao et al.** Mining Behavior Graphs [SDM’05]
  - Their work detect if a trace is erroneous or not. We find the discriminating signature from two sets of traces.
  - They mine for all closed patterns and then use them as features for the classification of two sets of traces. Our approach directly mine for top-k discriminative graphs.

- **Chang et al.** Neglected Conditions [ISSTA’07]
  - Their work mine patterns from code rather than traces.
  - Used for bug finding rather than for finding bug signatures.
  - They find frequent graphs, while we find discriminating graphs.
Other Related Work

0 Christodorescu et al. Mining Specifications of Malicious Behaviors [FSE’07]

- Detect only if a graph appear in malware but never in normal.

- We detect **discriminating features**, including cases where a graph pattern appear 500 times in faulty, 1 time in normal.

- At times we only have **partial information** unless we model everything about software systems. Due to this often we do not have a perfectly discriminating feature.