Data Science *elevating* Software Engineering
Software Engineering *elevating* Data Science

Miryung Kim
University of California, Los Angeles
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. 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**: implemented using the Eclipse Zest Visualization Toolkit. It visualizes 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 a selected 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 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 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 execution:

Data Science *elevating* Software Engineering

**Software Refactoring**
- Refactoring Field Study
- Quantifying Refactoring Cost and Benefits
- Impact on Regression Testing
- Role of API Refactoring

**API Evolution**
- Role of API Refactoring
- API Stability

**Empirical Studies of Software Changes**

**Code Redundancy**
- Clone genealogy
- Copy and paste practices
- Long lived clones
- Software forking and code porting

**Software Patches**
- Supplementary patches
- Omission errors
Data Science *elevating* Software Engineering

**Empirical Studies of Software Changes**

- Software Refactoring
  - Refactoring Field Study
  - Quantifying Refactoring Cost and Benefits
  - Impact on Regression Testing
  - Role of API Refactoring

- API Evolution
  - Role of API Refactoring
  - API Stability

**Automated and Interactive Software Dev Tools**

- Code Redundancy
  - Clone genealogy
  - Copy and paste practices
  - Long lived clones
  - Software forking and code porting

- Software Patches
  - Supplementary patches
  - Omission errors

- Logical Program Differencing
  - LSdiff
  - Vdiff for VHDL

- Refactoring Reconstruction
  - Reffinder

- API Usage Adaptation
  - LibSync6
  - AURA
  - API Matching

- Interactive Code Review
  - Critics

- Transplantation and Test Reuse
  - Grafter

- Clone Removal Refactoring
  - RASE

**Tools and Techniques**

- API Evolution
- Code Redundancy
- Software Patches
- Logical Program Differencing
- Refactoring Reconstruction
- API Usage Adaptation
- Interactive Code Review
- Transplantation and Test Reuse
- Clone Removal Refactoring
Data Science *elevating* Software Engineering

**Empirical Studies of Software Changes**

- **Code Redundancy**
  - Clone genealogy
  - Copy and paste practices
  - Long lived clones
  - Software forking and code porting

- **Software Patches**
  - Supplementary patches
  - Omission errors

- **Logical Program Differencing**
  - LSdiff
  - Vdiff for VHDL

- **Refactoring Reconstruction**
  - Reffinder

- **API Usage Adaptation**
  - LibSync6
  - AURA
  - API Matching

- **Interactive Code Review**
  - Critics

- **Transplantation and Test Reuse**
  - Grafter

**Recommendation Systems**

- **Program Transformation from Examples**
  - Sydit
  - LASE
  - Cookbook

- **Bug Finding**
  - Refactoring Bugs
  - Cloning Inconsistencies
  - Fault Tracer
  - Modularity Violations
  - Prioritizing Tests for Refactoring

**Automated and Interactive Software Dev Tools**

- **Clone Removal Refactoring**
  - RASE

**Related Topics**

- **Software Refactoring**
  - Refactoring Field Study
  - Quantifying Refactoring Cost and Benefits
  - Impact on Regression Testing
  - Role of API Refactoring

- **API Evolution**
  - Role of API Refactoring
  - API Stability

**Additional Resources**

- RASE
Part 1:
Software Engineering *elevating* Data Science

**Data Scientists in Software Teams**
- Background
- Work Activities
- Challenges
- Best Practices
- Quality Assurance

**SE Tools for Big Data Analytics**
- Interactive Debugger
- Data Provenance
- Automated Debugging
The Emerging Roles of Data Scientists on Software Teams [ICSE 2016]

We are at a tipping point where there are large scale telemetry, machine, process and quality data.

Data scientists are emerging roles in SW teams due to an increasing demand for experimenting with real users and reporting results with statistical rigor.

We have conducted the first in-depth interview study and the largest scale survey of professional data scientists to characterize working styles.
Challenges in Ensuring “Correctness”

**Validation** is a major challenge.

“Honestly, we don’t have a good method for this.” [P457]
“Just because the math is right, doesn’t mean that the answer is right.” [P307]

**Explainability** is important. Participants warned about overreliance on aggregate metrics— “to gain insights, you must go one level deeper.”

Develop locally → Hope it works → Run in cloud → Bug!

Guesswork
BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark

Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, Miryung Kim
[ICSE 2016, FSE Tool Demo 2016, SIGMOD Tool Demo 2017]
Running a Map Reduce Job on Cluster

A user submits a job

A job is distributed to workers in cluster

Each worker performs pipelined transformations on a partition with millions of records
Motivating Scenario: Election Record Analysis

- Alice writes a Spark program that runs correctly on local machine (100MB data) but crashes on cluster (1TB)
- Alice cannot see the crash-inducing intermediate result.
- Alice cannot identify which input from 1TB causing crash
- When crash occurs, all intermediate results are thrown away.

<table>
<thead>
<tr>
<th>VoterID</th>
<th>Candidate</th>
<th>State</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>9213</td>
<td>Sanders</td>
<td>TX</td>
<td>1440023087</td>
</tr>
</tbody>
</table>

```scala
val log = "s3n://poll.log"
val text_file = spark.textFile(log)
val count = text_file
  .filter(line => line.split()[3].toInt > 1440012701)
  .map(line => (line.split()[1], 1))
  .reduceByKey(_ + _).collect()
```

Task 31 failed 3 times; aborting job
ERROR Executor: Exception in task 31 in stage 0 (TID 31)
java.lang.NumberFormatException
Why Traditional Debug Primitives Do Not Work for Apache Spark?

Enabling interactive debugging requires us to **re-think the features of traditional debugger** such as GDB

- Pausing the entire computation on the cloud could reduce throughput
- It is clearly infeasible for a user to inspect billion of records through a regular watchpoint
- Even launching remote JVM debuggers to individual worker nodes cannot scale for big data computing
1. Simulated Breakpoint

Simulated breakpoint replays computation from the latest materialization point where data is stored in memory.
1. Simulated Breakpoint – Realtime Code Fix

Allow a user to fix code after the breakpoint
2. On-Demand Guarded Watchpoint

state.equals("TX") || state.equals("CA")

Watchpoint captures individual data records matching a user-provided guard
3. Crash Culprit Remediation

A user can either correct the crashed record, skip the crash culprit, or supply a code fix to repair the crash culprit.

Task 31 failed 3 times; aborting job
ERROR Executor: Exception in task 31 in stage 0 (TID 31)
java.lang.NumberFormatException
4. Backward and Forward Tracing

A user can also issue tracing queries on intermediate records at realtime.
Demo: BigDebug Interactive Debugger
[FSE 2016 Demo, SIGMOD 2017 Demo]
Q1: How does BigDebug scale to massive data?

BigDebug retains scale up property of Spark. This property is critical for Big Data processing frameworks.
Q2 : What is the performance overhead of debugging primitives?

<table>
<thead>
<tr>
<th>Program</th>
<th>Dataset size (GB)</th>
<th>Max</th>
<th>Max w/o Latency Alert</th>
<th>Watchpoint</th>
<th>Crash Culprit</th>
<th>Tracing</th>
</tr>
</thead>
<tbody>
<tr>
<td>WordCount</td>
<td>0.5 - 1000</td>
<td>2.5X</td>
<td>1.34X</td>
<td>1.09X</td>
<td>1.18X</td>
<td>1.22X</td>
</tr>
<tr>
<td>Grep</td>
<td>20 - 90</td>
<td>1.76X</td>
<td>1.07X</td>
<td>1.05X</td>
<td>1.04X</td>
<td>1.05X</td>
</tr>
<tr>
<td>PigMix-L1</td>
<td>1 - 200</td>
<td>1.38X</td>
<td>1.29X</td>
<td>1.03X</td>
<td>1.19X</td>
<td>1.24X</td>
</tr>
</tbody>
</table>

Max : All the features of BigDebug are enabled

BigDebug poses at most 2.5X overhead with the maximum instrumentation setting.
Titian: Data Provenance Support in Spark

Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, Tyson Condie

[42nd Conference on Very Large Data Bases, VLDB 2016]
Data Provenance – Example in SQL

SELECT time, AVG(temp)
FROM sensors
GROUP BY time

Why ID-2 and ID-3 have those high values?
Step 1: Instrumented Workflow in Spark

Stage 1
- **Input ID** | **Output ID**
  - offset1 | id1
  - offset2 | id2
  - offset3 | id3

- **Hadoop LineageRDD**
  - lines

- **Combiner LineageRDD**
  - pairs

- **Stage 2
- **Input ID** | **Output ID**
  - [p1, p2] | 400
  - [p1] | 4

- **Reducer LineageRDD**
  - counts

- **Stage LineageRDD**
  - reports

- **Input ID** | **Output ID**
  - {id1, id3} | 400
  - {id2} | 4

**Reports**

**Step 1:** Instrumented Workflow in Spark

- **Input ID**
  - offset1
  - offset2
  - offset3

- **Output ID**
  - id1
  - id2
  - id3

- **Input ID**
  - [p1, p2]
  - [p1]

- **Output ID**
  - 400
  - 4

- **Input ID**
  - 400
  - 4

- **Output ID**
  - id1
  - id2
Step 2: Example Backward Tracing

<table>
<thead>
<tr>
<th>Hadoop</th>
<th>Combiner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input ID</td>
<td>Output ID</td>
</tr>
<tr>
<td>offset1</td>
<td>id1</td>
</tr>
<tr>
<td>offset2</td>
<td>id2</td>
</tr>
<tr>
<td>offset3</td>
<td>id3</td>
</tr>
</tbody>
</table>

Reducer:

<table>
<thead>
<tr>
<th>Input ID</th>
<th>Output ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>[p1, p2]</td>
<td>400</td>
</tr>
<tr>
<td>[ p1 ]</td>
<td>4</td>
</tr>
</tbody>
</table>

Stage:

<table>
<thead>
<tr>
<th>Input ID</th>
<th>Output ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>id1</td>
</tr>
<tr>
<td>4</td>
<td>id2</td>
</tr>
</tbody>
</table>
Step 2: Example Backward Tracing

<table>
<thead>
<tr>
<th>Hadoop</th>
<th>Combiner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input ID</td>
<td>Input ID</td>
</tr>
<tr>
<td>Output ID</td>
<td>Output ID</td>
</tr>
<tr>
<td>offset1</td>
<td>{ id1, id 3}</td>
</tr>
<tr>
<td>id1</td>
<td>400</td>
</tr>
<tr>
<td>offset2</td>
<td>{ id2 }</td>
</tr>
<tr>
<td>id2</td>
<td>4</td>
</tr>
<tr>
<td>offset3</td>
<td>p1</td>
</tr>
<tr>
<td>id3</td>
<td>400</td>
</tr>
</tbody>
</table>

Worker1

Reducer.Output ID

Worker2

Combiner.Output ID

Combiner

Reducer_Output ID

Input ID Output ID

p1        400

Combiner

Reducer_Output ID

Input ID Output ID

p1        400
Step 2: Example Backward Tracing

<table>
<thead>
<tr>
<th>Hadoop</th>
<th>Combiner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input ID</td>
<td>Output ID</td>
</tr>
<tr>
<td>offset1</td>
<td>id1</td>
</tr>
<tr>
<td>offset2</td>
<td>id2</td>
</tr>
<tr>
<td>offset3</td>
<td>id3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Combiner.Input ID</th>
<th>Hadoop.Output ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1, id 3</td>
<td>400</td>
</tr>
<tr>
<td>id2</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hadoop</th>
<th>Combiner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input ID</td>
<td>Output ID</td>
</tr>
<tr>
<td>offset1</td>
<td>id1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Combiner.Input ID</th>
<th>Hadoop.Output ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1, ...</td>
<td>400</td>
</tr>
</tbody>
</table>
Automated Debugging in Data Intensive Scalable Computing

Muhammad Ali Gulzar, Matteo Interlandi, Xueyuan Han, Mingda Li Tyson Condie, Miryung Kim

[ACM Symposium on Cloud Computing, SoCC 2017]
Motivating Example

• Alice writes a Spark program that identifies, for each state in the US, the delta between the minimum and the maximum snowfall reading for each day of any year and for any particular year.

• An input data record that measures 1 foot of snowfall on January 1st of Year 1992, in the 99504 zip code (Anchorage, AK) area, appears as

99504, 01/01/1992, 1ft
Problem Definition

- Using a test function, a user can specify incorrect results

```scala
def test(key: String, delta: Float): Boolean = {
  delta < 6000
}
```

Given a test function, the goal is to identify a minimum subset of the input that is able to reproduce the same test failure.
Existing Approach 1: Data Provenance for Spark

It over-approximates the scope of failure-inducing inputs i.e. records in the faulty key-group are all marked as faulty.
Existing Approach 2: Delta Debugging

• Delta Debugging performs a systematic binary search-like procedure on the input dataset using a test oracle function

It does not prune input records known to be irrelevant because of the lack of semantic understanding of data-flow operators
Existing Approach 2: Delta Debugging

- Delta Debugging performs a systematic binary search-like procedure on the input dataset using a test oracle function.

It does not prune input records known to be irrelevant because of the lack of semantic understanding of data-flow operators.
Existing Approach 2: Delta Debugging

- Delta Debugging performs a systematic binary search-like procedure on the input dataset using a test oracle function.

It does not prune input records known to be irrelevant because of the lack of semantic understanding of data-flow operators.
Existing Approach 2: Delta Debugging

- Delta Debugging performs a systematic binary search-like procedure on the input dataset using a test oracle function.

It does not prune input records known to be irrelevant because of the lack of semantic understanding of data-flow operators.
Existing Approach 2: Delta Debugging

- Delta Debugging performs a systematic binary search-like procedure on the input dataset using a test oracle function.

It does not prune input records known to be irrelevant because of the lack of semantic understanding of data-flow operators.
Existing Approach 2: Delta Debugging

- Delta Debugging performs a systematic binary search-like procedure on the input dataset using a test oracle function.

It does not prune input records known to be irrelevant because of the lack of semantic understanding of data-flow operators.
Existing Approach 2: Delta Debugging

- Delta Debugging performs a systematic binary search-like procedure on the input dataset using a test oracle function.

It does not prune input records known to be irrelevant because of the lack of semantic understanding of data-flow operators.
Existing Approach 2: Delta Debugging

- Delta Debugging performs a systematic binary search-like procedure on the input dataset using a test oracle function

It does not prune input records known to be irrelevant because of the lack of semantic understanding of data-flow operators
Existing Approach 2: Delta Debugging

- Delta Debugging performs a systematic binary search-like procedure on the input dataset using a test oracle function.

It does not prune input records known to be irrelevant because of the lack of semantic understanding of data-flow operators.
Automated Debugging in DISC with BigSift

Input: A Spark Program, A Test Function

Output: Minimum Fault-Inducing Input Records

Data Provenance + Delta Debugging

Test Predicate Pushdown

Prioritizing Backward Traces

Bitmap based Test Memoization
Optimization 1: Test Predicate Pushdown

• **Observation:** During backward tracing, data provenance traces through all the partitions even though only a few partitions are faulty.

If applicable, BigSift pushes down the test function to test the output of combiners in order to isolate the faulty partitions.
Optimization 2: Prioritizing Backward Traces

- **Observation:** The same faulty input record may contribute to multiple output records failing the test.

In case of multiple faulty outputs, BigSift overlaps two backward traces to minimize the scope of fault-inducing input records.
Optimization 3: Bitmap Based Test Memoization

- **Observation:** Delta debugging may try running a program on the same subset of input redundantly.
- BigSift leverages bitmap to compactly encode the offsets of original input to refer to an input subset.

We use a bitmap based test memoization technique to avoid redundant testing of the same input dataset.
RQ1: Performance Improvement over Delta Debugging

<table>
<thead>
<tr>
<th>Subject Program</th>
<th>Fault</th>
<th>Running Time (sec)</th>
<th>Debugging Time (sec)</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Original Job</td>
<td>DD</td>
<td>BigSift</td>
</tr>
<tr>
<td>Movie Histogram</td>
<td>Code</td>
<td>56.2</td>
<td>232.8</td>
<td>17.3</td>
</tr>
<tr>
<td>Inverted Index</td>
<td>Code</td>
<td>107.7</td>
<td>584.2</td>
<td>13.4</td>
</tr>
<tr>
<td>Rating Histogram</td>
<td>Code</td>
<td>40.3</td>
<td>263.4</td>
<td>16.6</td>
</tr>
<tr>
<td>Sequence Count</td>
<td>Code</td>
<td>356.0</td>
<td>13772.1</td>
<td>208.8</td>
</tr>
<tr>
<td>Rating Frequency</td>
<td>Code</td>
<td>77.5</td>
<td>437.9</td>
<td>14.9</td>
</tr>
<tr>
<td>College Student</td>
<td>Data</td>
<td>53.1</td>
<td>235.3</td>
<td>31.8</td>
</tr>
<tr>
<td>Weather Analysis</td>
<td>Data</td>
<td>238.5</td>
<td>999.1</td>
<td>89.9</td>
</tr>
<tr>
<td>Transit Analysis</td>
<td>Code</td>
<td>45.5</td>
<td>375.8</td>
<td>20.2</td>
</tr>
</tbody>
</table>

BigSift provides up to a 66X speed up in isolating the precise fault-inducing input records, in comparison to the baseline DD.
RQ2: Debugging Time vs. Original job time

<table>
<thead>
<tr>
<th>Subject Program</th>
<th>Fault</th>
<th>Running Time (sec)</th>
<th>Debugging Time (sec)</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Original Job</td>
<td>DD</td>
<td>BigSift</td>
</tr>
<tr>
<td>Movie Histogram</td>
<td>Code</td>
<td>56.2</td>
<td>232.8</td>
<td>17.3</td>
</tr>
<tr>
<td>Inverted Index</td>
<td>Code</td>
<td>107.7</td>
<td>584.2</td>
<td>13.4</td>
</tr>
<tr>
<td>Rating Histogram</td>
<td>Code</td>
<td>40.3</td>
<td>263.4</td>
<td>16.6</td>
</tr>
<tr>
<td>Sequence Count</td>
<td>Code</td>
<td>356.0</td>
<td>13772.1</td>
<td>208.8</td>
</tr>
<tr>
<td>Rating Frequency</td>
<td>Code</td>
<td>77.5</td>
<td>437.9</td>
<td>14.9</td>
</tr>
<tr>
<td>College Student</td>
<td>Data</td>
<td>53.1</td>
<td>235.3</td>
<td>31.8</td>
</tr>
<tr>
<td>Weather Analysis</td>
<td>Data</td>
<td>238.5</td>
<td>999.1</td>
<td>89.9</td>
</tr>
<tr>
<td>Transit Analysis</td>
<td>Code</td>
<td>45.5</td>
<td>375.8</td>
<td>20.2</td>
</tr>
</tbody>
</table>

On average, BigSift takes 62% less time to debug a single faulty output than the time taken for a single run on the entire data.
RQ2: Debugging Time

On average, BigSift takes 62% less time to debug a single faulty output than the time taken for a single run on the entire data.
BigSift leverages DD after DP to continue fault isolation, achieving several orders of magnitude $10^3$ to $10^7$ better precision.
Part 2:
Data Science *elevating* Software Engineering

Mining API Usage
GitHub

Finding Defects
stack overflow

Visualizing Code
Examples at Scale
API Usage Mining from GitHub [ICSE 2018]

We contrast SO snippets with API usage patterns mined from 380K GitHub projects.
Insight 1: Mining a Large Code Corpus

Our code corpus includes 380K GitHub projects with at least 100 revisions and 2 contributors.

Insight 2: Removing Irrelevant Statements via Program Slicing

We perform backward and forward slicing to identify data- and control-dependent statements to an API method of interest.

![Diagram showing the process of code search, program slicing, call sequence extraction, frequent sequence mining, SMT-based guard condition mining, and resulting API usage patterns.]
void initInterfaceProperties(String temp, File dDir) {
  if(!temp.equals("props.txt")) {
    log.error("Wrong Template.");
    return;
  }
  // load default properties
  FileInputStream in = new FileInputStream(temp);
  Properties prop = new Properties();
  prop.load(in);
  ... init properties ...
  // write to the property file
  String fPath=dDir.getAbsolutePath()+"/interface.prop";
  File file = new File(fPath);
  if(!file.exists()) {
    file.createNewFile();
  }
  FileOutputStream out = new FileOutputStream(file);
  prop.store(out, null);
  in.close();
}
```java
void initInterfaceProperties(String temp, File dDir) {
    if (!temp.equals("props.txt")) {
        log.error("Wrong Template.");
        return;
    }
    // load default properties
    FileInputStream in = new FileInputStream(temp);
    Properties prop = new Properties();
    prop.load(in);
    ... init properties ...
    // write to the property file
    String fPath = dDir.getAbsolutePath() + "/interface.prop";
    File file = new File(fPath);
    if (!file.exists()) {
        file.createNewFile();
    }
    FileOutputStream out = new FileOutputStream(file);
    prop.store(out, null);
    in.close();
}
```
void initInterfaceProperties(String temp, File dDir) {
    if (!temp.equals("props.txt")) {
        log.error("Wrong Template.");
        return;
    }
    // load default properties
    FileInputStream in = new FileInputStream(temp);
    Properties prop = new Properties();
    prop.load(in);
    ... init properties ...
    // write to the property file
    String fPath = dDir.getAbsolutePath() + "/interface.prop";
    File file = new File(fPath);
    if (!file.exists()) {
        file.createNewFile();
    }
    FileOutputStream out = new FileOutputStream(file);
    prop.store(out, null);
    in.close();
}
Insight 3: Capture Semantics Info in API Usage

It is important to capture the temporal ordering, enclosing control structures, and appropriate guard conditions of API calls.

1. Code Search
2. Frequent Sequence Mining
3. SMT-based Guard Condition Mining
Insight 3: Capture Semantics Info in API Usage

It is important to capture the temporal ordering, enclosing control structures, and appropriate guard conditions of API calls.

```java
new File (String); try {
    new FileInputStream(File)arg0.exists();
} catch (IOException) {
}
```
Insight 3: Capture Semantics Info in API Usage

It is important to capture the temporal ordering, enclosing control structures, and appropriate guard conditions of API calls.

```java
new File(String); try {; new FileInputStream(File)@arg0.exists(); } catch (IOException) {; }
```
Insight 3: Capture Semantics Info in API Usage

It is important to capture the temporal ordering, enclosing control structures, and appropriate guard conditions of API calls.

```java
new File (String); try {; new FileInputStream(File)arg0.exists(); } catch (IOException) {; }
```
Insight 3: Capture Semantics Info in API Usage

It is important to capture the temporal ordering, enclosing control structures, and appropriate guard conditions of API calls.

```
new File (String); try {; new FileInputStream(File)@arg0.exists(); } catch (IOException) {; }
```
Insight 4: Variations in Guard Conditions

Guard conditions are canonicalized and grouped based on logical equivalence.

Two equivalent guard conditions for `String.substring`:

\[ \text{arg0} \geq 0 \land \text{arg0} \leq \text{rcv.length}() \iff \text{arg0} > -1 \land \text{arg0} < \text{rcv.length}() + 1 \]
Insight 4: Variations in Guard Conditions

Guard conditions are canonicalized and grouped based on logical equivalence.

Two equivalent guard conditions for String.substring:

\[ \text{arg0} \geq 0 \land \text{arg0} \leq \text{rcv.length}() \iff \text{arg0} > -1 \land \text{arg0} < \text{rcv.length}()+1 \]
Insight 4: Variations in Guard Conditions

Guard conditions are canonicalized and grouped based on logical equivalence.

Two equivalent guard conditions for String.substring:

\[ \text{arg0} \geq 0 \land \text{arg0} \leq \text{rcv.length()} \iff \text{arg0} > -1 \land \text{arg0} < \text{rcv.length()} + 1 \]
API Misuse Detection in StackOverflow

We examine 220K SO posts with 180 confirmed patterns.

=> 31% of SO posts contain API usage violations!

API Misuses are Prevalent on Stack Overflow

Highly-voted posts are not necessarily more reliable in terms of correct API usage.

Network, database, IO, crypto, string manipulation APIs are more likely to be misused.
ExampleCheck: Augmenting Stack Overflow with API Usage Patterns Mined from GitHub

① Pop-up window
② API misuse description
③ Fix suggestion
④ Supporting GitHub examples
⑤ Pagination for multiple misuses
⑥ Like or dislike this reported misuse
ExampleCheck
Offered by: Tianyi Zhang

Available on Chrome

Overview  Reviews  Related
Examplore: Visualizing Code Examples at Scale [CHI 2018]

Examplore visualizes hundreds of code examples using the same API.
Examplore: Visualizing Code Examples at Scale

CHI 2018

http://examplore.cs.ucla.edu:3000

```java
@override
public void readFromImage(String filename) throws IOException {
  in = new FileInputStream(filename);
  prop.load(in);
}
```

```java
private synchronized InputStream openInputStream() throws IOException {
  if (file != null) {
    return new FileInputStream(file);
  }
  else {
    return new ByteArrayInputStream(memory.getBuffer(), 6, memory.getCount());
  }
}
```

```java
public InputStream getResourceContents(String path) {
  File file = new File(_basePath + "" + path);
  try {
    return new FileInputStream(file);
  }
  catch (FileNotFoundException e) {
    throw new IllegalArgumentException(e);
  }
}
```

```java
public InputStream getInputStream() throws MessagingException {
  try {
    return new FileInputStream(mFile);
  }
  catch (IOException ioe) {
    throw new MessagingException("Unable to open body", ioe);
  }
}
```

```java
/** ファイルから画像情報を生成 */
public static ImageInfo getImageInfo(File imageFile) throws IOException {
  BufferedInputStream bis = new BufferedInputStream(new FileInputStream(imageFile));
  ImageInfo imageInfo = ImageInfo.getImageInfo(bis, -1);
  bis.close();
  return imageInfo;
}
```
Thanks to my collaborators

**UCLA on Big Data Debugging:** Muhammad Ali Gulzar*, Tyson Condie, Matteo Interlandi, Mingda Li, Michael Han, Sai Deep Tetali, Todd Millstein

**Microsoft Research on Data Scientist Studies:** Tom Zimmermann, Andrew Begel, and Rob DeLine

**UCLA, UC Berkeley, Iowa State on API Usage Mining:** Tianyi Zhang*, Elena Glassman, Bjorn Hartmann, Ganesha Upadhyaya, Anastasia Reinhart, Hridesh Rajan
Big Data needs **awesome software engineering tools**

- Diagnose
  - Debugging
  - Intelligent sampling and testing
  - Root cause analysis
- Fix
  - Data cleaning
- Optimize
  - Performance analytics
  - Code analytics