Ecosystem-scale call graphs

Mehdi Keshani
12:30 - 13:30

SERG Lunch
01 April 2020
● Outline
  ○ What is FASTEN and how it works
  ○ FASTEN plugins
  ○ How to scale call graph construction
  ○ Introducing a new approach for call graph construction on scale
  ○ Evaluation of the approach
What is FASTEN?

- The main aim of the FASTEN project is to make software package management systems more robust and intelligent.
- Call graph level analysis
- The project’s scientific objectives:
  - Fine-grained ecosystem analysis for C, Java and Python
  - Ecosystem-wide change impact analysis
  - Compliance monitoring
  - ...
How does it look like?
How it works?

Graph

Call Graph Creator

Dependency Resolver

Connecto

FASTEN Server

REST API

USERS

CI Servers

Connecto

Dataflow plug-ins

Database

Analyzers

Query

<temporal/graph/ts>

<events/package/new>

<resolve/graph/ts>

<temporal/graph/ts>

<events/insert>

<query/ufi/meta>

Query

Kafka Producers

Kafka Consumers

External plug-ins

Quality and Risk

Security

Impact

Compliance
Dataflow

- There is a combination of plugins interacting via Kafka
- A dataflow plugin is a tool that accepts a record from a Kafka topic and produces one or more records to a Kafka topic
- Inputs, outputs and Error handling is occurring within Kafka
- Distribution is handled by subscribing to the same Kafka consumer group
Analyzers

- It’s the core component of the FASTEN KB, which consists of:
  - Security, Quality, Risk
    - E.g. property propagation of quality measurements
  - License and Compliance
    - E.g. Investigating licencing per file using build graphs for Java, C and Python
  - Change Impact Analysis
    - E.g. Algorithms and heuristics for reachability on the call graphs like Updatera
CG Plug-in: External sources

- A Kafka Topic of all ecosystem libraries
- A crawler was developed in Python to extract Maven coordinates

```
"groupId": "avalon", "artifactId": "avalon-framework", "version": "4.1.4", "date": "1127187900"
```
Different frameworks

- WALA
  - Heavy compare to OPAL
  - FASTEN plugin

- OPAL
  - Fast and Lightweight [1]
  - Highly-configurable software product line [2]
  - FASTEN plugin
  - Usage
    - As a Maven library
    - Scala convertors in the plugin

## Java call graph generators

### Table 4: Comparison of algorithms w.r.t. call graph size and runtime.

<table>
<thead>
<tr>
<th>Project</th>
<th>#Methods</th>
<th>SootCHA</th>
<th>SootRTA</th>
<th>SootVTA</th>
<th>SootSPARK</th>
<th>OPALRTA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>#RM</td>
<td>#RM</td>
<td>#RM</td>
<td>#RM</td>
<td>#RM</td>
</tr>
<tr>
<td></td>
<td>all (incl. JDK)</td>
<td>project</td>
<td>time</td>
<td>time</td>
<td>time</td>
<td>time</td>
</tr>
<tr>
<td>jasml</td>
<td>160 564</td>
<td>265</td>
<td>12 184</td>
<td>18 s</td>
<td>12 134</td>
<td>75 s</td>
</tr>
<tr>
<td>javacc</td>
<td>162 484</td>
<td>2 185</td>
<td>13 035</td>
<td>22 s</td>
<td>12 986</td>
<td>97 s</td>
</tr>
<tr>
<td>jext</td>
<td>163 569</td>
<td>3 270</td>
<td>34 604</td>
<td>97 s</td>
<td>34 470</td>
<td>697 s</td>
</tr>
<tr>
<td>proguard</td>
<td>165 797</td>
<td>5 498</td>
<td>36 425</td>
<td>84 s</td>
<td>36 256</td>
<td>647 s</td>
</tr>
<tr>
<td>sablecc</td>
<td>162 670</td>
<td>2 371</td>
<td>14 138</td>
<td>18 s</td>
<td>14 088</td>
<td>104 s</td>
</tr>
<tr>
<td><strong>average</strong></td>
<td></td>
<td></td>
<td>47.8 s</td>
<td>324 s</td>
<td>52 s</td>
<td>54.4 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Project</th>
<th>#Methods</th>
<th>WALA_{RTA}</th>
<th>WALA_{0-CFA}</th>
<th>WALA_{N-CFA}</th>
<th>WALA_{0-1-CFA}</th>
<th>DOOP_{CI}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>#RM</td>
<td>time</td>
<td>#RM</td>
<td>time</td>
<td>#RM</td>
</tr>
<tr>
<td></td>
<td>all (incl. JDK)</td>
<td>project</td>
<td>WALA_{RTA}</td>
<td>WALA_{0-CFA}</td>
<td>WALA_{N-CFA}</td>
<td>WALA_{0-1-CFA}</td>
</tr>
<tr>
<td>jasml</td>
<td>160 564</td>
<td>265</td>
<td>75 817</td>
<td>362 s</td>
<td>timed out</td>
<td>timed out</td>
</tr>
<tr>
<td>javacc</td>
<td>163 484</td>
<td>2 185</td>
<td>76 643</td>
<td>399 s</td>
<td>timed out</td>
<td>timed out</td>
</tr>
<tr>
<td>jext</td>
<td>163 569</td>
<td>3 270</td>
<td>79 513</td>
<td>411 s</td>
<td>timed out</td>
<td>timed out</td>
</tr>
<tr>
<td>proguard</td>
<td>165 797</td>
<td>5 498</td>
<td>80 240</td>
<td>465 s</td>
<td>timed out</td>
<td>timed out</td>
</tr>
<tr>
<td>sablecc</td>
<td>162 670</td>
<td>2 371</td>
<td>77 607</td>
<td>460 s</td>
<td>timed out</td>
<td>timed out</td>
</tr>
<tr>
<td><strong>average</strong></td>
<td></td>
<td></td>
<td>419.4 s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Call Graph Plugins

- Reads from Kafka and writes to Kafka
- Its service is to generate call graphs using call graph module
- It is deployed on K8s
- Normally generates 10 CG per second with 10 workers using OPAL
But they are partial graphs!

- Partial program analysis
  - When we do not analyze the entire program but only some parts of it
- Existing tools need entire class path (including libraries) to generate a whole program CG
- A lot of duplicate calculation
- Is there a better approach?
Solution

- GC generators (e.g. WALA) expect a full transitive closure per client
- Dependency resolution is time dependent
- Idea: Split CG construction from CG linking
  - **construction**: make a call graph per package, mark *linkage points* and class hierarchy information
  - **linking**: after dependency resolution, link *linkage points*
What motivates us?

- Package management ecosystems are changing continuously
- There are almost 3M libraries only on Maven
- Duplicate calculations is a big challenge for scalability
  - A majority of packages depends on a small minority of other packages [3]
  - Variant dependency tree
- Use cases that need code analysis (e.g. FASTEN or CIs) with a lot of users
  - They have to do a lot of duplicate computation per client
  - Existing tools will calculate the full transitive closure CG per request
  - With this approach result is one query away!

Dynamic dispatch calls

- Example
  a. What will it print if we run it?
  b. What methods would be called at runtime?
  c. What edges should the ideal call graph have?

```java
public class Dynamic Dispatch Example {
  public static void main(String[] args) {
    A b1 = new B();
    A c1 = new C();
    A b2 = b1;
    A c2 = c1;

    // what will get printed?
    b2.print(c2);
  }

  public static class A extends Object {
    public void print(A object) {
      System.out.println("Instance of " + object.getClass().getSimpleName() + " passed to A");
    }
  }

  public static class B extends A {
    public void print(A object) {
      System.out.println("Instance of " + object.getClass().getSimpleName() + " passed to B");
    }
  }

  public static class C extends B {
    public void print(A object) {
      System.out.println("Instance of " + object.getClass().getSimpleName() + " passed to C");
    }
  }

  public static class D extends C {
    public void print(A object) {
      System.out.println("Instance of " + object.getClass().getSimpleName() + " passed to D");
    }
  }
}
```
Soundness

- Run time: \(\text{\texttt{b2.print(c2)}}\) to B's \texttt{print}
- It could be tricky to statically determine the runtime type of \texttt{b2} also to figure out exactly which method would get called at runtime
- We say a call graph is “sound” if it has all the edges that are possible at runtime
- We say a call graph is “precise” if it does not have edges that do not occur at runtime
- It is easy to be sound, but it is hard to be sound \textit{and} precise
- Soundness is very important in some use cases such as security
- Sound algorithms over approximate
What algorithm to pick as the basis?

- Popular call graph construction algorithms
  - Each of them has variations on the literature

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
<th>Sound</th>
<th>Precision</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td>Adds an edge to all reachable methods with similar signature.</td>
<td>✔</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>CHA</td>
<td>Adds edges to methods declared in the subtype hierarchy of the declared type of the receiver object (default for most static analysis)</td>
<td>✔</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>RTA</td>
<td>Filters CHA edges based on the allocated objects in the reachable methods.</td>
<td>✗</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>VTA</td>
<td>RTA + builds a graph of each variable and all of its assignments</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What is needed from each package version

- All internal calls of the library
- Marked external calls to package boundary
- All types existing in the library for further CHA analysis
  - List of its methods,
  - Classes that extends,
  - And interfaces that implements
Package version call graph

```json
{
  "product": "org.slf4j.slf4j-api",
  "forge": "mvn",
  "depset": [],
  "version": "1.7.29",
  "cha": {
    "/org.slf4j/LoggerFactory": {
      "sourceFile": "Log.java",
      "methods": [
        ["/org.slf4j/LoggerFactory.bind()%2Fjava.lang%2FVoid", 1],
        ["/org.slf4j/LoggerFactory.replayEvents()%2Fjava.lang%2FVoid", 2],
        ...
      ],
      "superInterfaces": [],
      "superClasses": ["/java.lang/Object"]
    },
    "/org.slf4j.helpers/FormattingTuple": { ... },
    ...
  },
  "graph": [
    "internalCalls": [
      "1",
      "2"
    ], ...
    "externalCalls": [
      ["2",
       "///java.lang/String.contains(CharSequence)Boolean",
       {
         "invokevirtual": "1"
       }]
    ]
  ],
  "timestamp": 1574072773
}
```
Merge assumption

- Dependency tree is variant
  - Merge algorithm should be independent of dependency tree
- Input: a package version call graph and a list of dependencies
- Output: fully resolved call graph of the first argument
- ResolvedCG_Pkg1:v1.0.0 = Merge(Pkg1:v1.0.0, List<Pkg>)
- Full dependency trees should be broken to pieces

1_resolved = Merge(1, {2, 3, 4})
4_resolved = Merge(4, {5, 6, 7})
5_resolved = Merge(5, {8})
8_resolced = Merge(8, {9, 10})
Merge revision call graphs

- **Entry points**
  - In within-library scenario: (!Abstract && !Private) methods
  - In merge scenario: External calls

- **RA**
  - Search for the external node’s signature in direct dependencies

```
for (call in external calls) {
    for (dependency in dependencies) {
        for (method in dependency.methods()) {
            if (call.target().signature() == method.signature()) {
                resolve(call);
            }
        }
    }
}
```

Pseudocode of RA merge algorithm
Merge revision call graphs

- **CHA**
  - For each call target of external call
  - Extract the receiver type
  - Search for receiver type in direct deps
  - Subtypes of the receiver type in direct deps
  - Search for the target’s signature
  - In receiver type and all of its subtypes

Pseudocode of CHA merge algorithm

```
for (call in external calls) {
    if (isDynamicDispatched(call)) {
        for (dependency in dependencies) {
            for (type in dependency.types()) {
                if (type == call.receiverType() or
                    type inherits from call.receiverType() or
                    type implements call.receiverType()){
                    if (type, implementsMethod(call.target())) {
                        resolve(call)
                    }
                }
            }
        }
    } else {
        for (dependency in dependencies) {
            for (type in dependency.types()) {
                if (type == call.receiverType() and
                    type, implementsMethod(call.target())) {
                    resolve(call)
                }
            }
        }
    }
}
```
How to Evaluate?

- **Soundness:**
  - Compare with the soundness of the base framework
  - Run both algorithms on a benchmark
  - Compare the soundness and precision
  - Goal: Be similar to the base framework as much as possible

- **Scalability**
  - Compare with the scalability of the base framework
  - Run both algorithms on the whole or a substantial portion of an ecosystem
  - Compare the computation time
  - Goal: be better than base framework
Soundiness

- There exists a paradox in static analysis
  - Some language features can make call graph construction undecidable
  - Static analysis tools
    - On one hand try to be sound
    - On the other hand deliberately not very supportive for all language features
- Experts in field came up with the concept of Soundiness
  - A soundy analysis aims to be as sound as possible without excessively compromising precision and/or scalability.
Benchmark

- There is a benchmark of 122 test cases considering all possible types of call in java annotated with the real edges [1]
- Steps:
  - Extract test cases
  - Compile and create jar files from them
  - Split the jar files to the different class files
  - Once generate CG for the jar file with the base framework
  - Once generate partial CGs for class files with the base framework
  - Merge partial CGs
  - Run CGMather on jar file CG to match with annotations
  - Run CGMather on Merged CG to match with annotations
  - Compare the output (sound/unsound/imprecise)

---

## Comparison?

<table>
<thead>
<tr>
<th>Language feature</th>
<th>Framework</th>
<th>Sound</th>
<th>Unsound</th>
<th>Imprecise</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL1</td>
<td>Merge</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>Base framework</td>
<td></td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>CL2</td>
<td>Merge</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>Base framework</td>
<td></td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Address why</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NJB6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Scalability

● Steps:
  ○ Calculate dependency trees for all maven libraries
  ○ Construct partial CGs using base framework
  ○ Store partial CGs in DB
  ○ Merge partial CGs with a DB query
  ○ Construct CGs using base framework
  ○ Compare the calculation time