Ecosystem-scale call graphs

Mehdi Keshani

12:30 - 13:30

SERG Lunch

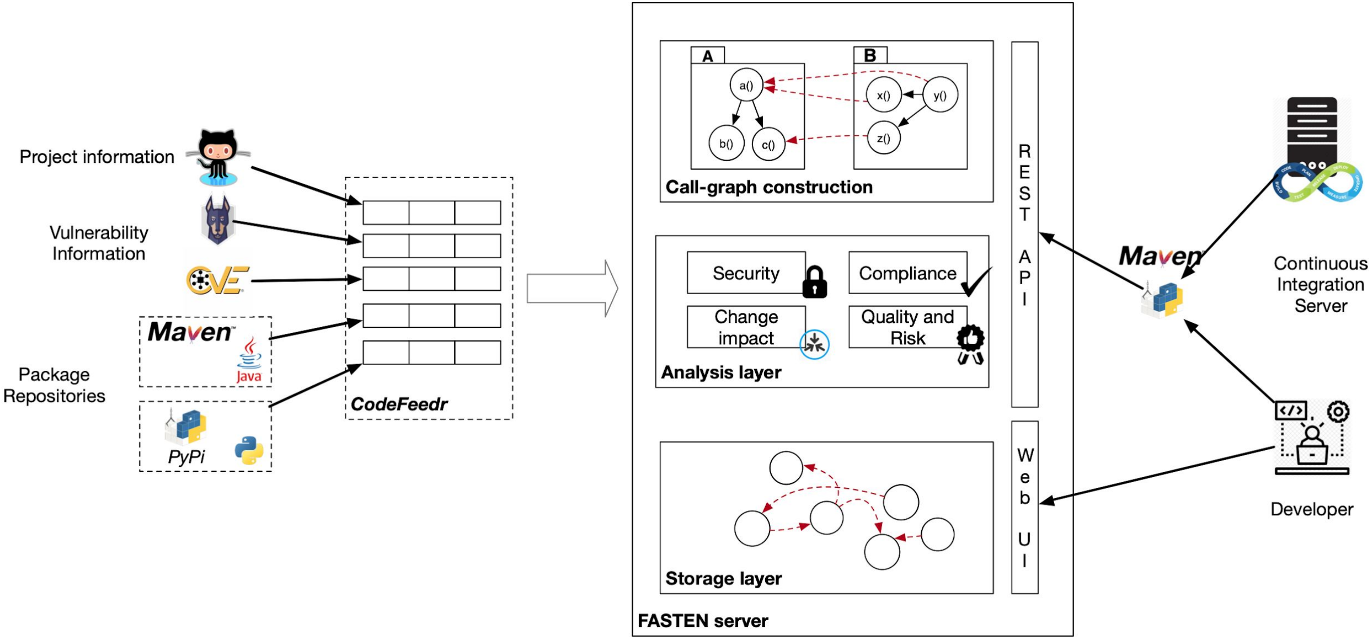
01 April 2020

- 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

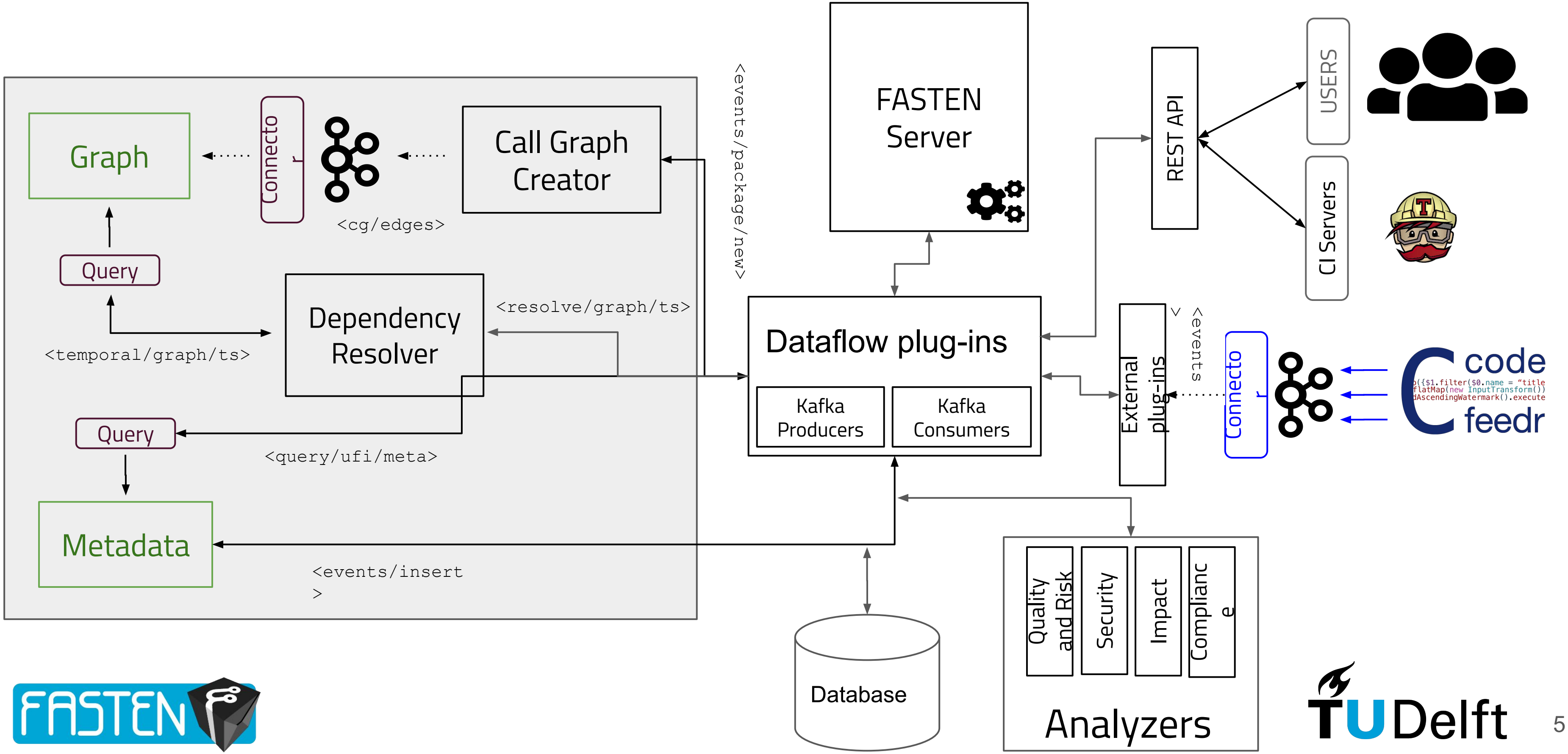
What is FASTEN?

- The main aim of the FASTEN project is 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?



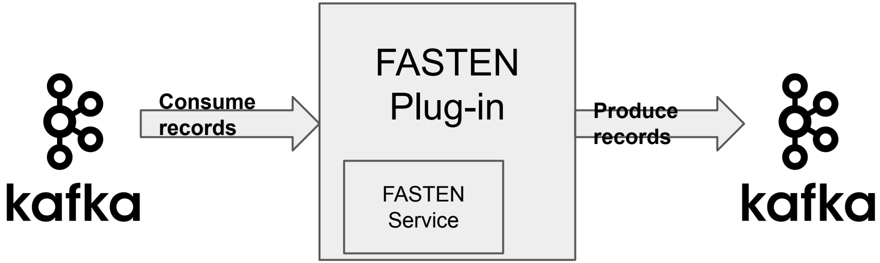
How it works?



Graph
Connector
Call Graph Creator
<cg/edges>
<events/package/new>
FASTEN Server
REST API
USERS
CI Servers
Query
Dependency Resolver
<resolve/graph/ts>
<temporal/graph/ts>
Dataflow plug-ins
Kafka Producers
Kafka Consumers
External plug-ins
<events>
Connector
code feedr
Query
<query/ufi/meta>
Metadata
<events/insert>
Database
Quality and Risk
Security
Impact
Compliance
Analyzers

Dataflow

- There is a combination of plugins interacting via Kafka
- A dataflow plugin is 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*

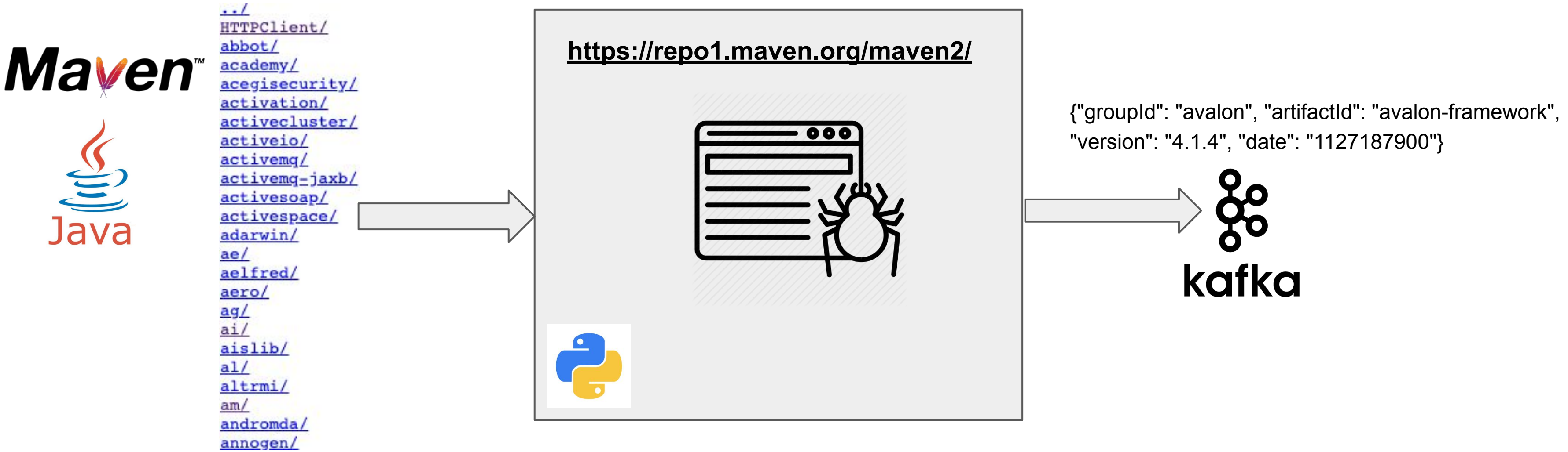


Analyzers

- 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



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

[1] Reif, Michael, et al. "Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs." Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 2019.
[2] Eichberg, Michael, and Ben Hermann. "A software product line for static analyses: the OPAL framework." Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis. ACM, 2014.

Java call graph generators

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

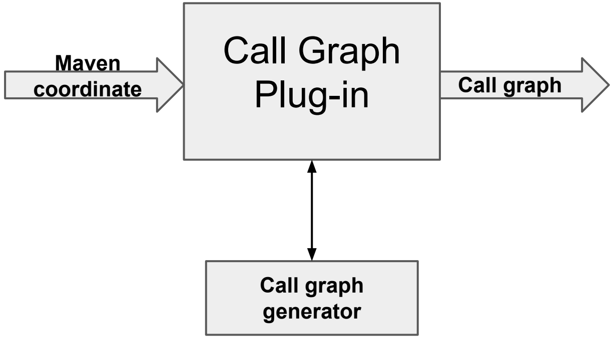
Project	#Methods		$Soot_{CHA}$		$Soot_{RTA}$		$Soot_{VTA}$		$Soot_{SPARK}$		$OPAL_{RTA}$	
	all (incl. JDK)	project	#RM	time	#RM	time	#RM	time	#RM	time	#RM	time
jasml	160 564	265	12 184	18 s	12 134	75 s	8 012	17 s	10 356	22 s	3 195	13 s
javacc	162 484	2 185	13 035	22 s	12 986	97 s	8 863	22 s	9 752	17 s	4 222	12 s
jext	163 569	3 270	34 604	97 s	34 470	697 s	20 259	97 s	20 605	73 s	15 705	15 s
proguard	165 797	5 498	36 425	84 s	36 256	647 s	20 928	100 s	28 912	136 s	7 771	11 s
sablecc	162 670	2 371	14 138	18 s	14 088	104 s	9 687	24 s	12 101	24 s	4 932	11 s
average				47.8 s		324 s		52 s		54.4 s		12.4 s

Project	#Methods		$WALA_{RTA}$		$WALA_{0\text{-}CFA}$		$WALA_{N\text{-}CFA}$		$WALA_{0\text{-}1\text{-}CFA}$		$DOOP_{CI}$	
	all (incl. JDK)	project	#RM	time	#RM	time	#RM	time	#RM	time	#RM	time
jasml	160 564	265	75 817	362 s	timed out		timed out		timed out		14 149	579 s
javacc	163 484	2 185	76 643	399 s	timed out		timed out		timed out		14 952	618 s
jext	163 569	3 270	79 513	411 s	timed out		timed out		timed out		27 194	1 698 s
proguard	165 797	5 498	80 240	465 s	timed out		timed out		timed out		18 205	949 s
sablecc	162 670	2 371	77 607	460 s	timed out		timed out		timed out		15 774	680 s
average				419.4 s	-		-		-			904.8 s

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

{"groupId": "ant", "artifactId": "ant-antlr", "version": "1.6", "date": "1127187840"}



{["/org.apache.spark.repl.h2o/H2OIMainHelper$class.newREPLDirectory(H2OIMainHelper)%2Fjava.io%2FFile","//SomeDependency/scala/Option.getOrElse(Function0)%2Fjava.lang%2FObject"],["/org.apache.spark.repl.h2o/H2OIMainHelper$class.newREPLDirectory(H2OIMainHelper)%2Fjava.io%2FFile","//SomeDependency/java.lang/NullPointerException.NullPointerException()Void"],["/org.apache.spark.repl.h2o/H2OIMainHelper$class.newREPLDirectory(H2OIMainHelper)%2Fjava.io%2FFile","//SomeDependency/org.apache.spark/SparkConf.getOption(%2Fjava.lang%2FString)%2Fscala%2FOption"],["/org.apache.spark.repl.h2o/H2OIMainHelper$class.newREPLDirectory(H2OIMainHelper)%2Fjava.io%2FFile","//SomeDependency/java.lang/NullPointerException.NullPointerException()Void"],["/org.apache.spark.repl.h2o/H2OIMainHelper$class.newREPLDirectory(H2OIMainHelper)%2Fjava.io%2FFile","//SomeDependency/org.apache.spark/SparkConf.SparkConf()%2Fjava.lang%2FVoid"]],"timestamp":1492742760}

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!

[3] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empirical Software Engineering 24, 1 (2019), 381–416.

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 DynamicDispatchExample {

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

}
```

Soundness

- Run time: (`b2.print(c2)`) to `B`'s `print`
- It could be tricky to statically determine the runtime type of `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 *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

Algorithm	Description	Sound	Precision	Scalability
RA	Adds an edge to all reachable methods with similar signature.	✔	-	+
CHA	Adds edges to methods declared in the subtype hierarchy of the declared type of the receiver object (default for most static analysis)	✔	↓	↑
RTA	Filters CHA edges based on the allocated objects in the reachable methods.	×	↓	↑
VTA	RTA + builds a graph of each variable and all of its assignments	×	+	-

FASTEN

TU Delft

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

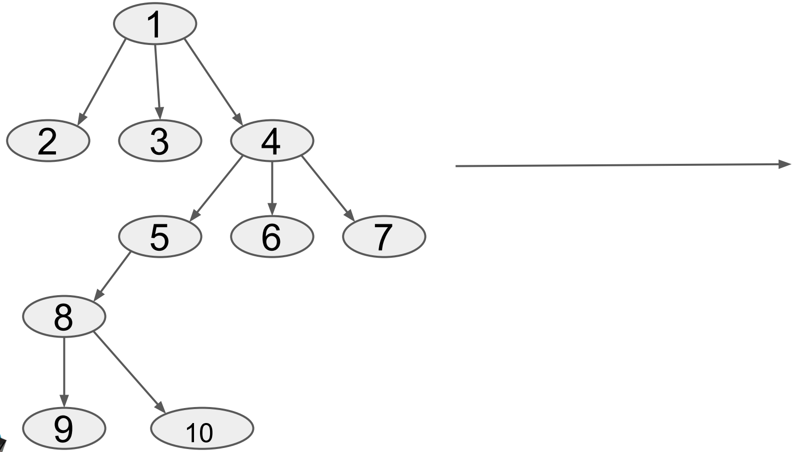

```
{
    "product": "org.slf4j.slf4j-api",
    "forge": "mvn",
    "depset": [],
    "version": "1.7.29",
    "cha": {
        "/org.slf4j/LoggerFactory": {
            “sourceFile”: “Log.java”
            "methods": [
            ["/org.slf4j/LoggerFactory.bind()%2Fjava.lang%2FVoid",1],
            ["/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
}
```


FASTEN

TU Delft

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})

FASTEN

TU Delft

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

```
 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)
             }
          }
       }
    }
}
```

Pseudocode of CHA merge algorithm

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 Soundines
  - A *soundy* analysis aims to be as sound as possible without excessively compromising precision and/or scalability.

In Defense of Soundiness: A Manifesto

Soundy is the new sound.

STATIC PROGRAM ANALYSIS is a key component of many software development tools, including compilers, development environments, and verification tools. Practical applications of static analysis have grown in recent years to include tools by companies such as Coverity, Fortify, GrammaTech, IBM, and others. Analyses are often expected to be *sound* in that their result models all possible executions of the program under analysis. Soundness implies the analysis computes an over-approximation in order to stay tractable; the analysis result will also model behaviors that do not actually occur in any program execution. The *precision* of an analysis is the degree to which it avoids such spurious results. Users expect analyses to be sound as a matter of course, and desire analyses to be as precise as possible, while being able to *scale* to large programs.

Soundness would seem essential for any kind of static program analysis. Soundness is also widely emphasized in the academic literature. Yet, in practice, soundness is commonly eschewed: we are not aware of a *single* that does not purposely make unsound choices. Similarly, virtually all published whole-program analyses are unsound and omit conservative handling of common language features when applied to *real programming languages*. dominant practice is one of treating soundness as an engineering choice.

In all, we are faced with a paradox: on the one hand we have the ubiquity of unsoundness in any practical whole-program analysis tool that has a claim

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*)

Table 1: Overview of the Test Suite.

Category	Abbreviation	# Test Cases
Classloading	CL	4
Dynamic Proxies	DP	1
Interface Default Methods	J8DIM	6
Static Interface Methods	J8SIM	1
Java 8 invokedynamics	MR/Lambda	11
JVM Calls	JVMC	5
Library Analysis	LIB	5
Trivial Reflection	TR	9
Locally Resolveable Reflection	LRR	3
Context-sensitive Reflection	CSR	4
Method Handles	MH	9
Class.forname Exceptions	CFNE	4
Non-virtual Calls	NVC	6
Serialization	Ser	9
Externalizable	ExtSer	3
Lambda Serialization	LamSer	2
Signature Polymorphic Methods	SPM	7
Static Initializers	SI	8
TYPES	-	6
Unsafe	-	7
Virtual Calls	VC	4
Java 9/10 Features	J9+	2
Non-Java Bytecode	NJB	6
Total		122

[1] Reif, Michael, et al. "Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs. " Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 2019.

Comparison?

Language feature	Framework	Sound	Unsound	Imprecise	Comparison
CL1	Merge	✓	×	×	✓
	Base framework	✓	×	×	✓
CL2	Merge	×	✓	×	Address why
	Base framework	✓	×	×	
…					
NJB6					

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

Thanks!