Acta Cybernetica **24** (2019) 131–155.

A Preparation Guide for Java Call Graph Comparison: Finding a Match for Your Methods[*]

Zoltán Ságodi[a] and Edit Pengő[a]

Abstract

Call graphs provide a basis for numerous interprocedural analyzers and tools, therefore it is crucial how precisely they are constructed. Developers need to know the features of a call graph builder before applying it to subsequent algorithms. The characteristics of call graph builders are best understood by comparing the generated call graphs themselves. The comparison can be done by matching the corresponding nodes in each graph and then analyzing the found methods and calls.

In this paper, we developed a process for pairing the nodes of multiple call graphs produced for the same source code. As the six static analyzers that we collected for call graph building handles Java language elements differently, it was necessary to refine the basic name-wise pairing mechanism in several steps. Two language elements, the anonymous and generic methods, needed extra consideration. We describe the steps of improvement and our final solution to achieve the best possible pairing we are able to provide, through the analysis of the Apache Commons-Math project.

Keywords: call graph, Java, static analysis

1 Introduction

Static source code analyzers play an important role in producing high-quality software that satisfies the requirements of today's industrial development. They help programmers eliminate flaws and rule violations early on by automatically analyzing the subject system and highlighting its potentially erroneous parts. Usually, the source code is converted into an Abstract Syntax Tree[1] (AST) - like representation,

[*]This research was supported by the EU-funded Hungarian national grant GINOP-2.3.2-15-2016-00037 titled "Internet of Living Things". This research was supported by the project "Integrated program for training new generation of scientists in the fields of computer science", no EFOP-3.6.3-VEKOP-16-2017-0002. The project has been supported by the European Union and co-funded by the European Social Fund.

[a]University of Szeged, Department of Software Engineering, E-mail: `{sagodiz,pengoe}@inf.u-szeged.hu`

[1]Abstract Syntax Tree represents the syntactic structure of the source code in a hierarchical tree-like form

DOI: 10.14232/actacyb.24.1.2019.10

which is the basis for further transformations, optimizations, and operations, for example, call graph creation. The capabilities of such analyzer tools depend on the complexity of the internal representations and algorithms they use.

Call graphs are directed graphs representing control flow relationships among the methods of a program. The nodes of the graph denote the methods, while an edge from node a to node b indicates that method a invokes method b. Call graphs are essential building blocks of interprocedural control and data flow modeling. They can be used during control flow analysis, program slicing, program comprehension, bug prediction, refactoring, bug-finding, verification, security analysis, and whole-program optimization [8, 13, 35, 37]. The accuracy of call graphs influences the results of the subsequent analyses, consequently, careful consideration is needed during the selection of the construction method. The most obvious difficulty that a static call graph builder has to face is the handling of polymorphic calls and other cases when the target of a call depends on the runtime behavior of the program. There are plenty of call graph builder algorithms that address this challenge and try to make assumptions about what methods could be called. They have extensive literature, including detailed comparisons [14, 15, 20, 21, 24, 32]. There are other factors that can cause differences in the output of two call graph creator tools, for example, the handling of different kinds of initializations or anonymous classes.

In the future, we plan to compare and characterize call graph builders based on how they handle such factors. However, the first step towards this goal is to make the produced call graphs comparable. To compare the call graphs that were generated by different tools for the same source code, we have to match the nodes – i.e. the methods – that correspond to each other and then evaluate what types of methods and calls were found by each tool. However, matching the methods to each other is challenging, since it is not certain that all tools will find the same methods or they might name them differently. Using the line information for refining the pairing mechanism can also cause difficulties. Although node pairing is the basis of call graph comparison, we found no satisfactory description about it in previous works. Therefore, we decided to summarize the problems we encountered and our attempts at solving them. Section 2 provides the related work, whilst Section 3 introduces the investigated call graph builders. Section 4 illustrates the obstacles of the method pairing mechanism and our step-by-step improvements with results. In Section 5, our approach is compared with a topology based solution. Finally, we draw our conclusions and outline future work in Section 6.

2 Related work

The way to compare the capabilities of call graph builder tools is through comparing the call graphs they generate. Due to the increasing number of extremely large graphs and their wide area of usability, there are many algorithms and metrics available for comparing general directed and undirected graphs [19, 22, 34]. However, these methods cannot be directly applied to call graphs, especially if they were produced by different analyzer tools. Call graphs are directed graphs whose nodes correspond to the methods in the source code. Even if the structure of two call graphs is isomorphic, they can be considered entirely different because of the labeling of the nodes. Therefore, to make them comparable, first we have to find a mapping, which is the aim of this paper.

There are several proposals for comparing labeled graphs if a mapping is already present. Champin and Solnon defined a similarity with respect to a given mapping between two graphs that have multiple labels both on their nodes and edges [7]. A graph is described by the set of all of its features, e.g. the set of node-label and edge-label pairs. The similarity measure is calculated based on a simplified version of Tversky's formula [33]. They also proposed an algorithm for finding the best mapping for reaching the maximum similarity, which provides a qualitative description of the differences between the two graphs.

There are algorithms available exactly for comparing labeled graphs that share the same node set [38]. In case of call graphs that were produced by different tools and algorithms this condition cannot be ensured. The simplest way to compare graphs with the same node set is to handle the adjacency matrices as vectors and calculate an edit distance, in other words, the number of different edges [12, 30]. Wicker et al. introduced a dissimilarity measure for graphs like these based on their eigenvalues and eigenvectors [38], which takes into account the global graph structures as well.

The precision and structure of the call graphs greatly depends on the algorithms that the builder tools used. If several call targets are possible for a given call site, more examination is needed to determine which edges should be connected. There are context-dependent and context-independent solutions; naturally, the choice influences the result. Context-dependent methods are more accurate, but in return they use more resources. To mitigate the resource demands of such methods, the analysis of the programs often starts only from the `main` method or a few entry points instead of starting from every method of the analyzed source code. This, however, will likely lower the accuracy of the method. Context-independent methods for object oriented languages can be improved with the following algorithms: *CHA* [9], *RTA* [4], *XTA*[32], *VTA* [31]. In case of the comparing these call graph building strategies [1, 14, 15], node matching is usually not an issue because the algorithms are implemented in the same environment and language elements are handled similarly. The nodes of the produced call graphs are the subset of each other's node set with the same naming convention, therefore, the main difference comes from the number of edges.

In this paper, we considered the results of static analyzer tools, meaning that we worked with the so called static call graphs. However, call graphs can be composed with dynamic tools as well from actual executions of the analyzed program. Lhoták [20] compared static call graphs generated by Soot [29] and dynamic call graphs created with the help of the *J [28] dynamic analyzer. He built a framework to compare call graphs, discussed the challenges of the comparisons, and presented an algorithm to find the causes of the potential differences in call graphs. The paper does not describe the difficulties of matching the nodes of the call graphs that were provided by different sources. The reason could be that Soot is a bytecode analyzer, therefore its output is close to the output of a dynamic analyzer. This also means that our work could be easily extended by including such dynamic call graphs.

Murphy et al. [24] carried out a study about the comparison of five static call graph creators for C in 1996. The outputs of the analyzers were compared to a baseline call graph created by the GCT test coverage tool, which was based on the GNU C compiler. They identified significant differences in how the tools handled typical C constructs, like macros. Mapping the graphs was made more complicated depending on which files were involved in the analysis. They applied a filtering mechanism to solve this. Other difficulties of the matching mechanism were not discussed, although C functions are clearly identified by their name only.

Naturally, it is possible to compare two graphs by considering only the structure of the graph without label information. Many similarity measures are based on iterative calculations. These methods repeatedly refine an estimated initial similarity value of the graph nodes by using an update formula. The update formulas consider the similarity of the edges and the neighboring nodes. When a termination condition is met the iteration finishes and a similarity matrix is produced. Nikolič proposed an iterative solution [25] called neighbor matching that addresses the insufficiencies of the previously existing approaches [5, 23, 39, 16]. An in - and out - similarity is defined for the update formula. To determine the in-similarity an optimal matching of in-neighbors has to be constructed. The calculation is analogous in case of the out-similarity. The introduced node similarity calculation was evaluated on isomorphic subgraph matching and on a social network, and concluded that it is more accurate than the previous approaches. Nikolič provided a C++ implementation of the proposed method. In Section 5 we compare the results of this topology based graph similarity tool with our pairing mechanism.

3 Analyzed tools

We studied numerous static analyzer tools for Java to decide whether they could generate – or could be easily modified to generate – call graphs. We aimed for widely available, open-source programs from recent years, which could analyze complex, real-life Java systems. The diversity of the tools was another important aspect of our selection criteria. We involved tools that provide a direct interface for call graph creation, whilst, in other cases, the graph had to be extracted directly from the inner representation of the analyzer. The investigated analyzers can also be categorized by whether they work on source or byte code, which, of course, affects their results. Most of the tools are command line based, although an Eclipse plug-in based solution was also examined. The selected analyzers support several call graph-creation algorithms, which greatly influences the characteristics, the accuracy and the size of the generated graph. It is the application that determines what type of call graph is the most useful, sometimes a small and less accurate call graph is better, while, in other cases, a large and precise one is needed. The goal of this paper is not to compare the output of these call graph-builder algorithms, but to pair the corresponding graph sections, which is the basis for further comparative studies. Therefore, we considered the algorithm only as an attribute of the given tool.

Table 1 summarizes the most important properties of the examined call graph builder tools. The grey lines correspond to two of the discarded tools that we tested in more detail. In both cases, the reason for the exclusion was their lack of robustness. For example, JavaParser [18] did not give enough information to reconstruct the caller-callee relationships between compilation units without major development. Call Hierarchy Printer[6] (CHP) failed to finish the analysis of projects larger than a few thousands lines of code. The selected tools, the analyzed sources, and the results are available as an online appendix[2].

The description of the six tools that were selected for the comparison is presented below.

Table 1: Summary of examined call graph creator tools

	version	**maintained**	**input**	**robustness**	**built-in call-graph construction**	**multiple algorithms available**
JCG	commit da81eeb on Oct 24 2018	✓	byte code	✓	✓	×
SPOON	7.0.0	✓	source code	✓	×	×
WALA	1.5.1	✓	byte code	✓	✓	✓
OSA	1.0.0	✓	source code	✓	×	×
Soot	3.2.0	✓	byte code	✓	✓	✓
JDT	Eclipse Oxygen.2 (4.7.2)	✓	source code	✓	×	×
CHP	commit 3316b4a on Mar 26 2015	×	both	×	×	×
JavaParser	3.5.16	✓	source code	×	×	×

3.1 Java Call Graph

The Java Call Graph (JCG) [17] is an Apache BCEL [3] based utility for constructing static and dynamic call graphs. It can be considered a small project as it only has one major contributor, Georgios Gousios, whose last commit (at the time of this writing) is from October, 2018. It supports the analysis of Java 8 features and requires a `jar` file as an input. A special feature of the analyzer is the detection of *unreachable*[3] code. As a result, the call graph does not include calls from code segments that are never executed.

3.2 SPOON

SPOON [27] is an open-source, feature-rich Java analyzer and transformation tool for research and industrial purposes. It is actively maintained, supports Java up

[2]`http://www.inf.u-szeged.hu/~pengoe/research/StaticJavaCallGraphs/`

[3]Unreachable code will never be executed as there is no control flow path to it from the entry point of the program.

to version 9, and while several higher-level concepts (e.g., reachability) are not provided "out of the box", the necessary infrastructure is accessible for users to develop their own. SPOON performs a directory analysis[4] of the source code and builds an AST-like metamodel, which is the basis for these further analyses and transformations. We extracted the call graph of our project by traversing this internal representation and collecting every available invocation information. The library is well-documented and provides a visual representation of its metamodel, which helped us in thoroughly studying its structure.

3.3 WALA

WALA [36] is a static and dynamic analyzer for Java bytecode (supporting syntactic elements up to Java 8) and JavaScript. Originally, it was developed by the IBM T.J. Watson's Research Center; now it is actively developed as an open-source project. WALA has a built-in call graph generation feature with a wide range of graph building algorithms. We used the *ZeroOneContainerCFA* graph builder for our experiments, as it performs the most complex analysis. It provides an approximation of the Andersen-style pointer analysis [2] with unlimited object-sensitivity for collection objects. The generator has to be parameterized with the entry points from which the call graphs would be built. To make the results similar to the results of the other tools, we treated all the methods as entry points (instead of just the `main` methods). For other configuration options, we used the default settings provided in the documentation and example source codes.

3.4 OpenStaticAnalyzer

OpenStaticAnalyzer (OSA) [26] is an actively maintained, multi-language static analyzer framework developed by the Department of Software Engineering at the University of Szeged. It calculates source code metrics, detects code clones, performs reachability analysis, and finds coding rule violations up to Java 8. Other languages such as Python and C# are supported as well. Besides the directory analysis of the source code, OSA is also capable of wrapping the build system (*maven* or *ant*) of the project under examination. This can make the analysis more precise as generated files will be handled too. Similarly to the above mentioned SPOON implementation, we extracted the call graphs by processing the AST-like inner representation of OSA.

3.5 Soot

Soot [29] is a widely used language manipulation and optimization framework developed by the Sable Research Group at the McGill University. It supports analysis up to Java 9 and works on the compiled binaries. Although its official website[5] has

[4]The static analyser processes recursively every Java file in a given root folder
[5]`https://www.sable.mcgill.ca/soot/soot_download.html`

the latest release from 2012, the project is active on GitHub, from where we acquired the 3.2.0 release, which was the latest version then. Like WALA, Soot also has a built-in call graph creator functionality. For the analysis of library projects the CHA algorithm was used for call-graph construction, while in case of standalone projects we used the SPARK framework, which employs a points-to analysis algorithm.

3.6 Eclipse JDT

The Eclipse Java development tools (JDT) [10] is one of the main components of the Eclipse SDK [11]. It provides a built-in Java compiler and a full model for Java sources. We created a JDT based plugin for Eclipse Oxygen that supports even Java 10 code, to extract the call graph from the extensive, AST-like inner representation.

4 Refining the pairing mechanism

There are numerous elements that could cause differences in call graphs, as tools process language elements differently. In this section, we discuss what attempts we made to handle these differences and what were the benefits and downsides to each approach. In this article, the pairing mechanism is illustrated through the Apache Commons Math 3.6.1[6] project (208,876 KLOC). We are only using one project as an example, since our aim is to showcase the process itself, not to compare data. More analyzed projects are presented in the online appendix mentioned in Section 3.

4.1 Overview of process

The following four subsections correspond to the process of developing a unified representation for Java method names. Figure 1 provides an overview of this development process. It was previously stated that the call graph creator tools produce the graphs in slightly different formats. Therefore, we had to implement a specific graph loader for each tool to handle the aspects of its method naming convention. A basic name pairing (1.) was introduced to treat the fundamental differences of the representations. However, anonymous language elements needed extra consideration for which the anonymous transformation method (2.) was developed. As the figure indicates this heuristical approach is not part of the final approach. We found that the anonymous transformation method could be improved by using line information (3.). This introduced a challenge in the handling of generic source code elements which had to be dealt with (4.). No other Java language elements were identified that impaired the pairing mechanism.

[6]`http://commons.apache.org/proper/commons-math/`

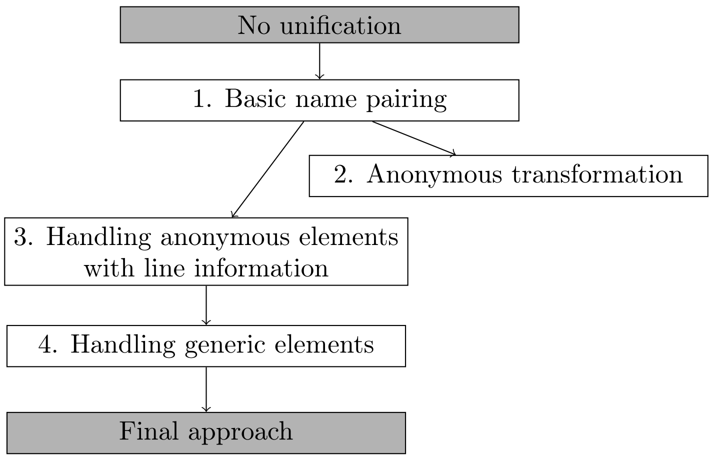
No unification
1. Basic name pairing
2. Anonymous transformation
3. Handling anonymous elements with line information
4. Handling generic elements
Final approach

Figure 1: Development process

4.2 Basic name pairing

In Java, methods can be distinguished by fully qualified names, which include the package name, the class name, the name of the method, and the list of the parameter types. The return value is not required for the identification, however, we encountered one case where it is indeed needed. According to the Java standard, overridden methods can differ in return type if the return-type-substitutability is satisfied, for example, the child class specializes the return type to a subtype. Some tools represent both the specialized and the not-specialized methods for a child class, although they only connect edges to one of them. Therefore, these rare cases could be easily detected.

Call graph comparison is based on identifying and matching the corresponding methods in each graph regardless of their representation. At first, we only used the method names produced by the static analyzers as basis of the method pairing. However, this was not enough because some fundamental features are represented differently, for example, some of the tools denote constructor methods with the class name, whilst others tag them with the name `<init>`. Therefore, we developed a common representation for the Java methods and as a first step of the comparison we transformed every call graph to this unified representation. Only the constructor methods, initializer blocks and other not-so-significant representational differences were subject to the name unification process. Instance initializer blocks are executed every time an instance of that class is created. They can be used to initialize class members. The Java compiler copies initializer blocks into every constructor. Therefore, initializer blocks can be used to share a block of code between multiple constructors. Byte code analyzer tools, such as WALA, represent initializer blocks as part of the constructor methods. However, source code analyzers such as SPOON represent the initialization blocks and the constructor methods with separate nodes for the given class. Our basic name pairing method aggregates the nodes of initializers blocks with every constructor of that class, making it pairable with the constructor methods found in the compared graph. This functionality can be turned off with a command line option, if the user wishes.

Figure 2 - 4 help in understanding the process of basic name pairing. Figure 2 shows a sample code containing constructors and initializer blocks. The reason we only included these two language elements in the sample code is because during basic name pairing only they require special consideration. The call graphs of the sample code are portrayed in Figure 3. These are produced by two of the tools, SPOON and WALA. The grey nodes belong to SPOON's graphs, the white ones belong to WALA's graph. As described in the previous paragraph, SPOON represents initializer blocks with separate nodes, while WALA treats them as part of the constructor methods, which causes a slight discrepancy between the two graphs. For this reason, during our pairing mechanism we aggregate the nodes of the initializer blocks with constructor nodes to ensure that none of them remain without a pair. A method name unification is also performed on each of the nodes. The results of the aggregation and the unification process can be seen in Figure 4. The borders of the rectangles indicate which nodes are paired between the two graphs. Two nodes are paired if their names match character by character.

```
class Test {
  Integer i;
  public Test(){}
  public Test(String s){}
  {
    i = new Integer(89);
  }
}
```

Figure 2: The basic name pairing in action: sample code

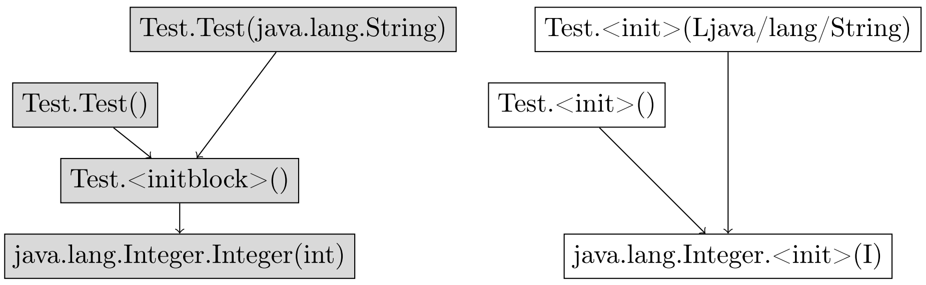
Test.Test(java.lang.String)
Test.Test()
Test.<initblock>()
java.lang.Integer.Integer(int)
Test.<init>(Ljava/lang/String)
Test.<init>()
java.lang.Integer.<init>(I)

Figure 3: The basic name pairing in action: input graphs

Table 2 summarizes the results of this initial attempt on the Commons Math project. The diagonal elements in bold show the number of different methods found

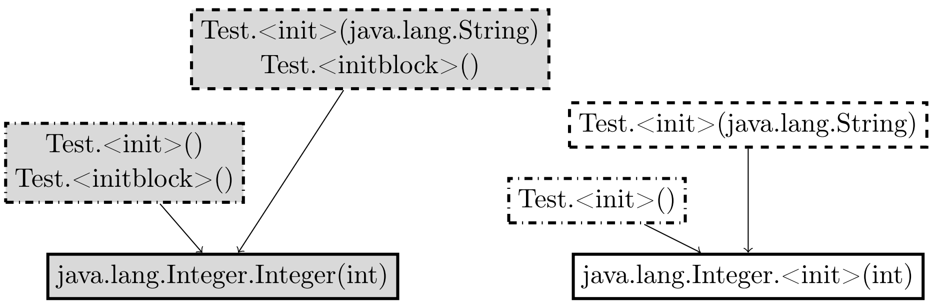
Test.<init>(java.lang.String)
Test.<initblock>()
Test.<init>()
Test.<initblock>()
java.lang.Integer.Integer(int)
Test.<init>(java.lang.String)
Test.<init>()
java.lang.Integer.<init>(int)

Figure 4: The basic name pairing in action: produced pairings

by each static analyzer tool. Every other cell in a row is a percentage that displays what percentage of the given tool's methods was found by the tool in the column.

Table 2: Results of the basic name pairing

	Soot	OSA	SPOON	JCG	WALA	JDT
Soot	**4,022**	51.24%	52.76%	57.46%	57.36%	50.97%
OSA	24.4%	**8,446**	100,00%	96.39%	80.88%	91.13%
SPOON	24.77%	98.5%	**8,551**	96.55%	81.24%	89.81%
JCG	23.22%	81.81%	83.24%	**9,951**	75.82%	75.98%
WALA	27.47%	81.33%	83.02%	89.83%	**8,399**	83.09%
JDT	21.41%	80.37%	80.43%	78.95%	72.87%	**9,577**

Looking at the table, it becomes apparent that the column of Soot contains quite low values. Soot found half of the methods compared to the other analyzers, therefore, its highest possible percentage is at about 50% - 60%. The reason for this discrepancy lies in the algorithmic differences between the tools, however, analyzing this is not the subject of the current paper.

4.3 Anonymous transformation

The basic name pairing cannot handle every Java language feature. One of them is the anonymous source code elements.

An anonymous class is an inner class without a name. It is useful when the programmer needs one instance of a class or interface with only certain overridden methods, so the actual subclass creation can be avoided. Lambda methods can be considered anonymous, however, most analyzers denote them with their interface name. Anonymous source code elements have a non-standardized, compiler generated name, meaning that static analyzers can name the same code element differently. Inner classes have a '$' sign in their name appended right after the name of the outer class. The '$' sign is followed by the name of the inner class.

In case of anonymous classes, a number is present after the '$' sign, however, the numbering is not consistent among the compilers and analyzer tools. Both global, project-wise numbering, and class level numbering is possible. The order of the numbering can also make a difference in the output of the tools. It is clear that our basic pairing approach that was introduced in the previous subsection is not sufficient for pairing anonymous code elements.

The transformation simply means that during the name unification process we replace the varying number after the '$' sign with a constant string. This means that multiple anonymous elements in a class will be aggregated into one, which is the explanation of smaller method numbers in the diagonal of Table 3. For example, if a class has multiple anonymous classes, all of them will be transformed for the unified anonymous class, causing a loss in the accuracy of the pairing. For projects that do not rely on anonymous classes very much - i.e. in a class there is at most one anonymous element - this heuristical approach is acceptable.

```java
class AnonymousTest{
  public void print(){
    //...
  }
}

class Test{
  public static void main(String args[]){
    AnonymousTest t1 = new AnonymousTest(){
      @Override
      public void print(){
        //...
      }};

    AnonymousTest t2 = new AnonymousTest(){
      @Override
      public void print(){
        //...
      }};
  }
}
```

Figure 5: Sample code containing two anonymous classes

Figure 5 shows an example code containing two anonymous classes. Figure 6 portrays a call graph constructed from this code (left side) and the class-level aggregation of anonymous code elements after the anonymous transformation (right side). If this code snippet is part of a larger project, then the two anonymous classes may not have the same numbering in all of the produced call graphs. However, after the aggregation the unified anonymous nodes can be paired.

Table 3, that is constructed similarly to Table 2, shows the results of the method pairing improved with anonymous transformation. The green cells highlight those percentages that are higher compared to Table 2.

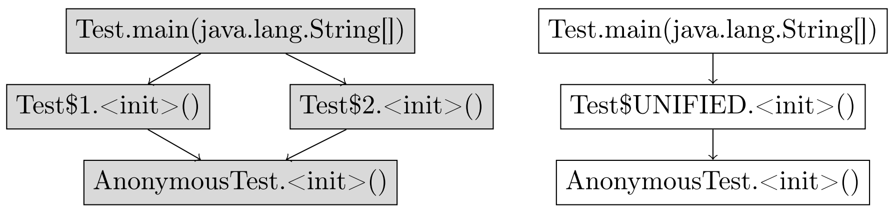

Figure 6: The process of anonymous transformation. The original graph is on the left, the transformed version is on the right side

Table 3: Results of the anonymous transformation

	Soot	OSA	SPOON	JCG	WALA	JDT
Soot	**3897**	51.78%	53.09%	56.58%	56.38%	54.68%
OSA	24.23%	**8329**	100,00%	96.96%	81.23%	94.38%
SPOON	24.54%	98.81%	**8406**	97.25%	81.68%	93.42%
JCG	22.56%	82.61%	83.94%	**9776**	75.39%	78.71%
WALA	26.75%	82.39%	83.97%	89.75%	**8212**	86.79%
JDT	22.58%	83.3%	83.46%	81.54%	75.52%	**9437**

4.4 Employing line information

We concluded that the previous heuristical solution should and could be improved, so that anonymous source code elements could be paired independently. It was a self-evident idea to include the line information in order to improve the accuracy of the method matching. However, we soon found out that the line information does not provide a perfect solution for the problems of method pairing because it is not as consistent among static analyzers as it was expected.

One obvious difficulty is that some of the tools process the source files themselves, while others work on the already compiled class files. Source code analyzers provide line information to the beginning and end of the method declaration. Byte code analyzers give the line information for the first statement of the method in question. In case of an empty method, the line information of the ending of the method declaration is present. This difference can be overcome by interval testing. Moreover, not every method has line information because they are compiler generated or they are part of the Java library. In other cases, only some of the tools can provide line information for a method. In addition to these difficulties, we realized that in a few cases tools provide the line information of the beginning of the class definition for some methods. These are inherited methods, whose return type was specialized by the child class (as it was described in the beginning of Section 4.2). As a consequence, some methods that certainly differ have the same line information.

Seeing these difficulties, it is clear that we cannot rely on line information blindly, because it would misguide the pairing mechanism. Therefore, the usage of line information was restricted only for anonymous and generic source code elements, whilst, for traditional methods, the name-wise pairing was used. The challenge of anonymous elements has already been discussed. In their case, we used only the line information for matchmaking. Generic elements raised a new type of issue that is introduced in the next section.

Figure 7 depicts the pseudocode of the line information based anonymous pairing. The condition on line 23 is true if the package names of the two methods are equal, and the class and method names only differ after the $ sign (anonymous

```
1   /*
2   The method returns true if two nodes (methods) considered equal
        according to their line information
3   */
4   func checkLineInfo(m1,m2)
5     if m1.endLine NOT valid
6       m1.endLine=m1.startLine
7     end
8
9     if m2.endLine NOT valid
10      m2.endLine=m2.startLine
11    end
12
13    return (m1.startLine <= m2.startLine AND m1.endLine >= m2.endLine
          ) OR (m1.startLine >= m2.startLine AND m1.endLine <= m2.
          endLine)
14  end
15
16  /*
17  This method returns true if the given nodes (methods) are
        considered equal otherwise false
18  */
19  func anonymousPairing(m1, m2) //m1 and m2 are anonymous methods
20  begin
21    isEqual=false
22    if line-info available
23      if m1 differs m2 only in anonymous names
24        if checkLineInfo(m1, m2)
25          for i in m1.parameterCount
26            if m1.param[i] NOT equals m2.param[i]
27              return false
28            end
29          end
30          isEqual=true
31        end
32      end
33    end
34    return isEqual
35  end
```

Figure 7: The pseudocode of the line information based anonymous pairing

part). On line 24 we check if the two methods are the same according to the line information. Byte code analyzers provide the line information of the first statement of the examined method, while source code analyzers detect the method declaration. Moreover, some tools consider the comment in front of a method as part of its declaration, making the line-information of a method even more diverse. Method `checkLineInfo` depicts how the interval checking of the line information of two methods is done. If a tool does not provide end line number for the methods it will be initialized with the number of the start line. The equality of the parameter lists is examined on line 25-29, although, it is not necessary if we consider the provided line information valid.

Table 4 shows the improvement of results compared to the basic name-wise pairing that is summarized in Table 2. In case of Soot and WALA we can see a slight decrease in the number of methods. It is because these tools - erroneously - provided the same line information for some anonymous methods, therefore, they could not be handled separately. The approach best improved the pairing of the JDT as this is the most reliable tool for providing line information.

Table 4: Results of the transformation based on line information (anonymous)

	Soot	OSA	SPOON	JCG	WALA	JDT
Soot	**3,976**	51.84%	53.37%	56.97%	56.87%	54.83%
OSA	24.4%	**8,446**	100,00%	96.39%	80.88%	93.27%
SPOON	24.77%	98.5%	**8,551**	96.55%	81.24%	92.13%
JCG	22.76%	81.81%	83.24%	**9,951**	75.19%	77.96%
WALA	27.12%	81.95%	83.65%	89.76%	**8,336**	86.32%
JDT	22.76%	82.26%	82.5%	81.01%	75.14%	**9,577**

4.5 Strategy for handling generic elements

As the previous subsection indicated, generic source code elements need extra consideration during the pairing mechanism. Java generic classes and methods were introduced in JDK 5.0. They allow programmers to specify a set of methods and a set of types with only one method and class declaration, respectively. A single generic method can be called with arguments of various types. One important trait of generic classes is that they can be parameterized differently during instantiation. Generic type parameters can be bounded, which restricts the types that are allowed to be passed.

Static analyzers represent generic elements in the call graph in various ways. Table 5 shows the diversity of representations after the method name unification. It can be seen that the tools represent them with varying accuracy. Sometimes generic parameters are represented by the prototype that is present in the declaration, optionally involving the type restriction too (e.g., SOOT). In other cases, the type of the actual parameter is used, that is, the tool represents the same generic method with multiple nodes but with differing generic parameters.

Table 5: Various representations of a generic method

Declared method	`<T, K extends Child2> Generic2<Child2, Generic1<Child2> > methodGen(K c, Generic1<K> g, Class<?>...objects)`
Usage	`methodGen(new Child2(), new Generic1<Child2>(), Integer.class)`
Representations	
JCG	`methodGen(Child2,Generic1,java.lang.Class[])`
WALA	`Generic2 methodGen(Child2,Generic1,java.lang.Class)`
OSA	`Generic2 methodGen(Child2,Generic1,java.lang.Class)`
Soot	`Generic2 methodGen(Child2,Generic1,java.lang.Class[])`
SPOON	`methodGen(K extends Child2,Generic1,java.lang.Class[])`
JDT	`Generic2<Interface,Generic1> methodGen(K,Generic1<Interface>, java.lang.Class<?>[])`

The ideal solution would be to pair the corresponding generic methods to each other, but because of the variety of the notations, matching them only through the basic pairing process caused inaccuracies. Although the package, class, and method names are the same, even the number of parameters are the same, the type of the parameters can differ. Unlike in the case of anonymous methods, it is not always possible to decide whether a generic method is generic or not, based on its name alone. Therefore, the line information is needed to decide if two methods with the same name and number of parameters correspond to the same generic method. If the line information is the same as well, then the two nodes apply to the same generic method. This heuristical assumption has a threat to validity if the tool provides false line information. What is more, the pairing is not possible if no line information is given. The pseudocode of the pairing algorithm for generic elements is shown in Figure 8. The `checkLineInfo` method is the same as in Figure 7. The heuristical method for matching the generic parameters is on line 11-15. As our pairing approach currently does not utilize the class hierarchy of the analyzed project, only a conservative matching is allowed with generic wildcards such as `K,T,E...` and with `java.lang.Object`, which is a base class for every other class. The reason for this conservative solution is that we want to avoid accidental matching of overridden methods. The manual validation proved this approach to be sufficient. Combining line information with generic elements caused another type of problem, which is summarized in Figure 9.

Figure 9 shows two static analyzers, Tool 1 and Tool 2 (denoted by grey ellipses) and methods they detected during analysis (denoted by white ellipses). The analyzed source code contains a generic method, `<T> void goo(T t)` and two normal methods, `void foo(int a, int b)` and `void foo(int a)`. Tool 1 represents `goo` in the call graph only with one node. As there is no restriction on the type, the tool denotes the parameter type as an `Object`. In contrast to this, Tool 2 associates three nodes to method `goo` based on the type of the actual parameters it was called with. All `goo` nodes have the same line information. The matching of the `foo`

```

/*
This method returns true if the given nodes (methods) are
     considered equal otherwise false
*/
func anonymousPairing(m1, m2)
begin
  isEqual=false
  if line-info available
     if checkLineInfo(m1, m2)
       for i in m1.parameterCount
         if (m1.param[i] equals m2.param[i] OR
            m1.param[i] is java.lang.Object OR
            m2.param[i] is java.lang.Object OR
            m1.param[i] is GENERIC_WILDCARD OR
            m2.param[i] is GENERIC_WILDCARD)
           isEquals=true
         else
           isEquals=false
           break
         end
       end
     end
  end
  return isEqual
end
```

Figure 8: The pseudocode of the line information based generic pairing

methods is obvious, as both of them are represented with one node each. This is not the case with the pairing of method goo. The left side of the figure shows a possible matching of the nodes of Tool 1 to the nodes of Tool 2. The pairing of goo is denoted with a dashed line, as other matches would be possible if it was allowed to pair one method to multiple others. The right side of the figure shows the opposite direction: the matching of the nodes of Tool 2 to the nodes of Tool 1. It can be seen that all goo nodes will be paired to the same node in the graph of Tool 1, because there is no other option. As a consequence, there is asymmetry in the results depending on the direction from which we start pairing the nodes.

This described pairing anomaly can be resolved in multiple ways. One solution is to use the results as they are, without any further modifications. This approach emphasizes the differences between the tools' capabilities. Another option is to keep only those node-matchings that can be found from both directions. Finally, we can collect every possible pairing from both directions and put them into a union. The union pairing was the solution we decided to use. Table 6 summarizes the results of this approach. The structure of the table is similar as before, the green cells highlight the higher percentages compared to Table 4. There is a decrease in the number of methods because we counted the corresponding generic methods as one.

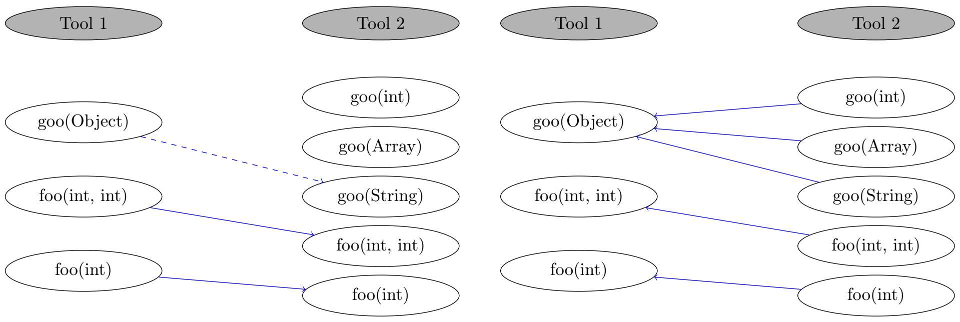

Figure 9: Pairing anomaly

Table 6: Results of the transformation based on line information (anonymous and generic elements)

	Soot	OSA	SPOON	JCG	WALA	JDT
Soot	**3,969**	51.85%	53.39%	56.97%	56.97%	55.1%
OSA	24.38%	**8,441**	100,00%	96.39%	80.9%	95.53%
SPOON	24.75%	98.5%	**8,545**	96.55%	81.26%	94.36%
JCG	22.76%	81.93%	83.36%	**9,928**	75.18%	79.97%
WALA	27.16%	81.99%	83.69%	89.74%	**8,332**	88.02%
JDT	23.1%	85.83%	86.07%	84.52%	77.05%	**9,547**

4.6 Remaining differences

Table 6 shows that we could not achieve 100% pairing for the tools, a significant number of nodes remained unmatched. We manually investigated the root causes for this, in order to find possible ways to improve our pairing mechanism. However, our in-depth examination revealed that most of the unmatchings cannot be resolved. The reasons for the differences can be categorized as follows:

- A tool detects a method type that other tools do not represent, therefore some nodes will not have images in the other tools' graph.
  - Soot represents much more static initializer nodes then other tools.
  - WALA places more Java library nodes and calls into the generated call graphs.
  - SPOON represents Java static field initialization with a unique node.
- The methods that can be found in the bytecode slightly differ from the methods of the source code.

  - Bytecode analyzers (JCG, WALA, Soot) find compiler generated `access$XXX` methods, which cannot be paired with improper line information.
  - Source code analyzers detect only default constructors for `Enum` classes. Byte code analyzer tools represent the valid constructors with an `Integer` and a `String` parameter.
  - In the compiled sources, the methods of inner classes have an extra parameter, a reference to the outer class. This parameter is missing from the findings of the source code analyers.
- There are algorithmic differences in the handling polymorphic calls.
  - Tools that employ less accurate analysis techniques represent more interface and base class methods instead of the methods of the subclasses.
  - JCG represents inherited methods as the method of the child class, while other tools represent them as part of the base class.
- Methods that do not have at least one method call are excluded. OSA and SPOON have this feature.
- Line information for anonymous and generic methods is missing.

We concluded from our findings that our pairing mechanism could only be improved with more reliable line information.

4.7 Edge similarity

Based on the implemented node pairing mechanisms, the pairing of the edges was also performed. Two edges are considered to be a pair if their endpoints are matched with each other. If one or both nodes of an edge are unmatched, then the edge itself is considered to be pairless too. This subsection discusses how the improvement of the node pairing affects the number of edges that can be paired with each other.

Table 7 and Table 8 present the call edge comparison results of the Commons Math project. Their structure is similar to the previous tables': the diagonal elements contain the number of calls detected by each tool, while every other cell in a row shows how many percent of the tool's calls were found by the tools in the columns. Table 7 corresponds to the basic name pairing mechanism and Table 8 shows the results achieved by using our final approach. Higher percentages are highlighted with green. A slight decrease can be observed in the number of call edges. As Table 2 and Table 6 show, some of the nodes were aggregated, and, because of this, a few duplicated edges were eliminated.

As expected, there are improvements in the number of successfully paired call edges, although the change is not really significant. Even if we take into account that there are possibly pairable methods, there are considerably low pairing ratios. This suggests that there are vital differences in the topology of the call graphs as well. The sampling of the unmatched call edges supports this assumption, however, a more in depth examination is needed to make further conclusions.

Table 7: Edge similarity using the basic name pairing method

	Soot	OSA	SPOON	JCG	WALA	JDT
Soot	**28,542**	12.46%	12.72%	14.01%	13.39%	11.57%
OSA	17.73%	**20,059**	99.83%	88.92%	58.03%	83.25%
SPOON	17.71%	97.67%	**20,501**	87.70%	57.98%	82.28%
JCG	17.52%	78.14%	78.77%	**22,826**	53.18%	66.94%
WALA	23.36%	71.14%	72.65%	74.19%	**16,363**	62.08%
JDT	17.1%	86.52%	87.4%	79.16%	52.63%	**19,302**

Table 8: Edge similarity using the final approach

	Soot0	OSA0	SPOON0	JCG0	WALA0	JDT0
Soot0	**28,485**	12.48%	12.74%	13.72%	13.28%	12.14%
OSA0	17.72%	**20,057**	99.83%	88.9%	58.03%	90.78%
SPOON0	17.7%	97.67%	**20,499**	87.68%	57.98%	89.82%
JCG0	17.13%	78.14%	78.77%	**22,817**	52.83%	72.29%
WALA0	23.18%	71.33%	72.84%	73.86%	**16,319**	65.23%
JDT0	17.92%	94.4%	95.46%	85.51%	55.19%	**19,288**

5 Comparison with a topology-based algorithm

In this Section we describe a comparison with the neighbor matching algorithm introduced in Section 2. Our goal was to study how a neighborhood-based algorithm performs in terms of accuracy and computational time compared to our approach.

5.1 Utilizing the topology-based method

We downloaded the C++ implementation [7] of Nikolič's work [25]. Only output formatting modifications were applied. We transformed the call graphs that were built by the six call graph creator tools so the iterative tool could take them as an input. The iterative tool requires only the call edge information, no node labeling is needed.

Like other iterative graph similarity algorithms, this one also produces a similarity matrix over the nodes of the compared graphs. Nikolič's innovation was the normalization of the similarity values between 0-1, so that the higher values indicate greater similarity. We computed and processed the similarity matrices of the examined projects for each call graph creator tool pair.

[7]`http://www.matf.bg.ac.rs/~nikolic/software.html`

5.2 Evaluation of results

To interpret the similarity values as pairings of nodes we searched for the maximum values in each row and column. Although this seems like a straightforward solution the similarity values were rather noisy, meaning that in many cases there were multiple similarity values around the maximum of a row or a column. Let us consider the similarity matrix of Soot and SPOON produced from their call graphs for the Commons Math project. If we pick a method by random there is a high chance that its pair defined by the maximum will be a noisy result. For example, in case of the `org.apache.commons.math3.util.FastMathLiteralArrays.loadExpFracB()` method which was detected by SPOON and has valid line information the highest similarity value is around 0.6. This corresponds to the following method: `org.apache.commons.math3.transform.FastFourierTransformer$MultiDimensionalComplexMatrix.<init>(java.lang.Object)`. It is clear that this pairing is invalid. To reduce the tremendous noise, we decided to take a pairing into consideration only when it is supported by both the column and row point of view, meaning that the value is both a column and a row maximum (ceratin matchings). If we examine the previous Soot-SPOON comparison this way we reduced the number of pairings reported only by the iterative approach from 12255 to 252. As it can be calculated from Table 6, our attempt detects 2120 pairings in the SPOON - Soot call graph comparison. There are 77 matches of the iterative tool for the SPOON - Soot comparison, which were also detected by our algorithm. If we consider the uncertain maximums as well, this number is 239. These matches were found valid, meaning that out of the 252 certain pairs of the Soot - SPOON comparison about 30% is valid.

Our main interest was to analyze the validity of pairings detected only by the iterative method. There were 2100 unique pairings out of the 15 pairwise comparisons of the 6 call graphs created for the Commons Math project. The manual investigation showed that 2036 of them were invalid, whilst 64 were valid, which is about 3%. The valid matchings had a specific characteristic. All of them were generated constructors of anonymous classes. Byte code analyzer tools represent these constructors with precise parameter lists containing references to the outer class and to the local fields used in the body of the anonymous class as well, while source code analyzers detect only the parameters that can be found in the sources. Naturally, without proper line information and with differing parameter lists, our node pairing mechanism is doomed to fail on these type of methods. The error could be resolved if the source code position of these constructor methods would be associated at least with the declarataion of the anonymous class, and by loosening our requirement for entirely equal parameter lists. It has to be noted that even the manual validation failed in a very few cases, when there was no line information for one member of the pair and the outer class contained multiple anonymous classes.

5.3 Summarization

The manual investigation showed that the results are similar for the other projects as well. In case of the Joda-Time project[8], only 3 pairings were valid out of 754. We concluded from our findings that on average the validity of the pairings that are found only by the iterative method less than 10%. Section 4.7 indicates that there are significant differences in the number and type of detected call edges, which can be a reason for the noisiness of this topology-based method.

The comparison with a topology-based method revealed a very specific weak point of our pairing mechanism. Although the problem is limited to a little subset of the nodes, it still has to be addressed in the future. If the static analyzer tools would provide line information for these anonymous initializers, for instance by associating them with the position of the declaration of the anonymous class, our approach could pair them. It would be a straightforward idea to combine our approach with this iterative method to resolve the described problem. However, the expansion would not be trivial, especially if we consider that our pairing mechanism took 12 minutes, while the iterative algorithm finished the Commons Math project over 17 hours.

Despite the problems of anonymous constructors, this comparison assured us that we find no matches that a neighbor-based algorithm would not and we miss a high percentage of noisy results that the iterative method reports. Moreover, the computation time of our pairing mechanism does not scale with the size of the input graphs as badly as that of the iterative method. On small sample graphs both implementations finish within seconds, however in case of projects with a few thousands lines of code the iterative method needs hours compared to the couple of minutes that our approach requires.

6 Conclusions

In the future, we plan to compare the capabilities of static call graph creator tools. This could be done by comparing what methods and calls are present in the generated call graphs. If the nodes of the call graphs are matched, then comparing the calls is a straightforward task. That is the reason why we paid so much attention to the unifying process of the methods. This paper was a necessary preliminary work for the upcoming quality comparison of the tools.

We collected and, where necessary, modified six Java static analyzer tools to generate call graphs for multiple large projects. By investigating the resulting graphs, we realized that the unification of method names is needed, in order to be able to match the corresponding nodes to each other. The unification process - and hence the pairing mechanism - has been refined in several steps. We highlighted two common language elements, the anonymous and generic methods that needed careful consideration and made the improvement of the process necessary. Multiple solutions were proposed. One heuristical - but less accurate - approach for

[8]`https://github.com/JodaOrg/joda-time`

anonymous elements is the anonymous transformation. However, with line information they could be handled better, along with the generic code elements. We performed a manual validation of the different pairing strategies on a sample code, containing all features of Java 8. The source and the results are available in the online appendix. The results of the large projects were also manually investigated. Our solution was compared to a topology based node pairing algorithm as well.

In our final solution, we used the basic name-wise pairing for normal methods, line information-based pairing for anonymous methods and a combined solution for generic methods. In this combined solution, if two methods have the same package, class and method name, have the same number of parameters and have the same line information, it is assumed that they correspond to the same generic method declaration. The analyzers may represent the same generic method with different number of nodes in their call graphs. This asymmetry was solved by collecting every possible pairing between these nodes.

The manual validation proved that better pairing could be achieved if we could acquire more accurate line information of the methods. However, the reason for matchless nodes lies in the differences of the static call graph creators themselves, therefore, the matching of some nodes is impossible.

References

[1] Ahmad Bhat, Sajad. A practical and comparative study of call graph construction algorithms. *IOSR Journal of Computer Engineering*, 1:14–26, 01 2012. DOI: `10.9790/0661-0141426`.

[2] Andersen, Lars Ole. Program analysis and specialization for the C programming language. Technical Report May, 1994. 10.1.1.109.6502.

[3] Apache BCEL Home Page.
`https://commons.apache.org/proper/commons-bcel`.

[4] Bacon, David F. and Sweeney, Peter F. Fast Static Analysis of C++ Virtual Function Calls. *SIGPLAN Not.*, 31(10):324–341, October 1996. DOI: `10.1145/236338.236371`.

[5] Blondel, Vincent, Gajardo, Anahi, Heymans, Maureen, Senellart, Pierre, and Van Dooren, Paul. A measure of similarity between graph vertices: Applications to synonym extraction and web searching. *SIAM Review*, 46:647–666, 12 2004. DOI: `10.2307/20453570`.

[6] Call Hierarchy Printer GitHub Page.
`https://github.com/pbadenski/call-hierarchy-printer`.

[7] Champin, Pierre-Antoine and Solnon, Christine. Measuring the similarity of labeled graphs. In Ashley, Kevin D. and Bridge, Derek G., editors, *Case-Based Reasoning Research and Development*, pages 80–95, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[8] Christodorescu, Mihai and Jha, Somesh. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[9] Dean, Jeffrey, Grove, David, and Chambers, Craig. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In Tokoro, Mario and Pareschi, Remo, editors, *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, pages 77–101, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[10] Eclipse JDT Home Page.
`http://www.eclipse.org/jdt/`.

[11] Eclipse JDT Home Page.
`www.eclipse.org/eclipse/`.

[12] Eshera, M. A. and Fu, K. A graph distance measure for image analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-14(3):398–408, May 1984. DOI: `10.1109/TSMC.1984.6313232`.

[13] Feng, Yu, Anand, Saswat, Dillig, Isil, and Aiken, Alex. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 576–587, New York, NY, USA, 2014. ACM. DOI: `10.1145/2635868.2635869`.

[14] Grove, David and Chambers, Craig. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, November 2001. DOI: `10.1145/506315.506316`.

[15] Grove, David, DeFouw, Greg, Dean, Jeffrey, and Chambers, Craig. Call Graph Construction in Object-oriented Languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 108–124, New York, NY, USA, 1997. ACM. DOI: `10.1145/263698.264352`.

[16] Heymans, Maureen and Singh, Ambuj K. Deriving phylogenetic trees from the similarity analysis of metabolic pathways. *Bioinformatics*, 19 Suppl 1:i138–46, 2003.

[17] Java Call Graph GitHub Page.
`https://github.com/gousiosg/java-callgraph`.

[18] JavaParser - for processing Java code Homepage.
`https://javaparser.org/`.

[19] Koutra, Danai, Parikh, Ankur, Ramdas, Aaditya, and Xiang, Jing. Algorithms for graph similarity and subgraph matching. 02 2019.

[20] Lhoták, Ondrej. Comparing call graphs. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 37–42, 2007.

[21] Lhoták, Ondřej and Hendren, Laurie. Context-Sensitive Points-to Analysis: Is It Worth It? In Mycroft, Alan and Zeller, Andreas, editors, *Compiler Construction*, pages 47–64, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[22] Macindoe, O. and Richards, W. Graph comparison using fine structure analysis. In *2010 IEEE Second International Conference on Social Computing*, pages 193–200, Aug 2010. DOI: `10.1109/SocialCom.2010.35`.

[23] Melnik, Sergey, Garcia-Molina, Hector, and Rahm, Erhard. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. pages 117 – 128, 02 2002. DOI: `10.1109/ICDE.2002.994702`.

[24] Murphy, Gail C., Notkin, David, Griswold, William G., and Lan, Erica S. An Empirical Study of Static Call Graph Extractors. *ACM Trans. Softw. Eng. Methodol.*, 7(2):158–191, April 1998. DOI: `10.1145/279310.279314`.

[25] Nikolić, Mladen. Measuring similarity of graph nodes by neighbor matching. *Intell. Data Anal.*, 16(6):865–878, November 2012. DOI: `10.3233/IDA-2012-00556`.

[26] OpenStaticAnalyzer GitHub Page.
`https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer`.

[27] Pawlak, Renaud, Monperrus, Martin, Petitprez, Nicolas, Noguera, Carlos, and Seinturier, Lionel. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015. DOI: `10.1002/spe.2346`.

[28] Sable *J Home Page.
`http://www.sable.mcgill.ca/starj/`.

[29] Sable/Soot GitHub Page.
`https://github.com/Sable/soot`.

[30] Sanfeliu, A. and Fu, K. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(3):353–362, May 1983. DOI: `10.1109/TSMC.1983.6313167`.

[31] Sundaresan, Vijay, Hendren, Laurie, Razafimahefa, Chrislain, Vallée-Rai, Raja, Lam, Patrick, Gagnon, Etienne, and Godin, Charles. Practical virtual method call resolution for java. *SIGPLAN Not.*, 35(10):264–280, October 2000. DOI: `10.1145/354222.353189`.

[32] Tip, Frank and Palsberg, Jens. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 281–293, New York, NY, USA, 2000. ACM. DOI: `10.1145/353171.353190`.

[33] Tversky, Amos. Features of similarity. *Psychological Review*, 84(4):327–352, 1977. DOI: `10.1037/0033-295X.84.4.327`.

[34] Ullmann, J. R. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976. DOI: `10.1145/321921.321925`.

[35] Wagner, Tim A., Maverick, Vance, Graham, Susan L., and Harrison, Michael A. Accurate static estimators for program optimization. *SIGPLAN Not.*, 29(6):85–96, June 1994. DOI: `10.1145/773473.178251`.

[36] WALA Home Page.
`http://wala.sourceforge.net/wiki/index.php/Main\_Page`.

[37] Weiser, Mark. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[38] Wicker, Nicolas, Nguyen, Canh Hao, and Mamitsuka, Hiroshi. A new dissimilarity measure for comparing labeled graphs. *Linear Algebra and its Applications*, 438(5):2331 – 2338, 2013. DOI: `10.1016/j.laa.2012.10.021`.

[39] Zager, Laura A. and Verghese, George C. Graph similarity scoring and matching. *Applied Mathematics Letters*, 21(1):86–94, jan 2008. DOI: `10.1016/j.aml.2007.01.006`.