UNIVERSITÀ DEGLI STUDI DI CATANIA
DIPARTIMENTO DI MATEMATICA E INFORMATICA

DOTTORATO DI RICERCA IN MATEMATICA E INFORMATICA
XXX CICLO

DATA DEPENDENCE AND TAINT ANALYSIS TO FOSTER SOFTWARE EVOLUTION

Andrea Francesco Fornaia

Tesi di Dottorato

Tutor: Chiar.mo Prof. Emiliano Tramontana
Coordinatore: Chiar.mo Prof. Giovanni Russo

To my mother and my brother

...

Special thanks go to Prof. Emiliano Tramontana and Prof. Giuseppe Pappalardo without their patient guidance this work would have not been possible

ABSTRACT

Modern software development prefers fast and frequent changes rather than a detailed and fixed design. However, the continuous evolution of a software system can lead to a deterioration of its structural quality, making the code harder to understand and maintain.

This thesis presents novel approaches based on data dependence and taint analysis to assist code inspection and automatically suggest complex refactoring opportunities, thus achieving modularity.

Automatic tools are also proposed, which leverage API calls and data flows between different parts of the application to automatically give insights on the responsibility of each component. Then, these tools suggest a better decomposition of code inside concern cohesive modules, thus enhancing code modularity.

Along with modularity, the proposed approaches and tools are effective to deal with the issues of platform dependence and data security. On the one hand, platform dependence can make porting difficult when spread widely on the application; on the other hand, one may need to guarantee code robustness against data leaks. Both issues may force the code to be modified, thus evolved.

Both these aspects related to software evolution have been addressed by using the same concepts adopted to enhance code modularity, thus giving effective tools helping the developer in such issues.

Furthermore, the proposed support can be used both during the development of new software components or to analyse and improve legacy applications.

Contents

Abstract ii

1 Introduction **1**
1.1 The role of quality metrics 2
1.2 Refactoring the code for quality improvement 3
1.3 Refactoring guided by metrics 4
1.4 Suggest complex refactoring opportunities 5
1.5 The proposed approach 6
1.6 Thesis structure 8
1.7 Published Papers 10

2 Background and Related Work **12**
2.1 Data flow and data dependence analysis 13
2.2 Code portability 25
2.3 Concern tagging 27
2.4 Automate refactoring opportunity detection 30
2.5 Suggest *Extract Method* opportunities 33
2.6 Data leak detection in Android applications 40
2.7 Conclusions 44

3 Platform Dependence and Portability assessment **45**
3.1 Assessing code portability 45
3.2 Limits of the traditional approaches 47
3.3 Taint Analysis to assess platform dependence 48
3.4 Framework implementation 56
3.5 Experiments 58

3.6 Conclusions 62

4 Concern Tagging and Refactoring **63**
4.1 Concern Tagging 63
4.2 DeDuCT 65
4.3 Implementation 73
4.4 Experiments 75
4.5 Tagging examples 78
4.6 Conclusions 84

5 Suggest Complex Refactorings **85**
5.1 Assist *Replace Method with Method Object* 85
5.2 Determine fragment data dependences 88
5.3 Field Set selection 98
5.4 Experiments 104
5.5 Conclusions 113

6 Data Leak detection **114**
6.1 Preventing data leak in Android applications 115
6.2 Hybrid Taint Analysis 117
6.3 JADAL implementation 124
6.4 Experiments 132
6.5 Conclusions 135

7 Conclusions **136**

A Listings **138**
A.1 The `storageIM()` method before applying the *Replace Method with Method Object* 138
A.2 Refactored version of the `storageIM()` method by means of a *Method Object* 141
A.3 The *Method Object* after other two *Extract Methods* on `compute()` . 145

Chapter 1

Introduction

One of the most important qualities a software product must have, in order to preserve its usefulness over time, is the ability to *evolve* and adapt to emerging needs. Generally, there is a sudden need to meet new requirements by planning and adding functionalities to the software already developed, avoiding code decay.

All activities related to the *software evolution* after code release are grouped into the *maintenance* process. These activities are carried out to fix code errors (*corrective maintenance*), to adapt existing software to new environments, such as other execution platforms (*adaptive maintenance*), to add new features or to improve performances (*perfective maintenance*) [92]. It is easy to understand that the more *maintainable* the software itself, the cheaper the maintenance activities.

Software products are extremely complex systems whose constant evolution can lead to a maintainability deterioration, even considering an effective initial design. It happens that after several modifications, applied over time, the software can become more complex, making harder for developers to understand the responsibility of certain modules, i.e. making the code less reusable. The inability to *reuse* existing modules, thus leveraging time and money investments afforded, is in most cases due to the low quality of the internal structure of the software system, worsened during its evolution. Therefore, it is important to assess and possibly change the structure in order to improve modularity.

We may think that problems related to code modularity can be avoided by spending time and efforts in a good initial design. However, on the one hand widely adopted *agile* approaches [31], such as *extreme programming* [13] and *scrum* [86], are

software development processes in which fast and frequent changes to incorporate new features are preferred to a detailed and fixed design. On the other hand, it may be hard for the developer to choose how to separate the code into different modules since the early stages of development. Therefore, even if detailed, the initial design may be unsatisfying.

The lack of a detailed design, and the need for constant modifications to be applied, both call for automatic tools able to monitor code quality and assist the developer in taking it under control, also suggesting structural changes that can improve code modularity during software development.

1.1 The role of quality metrics

To manage the complexity of a software product, we need supports to evaluate the quality of code. In the literature, various metrics for object-oriented systems have been proposed, helping developers to understand if the desired quality has been achieved, for example, in terms of class *coupling* and *cohesion*. These metrics are useful indicators of the amount of work needed to understand, modify, or reuse classes, and help identifying those parts that need *preventive maintenance* changes, i.e. changes that are intended to improve the system structure to foster the introduction of new features.

The Chidamber and Kemerer (CK) [26] suite includes six metrics (WMC, DIT, NOC, CBO, RFC and LCOM) based on the static object code analysis. This suite, along with others related to object-oriented systems [18, 19], provides information on specific class characteristics, such as complexity, cohesion, and coupling.

The class *coupling* measure, given especially by CBO (*coupling between objects*), provides information about the dependence of a class from others in the system, and is given by the number of distinct classes with which it interacts through method calls or object instantiations. A high coupling value indicates a class that is difficult to reuse due to its high dependence on other parts of the system. It also indicates a complex class, difficult to understand and modify, and it is a clear indication of the need for design modifications to improve modularity, thus reuse capability.

A class has a low *cohesion* when its fields and methods are not well connected:

the LCOM metric (*lack of cohesion on methods*) of the CK suite was originally defined as the difference between the number of method pairs sharing no attribute and the number of pairs sharing at least one. Other LCOM definitions were also given in [18] though expressing the same concept: a class has a lack of cohesion when a few of its methods use the fields within the class.

However, this is not the only cohesion definition given so far. According to Hitz and Montazeri, with LCOM4 [57], a class is not cohesive when its methods perform a few reciprocal calls. The cohesion measure can also be extended by considering the strength with which the methods interact with each other, that is, the amount of data passed from one method to another through the call. This can be achieved by considering either the number of parameters (according to Lee, with ICH [66]) or even the corresponding type, so given a weight to the call according to the actual amount of exchanged bytes [79].

Usually, a low cohesion value can be a symptom of an inaccurate aggregation of methods and features within the class. This problem can be solved, for example, by moving some methods to other classes, or by splitting the class into smaller classes which show a higher cohesion.

In conclusion, existing literature proved that the design quality of a program can be measured and monitored using appropriate metrics. For each metric, guidelines for the interpretation of measured values are given. Thus, these metrics let the developer identify *which* classes need to be modified in order to improve modularity. However, metrics alone cannot tell us *how* these classes can be modified without altering the program behaviour.

1.2 Refactoring the code for quality improvement

In the literature there are several *refactoring* [44] techniques aiming at improving the structure of an object system, while taking care not to alter its functionality. The application of a refactoring technique is based on symptoms, which can be detected on the code, known as *"code smells"*, e.g. finding very long methods or duplicated code. These are essentially *qualitative* indicators, telling that *the code needs to be changed* [44].

Refactoring techniques *catalogues* give guidelines and motivations, which help

programmers to understand in which cases there is a need for a certain refactoring rather than another. On the one hand this makes such techniques sufficiently general to allow contextualization in a large and significant set of cases. On the other hand, this puts some effort on the developers, which have to choose how and where to apply a refactoring technique. They also have to evaluate the effects a technique will have on code quality. The effectiveness highly depends on the developer ability to take bad smells clues and act accordingly.

For example, if two classes are highly coupled due to a method using more features from another class than from the one it belongs to, or simply if moving a method to another class can make responsibilities clearer, what a developer can do is to apply the *Move Method* refactoring technique [44]. It consists in copying the method body from the original class to a new method inside the class it was closely coupled.

In case of a very long class, where too many responsibilities are put together, so that the methods as a whole make ambiguous the actual role of the class, it could be a good choice to apply the *Extract Class* refactoring technique, thus improving the reuse of individual features [44]. This technique consists in splitting the starting class into two distinct parts by partitioning the methods so that the role of the single classes is clearer.

1.3 Refactoring guided by metrics

The biggest obstacle developers can encounter while refactoring their code is certainly this: how can they understand *where* there is a need to apply a refactoring? As mentioned earlier, catalogues only refer to qualitative parameters and this makes refactoring difficult, especially in the case of large software, due to the large number of components and complexity it can have.

On the one hand, we have seen how the metrics can be used to make considerations on the design quality of individual classes, but on the other hand it is not trivial to define how these can *effectively* suggest *refactoring opportunities*. As a first step, metrics can assist in identifying system components that need to be refactored.

In general, the effect of a change on some of the software components is not easy

to understand when dealing with a real-size system because of the high number of components and interactions, whose quality could be influenced in some unintended way by the refactoring action. Furthermore, taking into account that such structural changes must preserve the behaviour of the system, applying these changes can be cumbersome, even considering the support of some software quality metrics and a catalogue of refactoring techniques.

Having automated tools capable of suggesting refactoring opportunities in large-scale systems is of paramount importance, since a non-assisted developer, having to reason on a large number of classes, may not understand where it is actually possible to apply a refactoring without compromising the behaviour of the system. In addition, the high number of components present in large systems makes manual exploration of all the actual possibilities of improvement unfeasible.

However, before we can talk about modularity assessment, *responsibility* of methods and classes have to be determined somehow. In fact, one of the main problem is still characterising the code with the *concern* it addresses. This problem is known as *concern identification* or *concern tagging*, that means mapping source code elements on to human oriented concepts or programming features.

Once responsibility of each code portion has been determined, modularity can be assessed, and e.g. refactoring techniques can be used to promote the *separation of concerns*, that means moving code related to different concerns in to different *concern cohesive* modules [17, 97, 107]. Separation of concerns has been addressed in different areas of software engineering, such as in *multi agent systems* to separate the algorithmic concern from the agent interaction concern [20], or in *software security engineering* to separate domain code from security code, by properly separating otherwise *crosscutting* security concern, thus making it easier to change the security architecture [114].

1.4 Suggest complex refactoring opportunities

In the contributions seen so far, metrics have been used to automatically suggest *single* refactoring techniques, or applying one single structural improvement operation at a time. However, as is often the case, refactoring operations to be made are more than one. These will be typically applied in close working sessions,

often in the case of new features to be added, to effectively introduce a *design pattern* [45] on a set of classes, or simply to improve the modularity of the code.

In other cases, we can even have a single class or method showing a high level of complexity so that a single refactoring operation would not be enough. For example, let us suppose that we have a very long and complex method that needs to be split in smaller, fine grained, and reusable methods. In this case, we will need more than one *Extract Method* refactoring, which consists in moving a sequence of statements from the original method to a new one, then replacing it with a corresponding method call, making the code extraction transparent to original clients.

However, to let the extraction of a code portion feasible, thus preserving the original behaviour and data flow, the developer needs to select the variables to become input and output values for the new extracted method. This is not always an easy task, and becomes difficult if more than one extraction for the same long method needs to be addressed. In this case, the *Replace Method with Method Object* is to be preferred, which may be considered as an extension of the Extract Method technique [12, 44]. In this case, the whole method will be replaced with a new class, the *Method Object*, which will have a field for each of the original method variable, so that there will be no problem for variable dependencies, thus letting multiple method extractions straightforward.

However, without any further analysis, this will lead to a class with many fields, which in turn can hinder the class comprehension and worsen its cohesion, because it may happen that each field will be used by a few methods, the ones that were originally using the corresponding variable.

Despite the frequent occurrence of the need to apply more refactoring operations in sequence, commonly the approaches seen so far focus on suggesting only one refactoring at a time.

1.5 The proposed approach

This thesis presents several solutions to assist evolving software based on data usage, showing how information about data dependence and data propagation can give useful insights about different aspects concerning software modularity, portability and robustness.

Data flow analysis is a statical program analysis used to compute information about the flow of data (in terms of read and written values, like variables and fields being used) for each program point inside the application [63]. It has an iterative formulation that let us efficiently observe the way data are managed by a sequence of program points, statically obtaining an approximation of the runtime behaviour of the code over any possible program execution (e.g. regardless of the inputs configuration). By changing the analysis formulation, i.e. information representation, summarisation rules, and propagation algorithm, we can focus our analysis on different properties, thus solving different data related problems.

Data flow analysis can also be used to build a model of the information transmission inside the program. This is done by associating the point in a program where a value is produced (*defined*) with the point at which the value may be accessed (*used*). Definition-use relations, or pairs, fundamentally capture the flow of information through a program, from input to output. Sequences of definition-use pairs record a kind of program dependence called *data dependence*, which we can use to build a *data dependence graph*, where program points are nodes and definition-use relationships define edges.

Finally, *taint analysis* is a particular class of data flow analysis used to solve problems that can be modelled by considering *sources* for *tainted* data that need to be traced until they reach one of the possible *sinks*. This is typically performed using *graph traversal* algorithms on the data dependence graph, looking for paths connecting a source node with a sink node.

Both in the literature and industries, data flow analysis is commonly used for:

- Transforming a program for optimisation purposes. This is the classical application of data flow analysis, since it was originally conceived in this context.
- Determining the semantic validity of a program (e.g. prohibiting the use of uninitialised variables or checking type correctness).
- Understanding program behaviour to support debugging, maintenance, verification, or testing.

In this thesis, we will see how data flow analysis can be additionally used to endorse software evolution and modularity improvement. In fact, by analysing the

flow of data among different parts of a software unit it is possible not only to better statically understand its runtime behaviour, but also its role and its dependence on the rest of the environment. Different problems concerning software evolution have been addressed, such as:

- Assess the dependence of an application on a particular platform [43], thus suggesting refactoring operations to reduce it;
- Automatically assign to each program point the concern it covers (*concern tagging*), according to the APIs it is using and the data flow it is part of [42];
- Guarantee that sensitive data managed by Android applications will be preserved by leakage and disclosure [7, 72].

Identifying complex refactoring opportunities had an important role in this thesis. In fact, in all of the three cases stated above, different parts of code can be suggested for extraction in a separate unit, such as in a new method. Code extraction can be useful to: (i) limit the dependence on a given platform; (ii) create concern cohesive methods; (iii) find and enclose code dealing with sensitive data in a more policy restricted module. However, in some cases, such as with long and complex methods, more than one extraction could be needed, making them hard to be applied without proper assistance.

To address this problem, an approach to effectively assist a *Replace Method with Method Object* refactoring technique has been also proposed. This is a novel approach since it uses the analysis of data dependence to *optimise* the number of variables to become fields in the *Method Object*, both guaranteeing that all original data flows will be preserved and that a high level of cohesion for the new class will be obtained.

To evaluate the proposed solutions on real world scenarios, tools, such as JADAL and DeDuCT, have been implemented, demonstrating how data dependence and taint analysis can effectively assist software evolution.

1.6 Thesis structure

In Chapter 2 all the required fundamentals for the works described in further chapters will be discussed. The chapter is composed of different sections, starting

from the theory of *data flow*, *data dependence*, and *taint analysis*, giving the fundamentals which were of great importance for the works presented in this thesis. Moreover, concepts related to *code portability*, *refactoring automation*, *concern tagging* and *fragment extraction* will be also given, along with the related work presenting the state of the art given by the literature.

In Chapter 3 a taint analysis approach to assess code portability will be proposed, which identifies code portions that depend on a platform due to the use of data coming from, or further used by, platform specific APIs, thus binding the code and hindering the code portability.

In Chapter 4 the taint analysis approach used to assess the dependence of some code portion on platform specific APIs will be extended for automatically attributing concern tags to each instructions according to the Java APIs they depend on. These tags will then be used to suggest *Extract Method* refactoring opportunities by finding *concern cohesive* code fragments that can be extracted in separate methods, thus improving code modularity and comprehensibility.

In Chapter 5 data dependence analysis has been used to give an automatic approach for replacing a *long method* with a *Method Object*, with the aim of optimising the number of variables to become fields while minimising the *lack of cohesion* of the resulting *Method Object*. This will be used along with the concern tagger proposed in the previous chapter, finally providing an automatic tool to find, and to effectively apply, complex refactoring opportunities by automatically suggest a way to decompose a long method into smaller, concern cohesive parts.

In Chapter 6 a novel *hybrid* approach is described, which combines static and dynamic taint analysis for detecting data leaks in Java and Android applications. It addresses a relevant problem not only from a user point of view, i.e. to preserve personal data from being leaked by an installed application, but also from a developer point of view, i.e. to assess the robustness of a developed application from data leak attempts which may be caused by exploiting some execution flows.

Finally, Chapter 7 gives the conclusions for this thesis, while Appendix A gives three code listings that are referred by Chapter 5 to show a real world use case of decomposing a long method into a *Method Object* by using the suggestions given by the concern tagger described in Chapter 4.

1.7 Published Papers

Part of the work presented in this thesis is based on these co-authored papers:

- M. Mongiovì, G. Giannone, A. Fornaia, G. Pappalardo, E. Tramontana: *Combining static and dynamic data flow analysis: a hybrid approach for detecting data leaks in Java applications*. **Proceedings** of the 30th Annual ACM Symposium on Applied Computing (SAC), 2015.

- G. Ascia, V. Catania, R. Di Natale, A. Fornaia, M. Mongiovì, S. Monteleone, G. Pappalardo, E. Tramontana: *Making Android Apps Data-Leak-Safe by Data Flow Analysis and Code Injection*. **Proceedings** of the 25th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2016.

- A. Fornaia, E. Tramontana: *Is My Code Easy to Port? Using Taint Analysis to Evaluate and Assist Code Portability*. **Proceedings** of the 26th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2017.

- A. Fornaia, E. Tramontana: *DeDuCT: a Data Dependence based Concern Tagger for modularity analysis*. **Proceedings** of the 41th IEEE International Conference on Computer Software and Applications (COMPSAC), 2017.

During the three PhD years also other research areas have been addressed, such as *combinatorial software testing*, *workflow management systems* for cloud infrastructures, *agent-oriented programming*, and *named entity recognition* in Italian unstructured text, co-authoring the following papers:

- G. Borowik, M. Woniak, A. Fornaia, R. Giunta, C. Napoli, G. Pappalardo, E. Tramontana: *A Software Architecture Assisting Workflow Executions on Cloud Resources*. **International Journal** of Electronics and Telecommunications (IJET), 2015.

- A. Calvagna, A. Fornaia, E. Tramontana: *Random versus combinatorial effectiveness in software conformance testing: a case study*. **Proceedings** of the 30th Annual ACM Symposium on Applied Computing (SAC), 2015.

- A. Fornaia, C. Napoli, G. Pappalardo, E. Tramontana: *An AOP-RBPNN approach to infer user interests and mine contents on social media.* **International Journal** of the Italian Association for Artificial Intelligence (AI*IA), IOS Press, 2016.

- A. Fornaia, C. Napoli, G. Pappalardo, E. Tramontana: *Enhancing City Transportation Services Using Cloud Support.* **Proceedings** of the 22nd International Conference on Information and Software Technologies (ICIST), 2016.

- D. Cantone, A. Fornaia, M. Nicolosi-Asmundo, D. Santamaria, E. Tramontana: *An OWL framework for rule-based recognition of places in Italian non-structured text.* **Proceedings** of the 2nd International Workshop on Knowledge Discovery on the Web (KDWEB), 2016.

Chapter 2

Background and Related Work

In this chapter all the required fundamentals for the works described in further chapters will be discussed. The chapter is composed of different sections, starting from the theory of *data flow*, *data dependence*, and *taint analysis* in Section 2.1, giving the fundamentals which was of grate importance for the works presented in this thesis. In particular, *taint analysis* is a particular class of bot static and dynamic data flow algorithm that are typically used to solve problems related to data security. In this thesis, a contribute in this direction will be presented in Chapter 6, showing how it can be used to prevent data leak in Android application. However, in Chapter 4 and Chapter 3 taint analysis will be used in a different field, related to the assistance of software evolution.

In Section 2.2 the problem of *platform dependence assessment* for the *portability measurement* will be presented, giving related work both for portability metrics and tools for platform dependence assessment. Both a novel metric and tool for these kind of problems will be discussed in Chapter 3, where concept and tools for *taint analysis*, typically used to address data security, have been used to solve a different problem, i.e. related to software evolution. This is the same for the the approach presented in Chapter 4, that can be see as a generalisation of the approach used to address platform dependence and code portability.

In Section 2.3 the problem of *concern tagging* will be tackled, showing the related work in the literature and a wide range of approaches that can be used to automatically assign to each instruction the related concern. These will be compared with the approach presented in Chapter 4, where an extension and

generalisation of the work proposed in Chapter 3 has been given specifically to address concern tagging and to suggest *Extract Method* refactoring opportunities.

Since the final objective of this thesis is to provide an approach to automatically suggest *complex refactoring opportunities*, in Section 2.4 several approaches in this field will be presented.

In Section 2.5 the problem of *program slicing* and *fragment extraction* will be deeply discussed, thus giving some notable examples in the literature. This fundamentals will be of great relevance to understand the work presented in Chapter 5, where the data dependence model introduced in Section 2.1 and the concern tagging approach described in Chapter 4 will be used to automatically suggest complex refactoring opportunities by means of the *Replace Method with Method Object* refactoring technique, showing how a long and complex method can be split in smaller methods for a separate class, while guaranteeing a high level of *concern cohesion* of the extracted methods and a high cohesion for the obtained class.

Finally, in Section 2.6 the problem of *data leak* in Android applications will be discussed, presenting how static, dynamic and hybrid taint analysis are used in this context, together with the related work in this wide researching area, giving the support for the work presented in Chapter 6.

2.1 Data flow and data dependence analysis

Static data flow analysis, which can be seen as a way to simulate the program execution, has been originally used by compilers to determine dependencies among variables and to optimize the compiled code. For instance, the set of instructions that are *reachable* from a definition, or the set of instructions in which a variable is still *alive* (i.e. its value can be read by a following instruction).

Another important use is to trace the data dependence among different program points, thus giving a code representation called *data dependence graph* that has been widely used in the approaches described in further chapters, i.e. to solve problems related to data security (see Chapter 6), portability assessment (see Chapter 3), concern tagging (see Chapter 4), and code refactoring (see Chapter 5).

In this section some basic concepts of data flow analysis, and in particular in

the case of Java applications and Java bytecode, will be described, providing the fundamentals that will be helpful in the description of the approaches described in this thesis.

2.1.1 Code optimisation

Data flow analysis was initially used by compilers to efficiently perform code optimisation tasks, and to date this remains its most dominant application. A classical optimisation task performed using data flow analysis is the *available expressions* analysis [80]. It is used to determine if the value of an expression can be saved for future reuses rather than recompute it. This is possible only if we can assure that the value of such an expression remains unchanged regardless of the execution path from the first computation to the second one.

Liveness of variables is another classical data flow analysis that is used to determine whether the value held in a variable may be subsequently used, in any possible execution path [80] (i.e. it is still alive). This can be useful both for program optimisation and bug fixing: (i) to identify *useless definitions*, thus preventing the assignment of variables that will never be used; (ii) to help the compilers in optimising the use of registers by the application, timely assigning them to other variables when stored values are no more needed; (iii) to optimise the use of the heap memory (in this case, we refer to the slightly different problem of *liveness of a pointer* [63]) identifying when a particular memory location is no longer used[1], hence suggesting where a memory deallocation instruction can be introduced.

2.1.2 Data dependence analysis

Other than solving optimisation problems, data flow analysis can be used to build a model of the information transmission inside the program, i.e. by identifying *definition-use* relations among the program statements.

A *definition* occurs in all the statements that give a value to a variable, such as when a variable is either declared, initialised, assigned with a value, or received as

[1] *Liveness of a pointer* analysis finds also memory locations that are no longer used but still addressed by other pointers, thus preventing the garbage collector from reclaiming them.

a parameter. A *use* occurs in all the statements whose execution extracts a value from a variable, such as expressions, conditional statements, parameter passing, or return statements.

To form a definition-use pair we need to find a *definition clear* path from definition to use, that is, if there is a program path on which the value assigned in the definition can reach the point of use without being overwritten (or *killed*) by another assignments. This particular data flow analysis is called *reaching definitions* analysis [80].

Sequences of definition-use pairs record a kind of program dependence called *data dependence*, which we can use to build a *data dependence graph*, where program points are nodes and definition-use relationships define edges.

2.1.3 Build the Data Dependence Graph

To perform the reaching definition analysis and build the *Data Dependence Graph (DDG)* we need both the *Control Flow Graph (CFG)* and the information about the variables *used* and *defined* by each program instruction. In the case of the *Java Virtual Machine (JVM)* (the one we are interested in), data can be stored in a *local* or *member* variable, or in a slot of the *operand stack* (the operand stack temporarily stores data before processing them), and hence we refer to any of them as *memory cells*.

Given a program P as a set of instructions, each instruction s in P reads (*uses*) an input from a set $U(s)$ of *memory cells* and writes (*defines*) the result on a set $D(s)$ of memory cells. A runtime execution σ of P is the ordered sequence of instructions in P (with possible repetitions) that is executed by running P on certain inputs.

The order of execution of instructions within a method of a program can be described by a *Control Flow Graph* (CFG). A CFG is a graph where nodes are instructions and edges are defined as follows: there is an edge (u, v) between two nodes if v can be the successor of u in some execution[2]. A CFG can be built from a program by analysing the order of instructions in the code and the possible outcome of jump instructions. CFGs of method bodies in OO programs can be combined to

[2]Nodes may even be blocks of instructions that are always executed together. For simplicity, our description considers blocks as just single instructions.

```
private static int process(int value) {
    value ++;
    // 0: iinc %0 1

    if (value > 100000)
    // 3: iload_0
    // 4: ldc 100000
    // 6: if_icmple #16

        membSensitive = value;
        // 9: iload_0
        // 10: putstatic MyClass.membSensitive I (6)
        // 13: goto #18

    else
        value = 0;
        // 16: iconst_0
        // 17: istore_0

    return value;
    // 18: iload_0
    // 19: ireturn
}
```

Figure 2.1: An example of a method source code commented with bytecode instructions.

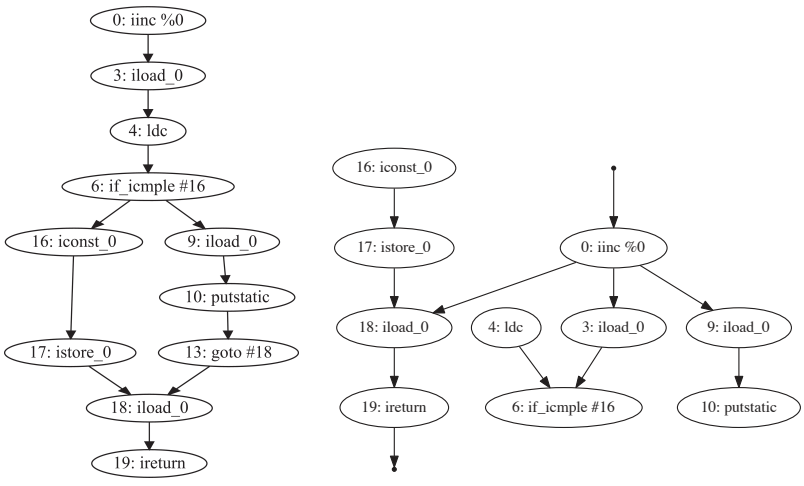

Figure 2.2: Control Flow Graph (a) and Data Dependence Graph (b) for the method shown in Figure 2.1.

build a CFG that describes the whole program. Figure 2.2(a) shows an example of a CFG corresponding to the method whose code is listed in Figure 2.1. Comments after each line indicate corresponding bytecode instructions. Each instruction is linked with the subsequent instruction, however conditional jump instructions (e.g. `6:if_icmple`) are typically connected with more than one instruction.

The data flow in a program can be described by data dependences.

Definition 1. There is a *data dependence* (u, v) between two instructions u and v if v can *use* (read) a value x previously *defined* (written) by u, hence a potential data flow occurs between two instructions. This is possible if: (i) there exists a path p in the CFG from u to v, and (ii) p is *definition clear* path for x, that is, no other instruction $w \in p$ defines (i.e. overwrites) its value.

A data dependence (u, v) refers to a set of memory cells $\lambda(u, v)$ that produce the data transfer. Data dependences can be grouped together in a *Data Dependence Graph* (DDG) that describes the whole (*potential*) data flow in a program.

A real data flow between instructions occurring during the execution of a program corresponds to a path traversal in the DDG. We refer to a *real traversal* of an edge (at runtime) as an *edge flow*. This distinction will be of great importance in Chapter 6, where static data flow analysis will be used to detect critical program points to be monitored at runtime, then instrumenting the Android application to let it able to detect edge flows occurred at runtime.

Figure 2.2(b) reports an example of DDG related to the code in Figure 2.1. Edges from and to dots correspond to input parameters and return values, respectively. Among others, there is a data dependence between `0:iinc %0` and `18:iload_0`. Indeed they both refer to the same local variable 0 (for writing and reading, respectively) and there is a path between instructions 0 and 18 in the CFG whose internal nodes do not modify variable 0.

2.1.4 Reaching definitions data flow analysis

Using Definition 1 it would be possible to compute definition-use pair by searching the CFG for *definition clear* paths between all possible instruction pairs. However, generally the number of paths in a graph is exponentially larger than the

number of nodes and edges [80]. Therefore, checking all possible individual paths can be unbearable.

An efficient algorithm for computing reaching definitions is based on the way reaching definitions at one node are related to the ones of predecessors in the CFG. Suppose we want to determine the reaching definitions for node n: we indicate with $ReachIn(n)$ the set of definitions (x, v) reaching node n, where x is the defined variable and v is the instruction which has defined it; we also indicate with $ReachOut(n)$ the set of reaching definitions (x, v) leaving node n. We can give a definition of these two sets by means of specific *data flow equations*:

$$ReachIn(n) = \bigcup_{p \in pred(n)} ReachOut(p)$$
$$ReachOut(n) = ReachIn(n) \setminus kill(n) \cup gen(n)$$

where $gen(n) = \{(x, n)\}$ if variable x is defined by n, $\emptyset$ otherwise; and $kill(n)$ is the set of all other input definitions of variable x *overwritten* at n (if x is defined by n, $\emptyset$ otherwise). In the particular case of the JVM, a typically for other stack based languages, also the uses of stack operands kill reaching definitions for this kind of memory cells, since once used, values are *popped* and removed from the stack, thus can be considered no more available.

By means of these data flow equations, reaching definitions can be efficiently calculated using an *iterative approach*: first, the reaching definitions at each node in the CFG need to be initialised to the empty set; then, the equations are iteratively applied until the results stabilize. Once reaching definition sets for each instruction are calculated, data dependence can be easily computed. That is, there is a data dependence (u, n) with every instruction u providing a reaching definition (x, u) for variable x used by n:

$$(u, n) \in DDG \leftrightarrow (x, u) \in ReachIn(n) : x \in U(n)$$

Figure 2.3 shows the reaching definitions for the method listed in Figure 2.5(a), obtained by using the iterative algorithm (only *ReachOut* sets are depicted for simplicity, since *ReachIn* can be easily derived by the union of predecessor ones).

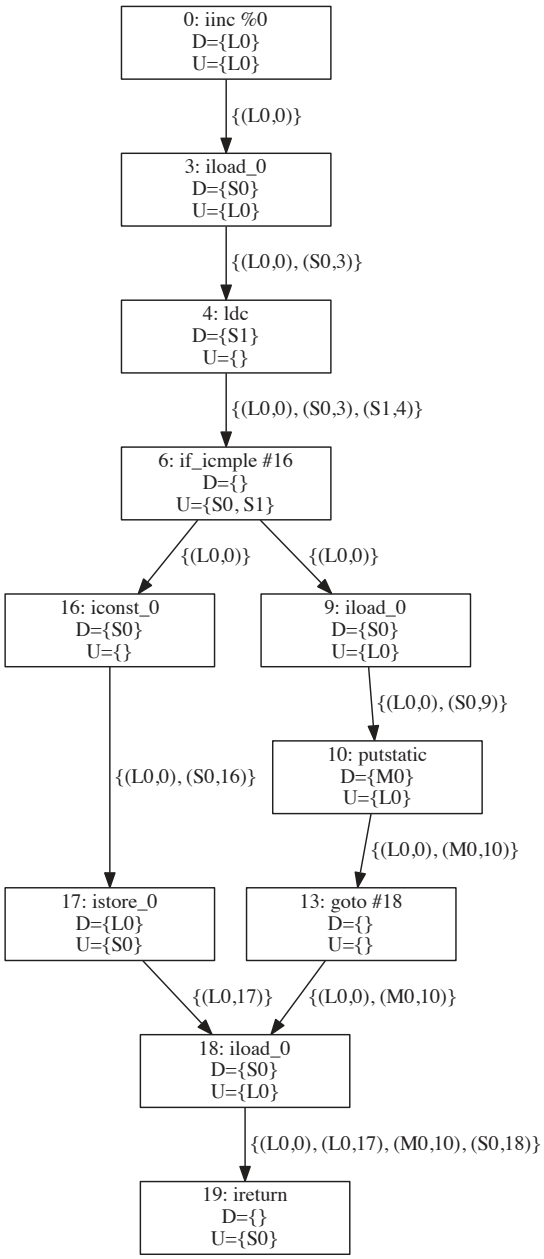

Figure 2.3: The Control Flow Graph, for the method shown in Figure 2.5(a), noted with information about memory cell *definitions* $D(s)$ and *usages* $U(s)$ for each instruction s (node) in the graph. Both are composed by the memory cells involved in the method, i.e. the local register *local_0* ($L0$) for the `value` variable, the memory location *M0* storing the member variable `membSensitive`, and the two slots for the operand stack $S0$ and $S1$ (starting from the base).

For example, let us consider instruction `18:iload_0`: $D(18) = S0$, where $S0$ is the first slot of the operand stack (the base), and $U(18) = L0$, where $L0$ is the local register in 0 position addressed by the instruction. $ReachIn(18) = ReachOut(17) \cup ReachOut(13) = \{(L0, 17), (L0, 0), (M0, 10)\}$, where $M0$ is the memory location used to store member variable `membSensitive`. There is a data dependence with both instruction 17 and 0 due to the use of $L0$; $gen(18) = (S0, 18)$ and $kill(18) = \emptyset$, since no definitions for $S0$ reached the node. Therefore $ReachOut(18) = ReachIn(18) \setminus \emptyset \cup \{(S0, 18)\} = \{(L0, 0), (L0, 17), (M0, 10), (S0, 18)\}$.

2.1.5 Dealing with Java bytecode

Dependence analysis for Java bytecode introduces some challenges [121], mainly due to its stack-based architecture, in which stack cells store intermediate calculations and may lead to implicit data flow between instructions [89]. The implicit stack masks the flow of data and thus makes the bytecode quite difficult to analyse: at a given bytecode instruction s, it is not at all obvious which previous s_0 produced the stack-based inputs of s [65].

This problem has been solved by data flow analysis frameworks and data dependence tools by storing all data, i.e. both for local, member variables and implicit stack slots, in named *abstract variables*, thus making the data flow much more obvious. For example, the key features of the Soot framework[3] [65] is that it creates a simplified *three-address* intermediate representation of the Java bytecode (Jimple [106]), where each instruction is represented using statements of at most 3 local variables or constants (such as $t1 := t2 + t3$), thus also stack slots are represented in these statements using named temporary variables. Also Wala[4] leverages an intermediate representations to solve this problem.

The JavaPDG [89] tool for program (and data) dependence uses abstract variables to handle the stack, but differently from Soot and Wala, it directly analyses the Java bytecode, without the need of an intermediate representation. This important difference motivated its adoption in the approach described in Chapter 6. In fact, a high precision about the bytecode instruction causing a data dependence was needed, in order to inject a monitoring component right before the

[3] https://sable.github.io/soot/
[4] http://wala.sourceforge.net/wiki/index.php/Main_Page

execution of that specific instructions, thus avoiding the actual data leak.

Soot was used instead in Chapter 5, because it is easier to customise to perform different kind of data flow analysis (in fact, is more a framework for data flow analysis than a tool for data dependence), and because the bytecode precision, e.g. provided by Java PDG, was not a requirement in this case (data dependences was required at a Java source code level).

Both Soot and JavaPDG do not directly analyse Java source code. This is not a limit, but a necessity for the analysis to be performed correctly and efficiently. In fact, Java language is too complex to be analysed directly, thus requiring a lower level representation able to simplify and linearise the code in a sequence of lower level steps, allowing a much better performance than direct interpretation of source code. This could be the bytecode itself (like for JavaPDG) or another intermediate representation (like for Soot).

For example, the linearisation process made by Soot using Jimple will split up the Java statement "`int x = (f.bar(21) + a)*b`" into a sequence of three-address statements: "`$i3 = virtualinvoke r2.<Foo:int bar()>(21);`", "`$i4 = $i3 + i0`", and "`i2 = $i4 * i1`" (where '$' starting variables are used to replace stack references, that would be made by the bytecode of that Java statement)[5].

However, having a data dependence graph for the bytecode or jimple representation of a method under analysis, it can be easily converted in a DDG for the corresponding Java source code by replacing groups of nodes, related to lower level statements, with a node related to the corresponding Java source line, as shown in Figure 2.4. Hence, using either Soot or JavaPDG we were able to obtain a DDG at any required level of detail, either at a bytecode or a source code level.

2.1.6 Scopes of data dependence analysis

According to the scope of performed data flow analysis, we distinguish three kinds of data dependence: (i) intra-method data dependences; (ii) inter-method data dependences; and (iii) member variable data dependences.

There is an intra-method data dependence between two instructions u and v if there is at least one path between u and v in the CFG such as a memory cell

[5]http://www.brics.dk/SootGuide/sootsurvivorsguide.pdf

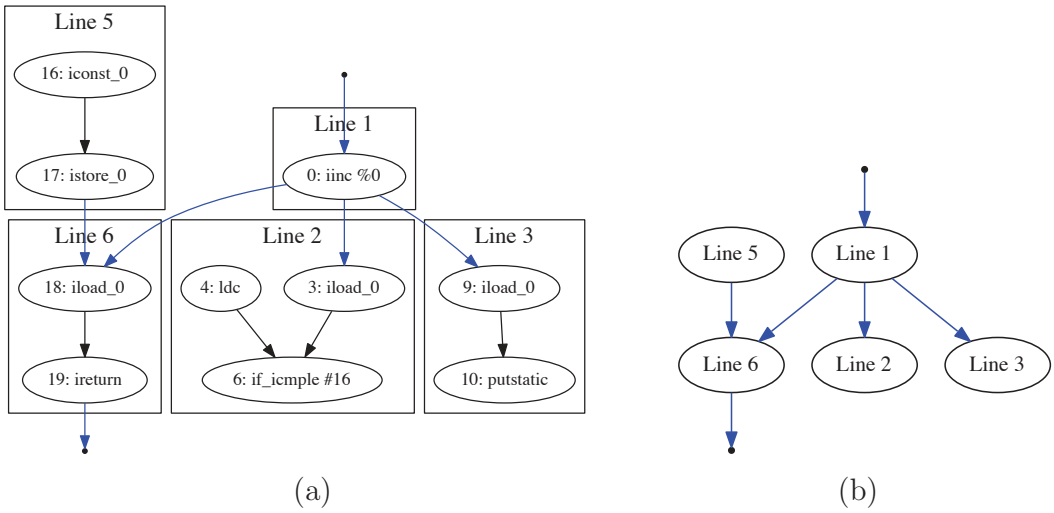

Figure 2.4: (a) Data Dependence Graph obtained by performing the data flow analysis at bytecode level. (b) Data Dependence Graph of the Java source code obtained from (a) by grouping the bytecode instructions according to the Java source line they come from.

(variable or operand stack) written by u is read by v without being replaced by another instruction of the path.

An inter-method data dependence represents the flow of parameters between caller and callee methods. Specifically, if method m_1 calls method m_2, for each parameter a of m_2 there is a data dependence between the instruction of m_1 that prepares a for the call and each instruction of m_2 that uses a. Not always it is possible to statically determine the invoked method. Indeed, virtual and dynamic method calls let the runtime support choose the method to be invoked from a list (dispatch table) that contains all the override implementations of the callee method. In this case, the outcomes of the static analysis let us consider that data dependences exist between the caller and each possible callee. This will ensure in Chapter 6 that any potential data flows will be considered.

Member variable data dependences model any (inter- and intra-method) data dependences that use member variables. Specifically, for every member variable λ there is a data dependence between any instruction that writes to λ and any instruction that reads from λ.

Figure 2.5 describes an example of a complete DDG. Solid lines represent intra-

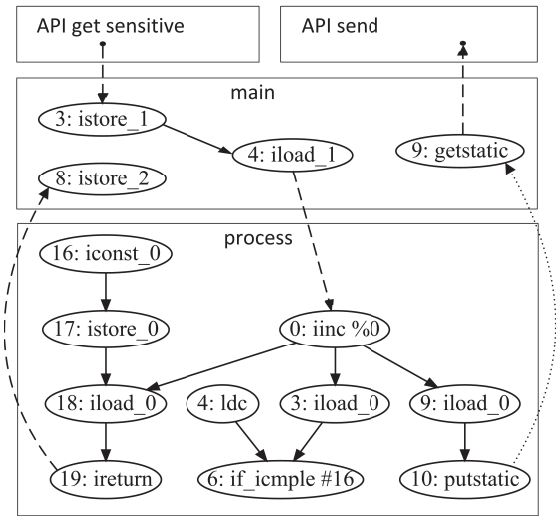

Figure 2.5: An example of DDG. Solid lines represent intra-method data dependences; dashed lines represent inter-method data dependences, whereas dotted lines represent data dependences due to the access to member variables or global variables.

method data dependences; dashed lines represent inter-method data dependences, whereas dotted lines represent member variable data dependences.

2.1.7 Taint Analysis

The *taint analysis* is a particular class of both static [5] and dynamic [87] data flow analysis that solves problems that can be modelled by considering *sources* for *tainted* data that need to be traced until they reach one of the possible *sinks*. While flowing from one instruction to another, instructions become *tainted* too, and a trace appears that can be used for the particular analysis.

It is mainly used in the context of *data security*, e.g. for the *data leak* detection, where we want to see if data coming from sensitive sources can flow to specific unauthorised channels (i.e. sinks) (see Section 2.6 for further details and related work).

In this thesis, however, taint analysis was also adapted to solve a different kind problems, i.e. related to the *software evolution*, such as to automatically assess the concern of each line in the source code to suggest refactoring opportunities (see Chapter 4), and to automatically assess platform dependence and code portability

(see Chapter 3).

Therefore, *taint analysis* was an important concept for this thesis, together with *data dependence*. Even if both belongs to the *data flow analysis* category, they are used for different reasons.

- Data dependence answers questions like: *which data does an instruction depend on?* Thus, *on which instructions does an instruction depend due to the data it uses?*

- Taint analysis answer questions like: *how do data produced by an instruction flow inside the application due to the data propagation? Do these data reach some points of interests into the application?*

The former is used to to solve problems like determining the input and the output variables of a code fragment, e.g. when an *Extract Method* refactoring technique is used (as will be shown in Chapter 5). However, as shown in Section 2.1.3, data dependence analysis is needed to construct the DDG, which taint analysis relies on, thus we can say that taint analysis is an extension of the data dependence one.

The latter solves problems like determining if a sensitive data can flow to an unauthorised channel (as will be shown in Chapter 6), to determine which instructions depends on a particular platform due the use of data coming from, or flowing in, platform specific APIs (as will be shown in Chapter 3), or to see how a concern tag applied to a particular instruction can be spread to other ones due to a cascade of data dependences (as will be shown in Chapter 4).

2.1.7.1 Static (STA) and Dynamic (DTA) Taint Analysis

Taint analysis approaches mainly differ in the way these two aspects are addressed:

- on how *sources* and *sinks* are defined: depending on the the analysis needs, the way sources for tainted data and sinks for stopping the tainted flow may change.

- on the timing of the analysis, whether *statically* or *dynamically*.

For example, to solve the problem of platform dependence assessment, instructions directly using platform dependent APIs will be used as sources, while instructions performing almost common operations will be used as sinks (see Chapter 3 for details). Whereas, to solve the data leak problem, instructions accessing sensitive data will be used as sources, while instructions sending data on untrusted channels will be used as sinks (see Chapter 6 for details).

Static Taint Analysis (STA) [5, 34, 48] consists in using a static data flow analysis to find all possible paths connecting a source with a sink, i.e. making the data produced by the considered sources to flow inside one of the possible sinks. This is typically performed using *graph traversal* algorithms on the Data Dependence Graph.

Dynamic Taint Analysis (DTA) [37] leverage monitoring components, which are typically introduced by customising the execution environment, to dynamically *trace* the data flow coming from a source program point until it reaches one of the possible sinks.

Both of the taint analysis approaches have some limitations. STA is liable to false positives, since it cannot be determined whether a data flow path causing a possible connection between a source and a sink will ever be executed (i.e. due to polymorphism and conditional statements). In some cases, this could be a limit, for example, in the context of data security, we cannot be sure whether sensitive data will actually be leaked at runtime.

DTA may introduce a considerable monitoring overhead because it usually requires every instruction to be monitored at runtime. In Chapter 6 an *Hybrid Taint Analysis (HTA)* approach will be presented, i.e. combining both static and dynamic taint analysis to reduce the monitoring overhead at runtime by statically selecting a minimal set of program point to be monitored, i.e. the ones involved in a probable data leak, that is a path in the DDG from a source to a sink.

2.2 Code portability

Despite the recognised importance for the portability *non-functional* requirement, few contributions in the literature provide a way to assess assess it. Although, different definitions of the portability requirement have been given, stated as “the

percentage of target dependent statements" [92] or as "how much modifying and adapt the code to a new environment is cheaper than redevelop it" [73].

Although those definitions are general enough, they both suggest that, from a developer point of view, the first step in assessing code portability is to identify the program points binding the code to a specific underlying environment or solution. Moreover, costs can be defined in terms of platform dependent instructions [74, 96], since the more code is platform bound, the more costs need to be afforded to change it and make the program able to interact with a different environment.

However, by only counting the program points directly interacting with a given platform, such as using platform APIs, only an under estimation of the platform dependent code can be given. In this section other related approach will be compared to the one proposed in Chapter 3, which provides both a new portability metric and a taint analysis approach to automatically assess it. The proposed approach will consider *data dependences* to give a more precise and realistic evaluation of the number of *platform dependent instructions*, thus extending the portability assessment also to code not directly interacting with the platform.

2.2.1 Related work

In [2], there is a survey of portability-related concepts (dispersed throughout the relevant standards), which describe, at varying levels of detail, the attributes of the portability requirement at the system, software, and hardware levels. In [71], the authors presented a survey of software architecture evaluation methods, considering several quality attributes like: performance, maintainability, testability, and portability. Their review considered many articles, selecting a set of ten evaluation methods. However, they conclude that none of these methods evaluated portability explicitly. My approach proposes instead an evaluation technique that is focused on the portability attribute.

In [96] and [74], the authors have shown how portability costs can be defined in terms of platform dependent instructions, since the more code is platform bound, the more costs have to be afforded to change it and make the program capable to interact with a different environment. The approach described in Chapter 3 gives an advanced measurement of the number of platform dependent statements, and can then be used in conjunction with the said cost metrics.

In [110], the authors give a metrics suite for measuring the reusability of a software component, based on a reusability model that includes aspects related to the understandability, adaptability, and portability factors. They statistically evaluated a metric assessing the portability of a software component. The statistical findings suggest that the smaller the number of business methods in a component without return value, the smaller the possibility of the component having external dependence, which leads to high portability. The difference with my approach mainly lies in the granularity and precision: while they estimate the portability at a component level, by observing some characteristics external to the methods, we carefully identify and properly count the platform dependent statements or lines of code within methods.

In [67], the authors discussed the portability evaluation for executable service-oriented processes. They stated that even if process languages are based on widely supported open standards, such as the Web Services Business Process Execution Language, these are only partially implemented by every platform, thus hampering process portability and locking in their users. Hence, they provide metrics to help the developers in assessing the costs of porting a process to another environment. Their portability assessment of service-oriented processes is based on cost metrics. My approach is instead more general, and can be adopted even in such a context by simply stating the APIs to be considered as a source of platform dependence.

2.3 Concern tagging

Identifying code addressing different *concerns* inside a module is of paramount importance to improve code readability e maintainability, finding opportunity for separating code performing different tasks in different *concern cohesive* modules. This problem is known as *concern identification* [85] or *concern tagging*, that means mapping source code elements on to human oriented *concepts* [14] or programming features [78].

Tagging code artefacts with descriptive keywords can be useful at different granularity levels. Coarse grained tags are used in *Service Oriented Architectures (SOA)* to classify re-usable software components, thus providing a way for *brokers* to find services with similar functionalities. An effective way to obtain this high level

description for web services is by leveraging *structured collaborative tagging* [47], i.e. by asking the community of users for annotating indexed web services with tags describing service behaviour and interface (input and output).

Collaborative manual tagging can be useful also during software development. Fine grained tags given inside code can effectively capture the relevant knowledge for a software development team. For example, in [94] it was proposed an IDE plugin for tagging code using properly formatted inline comments (with the Java annotation style), thus enabling the collaborative annotation, navigation, and coordination of developer activities. This can be useful to inform other developers that a code portion has been modified because of a bug fix, or to provide a step-by-step guide for newcomers to navigate a software library or framework.

However, for concern tagging to be effective, usually concern information need to be added to each line inside the source code [42, 97], thus making manual tagging as expensive as coding. Moreover, every time a line of code would be changed, concern tags should be updated. Therefore, even if collaborative, fine grained manual concern tagging can easily become unbearable even for small projects, thus calling for automatic tools to support concern tagging.

In this section, several concern tagging approaches will be shown, thus comparing them with the one proposed in Chapter 4, *DeDuCT*, where API accesses have been used as *concern seeds* for a taint analysis approach, which will automatically assess the responsibility of each instruction (its concern) according to the API data it depends on.

2.3.1 Related work

Several approaches aim at concern identification. Terms like *concept mapping* [68] or *feature location* [35] refer to similar problems. Additionally, *aspect mining* approaches aim at identifying *crosscutting* concerns that can be expressed as aspects. Concern identification and aspect mining have been based on: information retrieval [82], program execution traces [36] historical information [3], structural program dependencies [84], and data flow analysis [98].

In [69], the *fan-in* metric has been used, in order to locate methods with a high number of incoming calls (in contrast with the *fan-out* metric, which counts the number of outgoing calls [56]). According to the authors, a high fan-in

value may indicate the presence of a *crosscutting concern*, that is a feature whose implementation is spread over multiple system modules, thus difficult to isolate.

A typical example of crosscutting concern is the *logging* functionality: whenever there is a need for a log, a Logger class is involved. This can happen in multiple classes inside the program, causing the code for logging *concern* (functionality) to be inevitably spread in multiple modules, reducing the chance of reusing the code. The presence of such behaviours does not depend on an inadequacy of class design, but from an intrinsic limit of object oriented models [64].

To overcome this limitation, a new programming paradigm, called *Aspect Oriented Programming* [64], has been proposed, along with the object paradigm to encapsulate in separate entities, called *aspects*, the code related to a crosscutting concern, thus improving modularity. Different approaches have been proposed to identify crosscutting concerns within an object system, then suggesting refactoring opportunities by means of aspects [17, 97, 107].

In [82], the authors combine information retrieval, based on Latent Semantic Indexing (LSI), and Formal Concept Analysis (FCA) to locate concepts in a source code. Firstly, they create a corpus from comments and identifiers found in source code. Then, by means of queries using terms describing the concept of interest, (e.g., 'print page'), a list of ranked source code elements (classes, methods, files) is shown, according to their similarity to the query. FCA is finally used to organise the obtained results in a concept lattice in order to ease user inspection. As for other textual based techniques applied to source code, the main drawback is that it relies on the naming convention, whereas the approach described in Chapter 4 relates to APIs as a more robust and reliable indicator, as suggested in [97].

In [36], given a test suite for the software under analysis, the authors' approach asks the developer to suggest which test cases trigger the execution of some parts of a specific feature. Then, test execution traces are analysed to identify method invocation patterns, and heuristics used to rank the contribution of each invoked method to the considered feature. Like other dynamic approaches, this one highly depends on the quality of the available test suites and execution scenarios. Moreover, the portions of code identified as contributing to a feature are methods, hence expecting that all the method code implements one functionality. In my approach, by addressing each line of code instead, we have a much finer granularity, and can

identify which parts of a method are related to each concern.

In [3], the authors propose an history-based concern mining technique. Their approach clusters functions, variables, types and macros that have been changed together intentionally, thus possibly related to the same concern. The precision of such an approach depends on the distribution of changes occurring in several commits.

In [98], the author proposes concern identification based on data flow analysis. A concern has been defined as the functionality needed to produce a given set of related values. Each variable is used as a sink for a backward data flow analysis identifying all dependent variables. As a result, a single functional concern is outlined by means of a group of variables. Hence, the approach would suggest that portions of code contribute to the same concern only when there is a data dependence among variables. In contrast, in my approach we can identify portions of code contributing to the same concern even when having different data flows.

Moreover, his approach uses data flow information to group variables related to the same concern, but without giving clues about the concern being implemented, other than the ones suggested by the variable names. *DeDuCT* provides instead high level description of the responsibility of each instruction, according to the data it depends on. Finally, where the author's approach uses concern seeds as sinks for a backward data flow analysis, *DeDuCT* uses them as sources for both a backward and forward analysis.

2.4 Automate refactoring opportunity detection

In the literature there are several *refactoring* [44] techniques aiming at improving the structure of an object systems, while taking care not to alter its functionality. Each of these should be applied by the developers following these steps:

1. identify which classes seem to be appropriate for a refactoring technique to improve modularity;

2. determine the exacts refactoring steps to be applied onto selected classes;

3. apply refactoring steps;

4. test that such refactoring steps have not changed the program semantic, that is, its functional behaviour;

5. measure the effect of refactoring over the software quality, e.g. in terms of complexity, comprehensibility and maintainability;

6. keep the *consistency* between the refactored code and all the other artefacts involved in software development, such as design documents, test plans, and so on.

However, even though refactoring catalogues detail the steps for the application of each technique, the problem of efficiently understand *where* there is a need to apply a refactoring remains. Catalogues only refer to qualitative parameters, *code smells*, to detect code that need to be changes, such as looking for *duplicated code*, *long methods* or *long parameter lists* [44]. This makes refactoring difficult to be applied in the case of large software, due to the large number of components and complexity it can have.

In this regard, one of the research areas in Software Engineering is devoted to the development of approaches for the automatic suggestion of *refactoring opportunities* in large systems, many of which make extensive use of software quality metrics.

For example, we could measure the cohesion of each individual class using the LCOM metric, and then specify a threshold for the obtained distribution (such as the mean, or percentiles). All classes exceeding such threshold could be identified as affected by a lack of cohesion. As previously seen, this may be a symptom of a class that is too complex, i.e. whose role is not well defined because of its unrelated methods.

This can be solved by refactoring, moving some of its methods to other classes, but the problem of choosing which ones to migrate remains. Furthermore, we have to select the destination classes. Another possible refactoring solution is to split the class into two distinct ones, distributing the methods in such a way as to create well cohesive classes to be extracted (i.e. using the Extract Class refactoring) [44]. However, we still have to decide how these methods should be partitioned and how these changes will affect the rest of the software system.

It should be emphasized, however, that while much work has been done with regard to the definition and measurement of software quality, for which, for example,

several metric sets are available today, there is still much to do with techniques to suggest opportunities for refactoring. In Chapter 5 an approach for a long method decomposition will be discussed, which can take advantage of the DeDuCT approach (described in Chapter 4) to refactor a long and complicated method into smaller and concern cohesive parts.

2.4.1 Related work

In [91] authors have proposed a concept of cohesion based on the *Jaccard* similarity coefficient, then used it as a measure of the distance between the elements (attributes and methods) of a software system. Given two sets of properties, one for each entity, this coefficient is defined as the ratio of the cardinality of their intersection to the cardinality of their union. Considering a method as an entity, the set of properties can be defined as the methods it calls and the fields it uses. Considering instead a field, its property could be associated with the methods it serves. The similarity so defined can indicate how much two methods *should be close to each other*, i.e. how much would be appropriate for such methods to be in the same class.

Using similarity as a distance function, automated clustering techniques can be applied, so to see whether methods and attributes of the same class are in the same cluster, being close to each other, or identifying which class should host them. These kind of approaches can really help the developer in making his refactoring activity effective. However, it is important to note that techniques used for automatic refactoring suggestions may offer structural improvement opportunities in a capillary manner, yet leaving the developer to decide whether or not to apply them.

In [76] the similarity concept just seen is used in conjunction with the LCOM cohesion metric. In this case, when two methods show an high similarity (above a certain threshold) while being in different classes, it is observed how the LCOM measure can change by moving the method from the original class to the one containing the other method. A Move Method refactoring is then suggested only if the LCOM value of both classes will be improved.

In [79] instead, new metrics for class cohesion have been proposed, considering the strength with which their methods interact with each other, in terms of the number and type of parameters passed from one method to another through a call

(i.e. the amount of data). This information is then used to automatically suggest Extract Class refactoring opportunities, thus providing partitions of methods showing a high cohesion. These partitions are then extracted into separate new classes, thus decreasing the complexity of the original one.

In [100] an approach to suggest Move Method opportunities is proposed. It is based on the measurement of coupling between methods belonging to different classes. However, the cohesion or complexity of the original classes has not been considered. By combining the use of multiple metrics, thus handling simultaneously more contrasting quality aspects, it is possible to provide refactoring solutions that can lower, at the same time, both the lack of cohesion and the coupling between different classes [76].

There are different models that can be used to represent a program, such as an *Abstract Syntax Tree* in case you want to highlight the syntax structure of the individual classes. To support refactoring decisions, a *Software Network* can also be used, that is a graph where each node is a code entity, such as a class, an interface, a method, or an attribute, and where a relationship between two entities can be represented as a direct edge between the two nodes [103]. This last representation can also be enriched by annotating the nodes with metric values, indicating, for example, the relative cohesion or coupling level.

As shown in [16], representing a software system as a graph, it is possible to see refactoring operations as *graph transformations*, where the initial state can be described by the subgraph induced by the nodes involved in the refactoring operation, and where the final state is identified by the subgraph been reached. Thanks also to the quality metrics applied to each node, the improvement on quality can also be easily evaluated.

2.5 Suggest *Extract Method* opportunities

Extract Method is considered as one of the most important refactoring techniques, since it is often used to solve typical design problems such as Duplicated Code, Feature Envy and especially Long Method [44]. It consists in breaking large methods into smaller ones easier to reuse, thus improving code maintainability. Moreover, it is usually a first step for a sequence of other refactorings, such as

Move Method and Extract Class, which are applicable only when long methods have been already split into smaller pieces.

In the literature, most of the approaches about method extraction can be grouped in mainly two different categories, based on *program slicing* [113] or on *fragments extraction*.

The former considers the extraction of the complete computation of a given method variable as a reusable program entity [39]. Slices are computed by finding sets of directly or indirectly relevant statements based on control and data dependencies, which are commonly obtained by statically analysing the *program dependence graph (PDG)* [40] of the method. Thus, slices may possibly involve *non-contiguous* statements.

The latter, also referred to *block extraction*, identifies sequences of *contiguous* statements which can be legally extracted in a separate method, as long as the syntactical correctness of both the original method and the extracted one is preserved (e.g. considering an *if-then-else* statement, it is legal to select statements all belonging to the same branch, but is not possible to range the selection to both of the branches without extracting the conditional statements as a whole).

One of the main challenges tackled by program slicing is to guarantee that slice extraction does not change the program behaviour, which can be hard, since this kind of extraction may require some code to be *duplicated* or *reordered*. On the contrary, by extracting a contiguous sequence of statements (a fragment), moving it into a new method, and then replacing it with a related method call, there is no need for code to be reordered or duplicated. This makes guaranteeing the equivalence of program behaviour, before and after the method extraction, much easier.

2.5.1 Compare *slice* and *fragment* extraction

To better explain the problem of slice extraction compared to fragment extraction let us analyse a simple example. The method shown in Figure 2.6 uses the input variable x to control the computation of two variables, a and b. Let us suppose we want to extract all the statements that performs the complete computation of variable a, i.e. the related program slice. This means identifying all the statements that are used to compute its value, which are the ones on lines 2, 6 and 9, due

```
1  public void m(int x) {
2      int a = 0;
3      int b = 0;
4      while (someCondition(x)) {
5          if (someOtherCondition(x)) {
6              a += 1;
7              b += 0;
8          } else {
9              a += 0;
10             b += 1;
11         }
12     }
13     print(a,b);
14 }
```

Figure 2.6: A simple method used as an example to explain the differences between *slice* and *fragment* extraction. It uses the input variable x to control the computation of two variables, a and b.

to their *data dependence*. However, in order to preserve the proper computation, some statements need to be duplicated (not simply extracted), i.e. both the *while* and the if statements, due to their *control dependence*. Differently from the first group of statements, we cannot simply remove the control statements, since they are required in the original method to compute b. The final results will be the one shown in Figure 2.7.

Code duplication is often required to perform the slice extraction. However, in some cases this can lead to a change in the program behaviour, i.e. when we duplicate method invocations that change the object state. Let us suppose that the method $someCondition(x)$ (see Figure 2.7) internally affects the object state by updating a field to count the number of times the method is invoked. By duplicating this method invocation we have also affected the final value of this field, thus not preserving the original behaviour.

Another problem to be addressed is code reordering. In the previous example (see Figure 2.7), we have moved the entire computation of variable a into a new method, whose call has become the first statement of the original method. This obviously has changed the order in which the statements were originally executed. In some cases, this may also change the program behaviour. Let us slightly modify the previous example by adding a $print(a, b)$ invocation also in the first part of the method (see Figure 2.8).

In this case, by placing the $getA(x)$ method invocation *before* the added

```
public void m(int x) {
    int a = getA(x);
    int b = 0;
    while (someCondition(x)) {
        if (someOtherCondition(x)) {
            b += 0;
        } else {
            b += 1;
        }
    }
    print(a,b);
}

public void getA(int x) {
    int a = 0;
    while (someCondition(x)) {
        if (someOtherCondition(x)) {
            a += 1;
        } else {
            a += 0;
        }
    }
    return a;
}

private void someCondition(int x) {
    this.count++;
}
```

Figure 2.7: The code of method *m(x)* (see Figure 2.6) after the extraction of the program slice related the computation of variable *a*, here extracted into method *getA(x)*.

```
public void m(int x) {
    int a = 0;
    int b = 0;
    print(a,b);  // <-- added
    while (someCondition(x)) {
        if (someOtherCondition(x)) {
            a += 1;
            b += 0;
        } else {
            a += 0;
            b += 1;
        }
    }
    print(a,b);
}
```

Figure 2.8: A modified version of method *m(x)* (see Figure 2.6). Another call to the *print(a,b)* method has been added, in order to show how *code reordering* may change the program behaviour after a slice extraction.

print(a, b), the method will not print the expected zero value for variable a, but instead the final one. This is caused by the *anti-dependence* [100] between the slice and the printing statement on line 4: an anti-dependence exists from statement A to statement B (or statement B anti-depends on A), when statement A uses the value of a variable that is later modified by statement B. To preserve code behaviour, the invocation of the extracted method should be placed *after* statements that the slice statements anti-depend on [100]. In fact, if we move the method invocation after line 4, the behaviour is preserved.

While fragment extraction dose not have the code duplication and reordering problems, it has other ones. In fact, let us suppose that instead of extracting the program slice of variable a, we want to make the control flow more linear by extracting the *while* body into a new method (lines 5-11 in the *m(x)* shown in Figure 2.6), thus reducing *control program complexity*.

The first thing to do is to look for any variable or parameter that is used or modified by the fragment, excluding the ones that are local in scope for the fragment, i.e. not visible to the rest of the method. Any non-modified variable can be passed in as parameter (*input variable*). Modified variables need more care. If there is only one, it can be returned. In this case, a problem arises: since both a and b are *output variables* for the fragment, i.e. they are modified inside of it and later used by the original method, this can make the fragment extraction unfeasible, since commonly multiple return values are commonly not allowed by programming languages, such as Java.

A possible solution (as shown in Figure 2.9) is to make at least one of the two variables a *field*, thus avoiding the need for multiple values to be returned by the extracted method. However, it is important to note that by using fragment extraction in this case it is not possible to separate the computation of variable a and b into two separate methods, since the two *concerns* are *tangled* inside the method.

2.5.2 Program slicing approaches

In [100, 101] the authors propose a slice extraction approach to automatically identify Extract Method refactoring opportunities which are related with the complete computation of a given variable (*complete computation slice*) and the

```
1  private int a;
2
3  public void m(int x) {
4       a = 0;
5       int b = 0;
6       while (someCondition(x)) {
7            b = computeAB(b);
8       }
9       print(a,b);
10 }
11
12 private computeAB(int b) {
13      if (someOtherCondition(x)) {
14                a += 1;
15                b += 0;
16           } else {
17                a += 0;
18                b += 1;
19           }
20      return b;
21 }
```

Figure 2.9: An example of fragment extraction (lines 5-11) for the *m(x)* method shown in Figure 2.6. Since both a and b are output variables for the fragment to be extracted, one of them needs to become a field to let the extraction feasible.

statements affecting the state of a given object (*object state slice*). They extended the block-based program slicing approach proposed in [70] by a set of rules aiming at preserving the code behaviour after slice extraction, solving the problems caused by code duplication and code reordering described in Subection 2.5.1. To prevent the excessive duplication of code in the original and extracted method, slices are sorted by a *duplication ratio*, that is the ratio of the number of statements that will be duplicated after the slice extraction to the total number of statements to be extracted.

Even if smaller slices are desirable, sometimes the extraction of a full slice for a variable can be too large, having a size comparable to the original method. This can happen when the variable is last assigned at the end of the method, and most of the previous statements cooperate to compute its final value. Even though this may appear desirable in most of the cases, in case of long methods, finding a way to split this computation is still desirable. In [1] it has been presented the concept of *fine slicing*, that can produce executable and extractable slices while restricting their scopes and sizes by specifying which data and control dependences to ignore when computing a slice, i.e. the slice boundaries.

On the one hand, slice based approaches can effectively untangle code related to the computation of different variables, even considering complex control sequences. They are also able to extract methods showing a *good concern cohesion*, since performing all the computation steps related to the definition of a specific variable. On the other hand, the difficulty of automatically perform such an extraction while preserving code behaviour may result in a few extraction candidates compared with fragment-based approaches. Moreover, the problem of selecting the variables which indicate a concern of interest remains [98]. Chapter 4 will show a data flow analysis approach to characterise program statements with the concern they address according to the source of the data they are using.

By definition, program slicing is useful to identify *get* methods, i.e. methods providing the value for a particular variable, collecting all the instructions cooperating for the related computation. However, typically it does not address the code related to the *use* of that variable to perform a specific task, without performing any other computations for the variable value. It will be shown how characterising the data coming from *platform specific* APIs (Chapter 3) or *concern specific* APIs (Chapter 4) is useful in order to identify the instruction that use such data to perform respectively *platform* and *concern* related operations, thus suggesting opportunities for code to be extracted.

2.5.3 Fragment extraction approaches

Fragment extraction approaches mainly differ in the way these two tasks are performed: *candidate selection* and *candidate ranking*. The former refers to the way the Extract Method candidates, i.e. block of contiguous statements, are identified. The latter consists in ranking those candidates by means of different factors, often variations of the *minimize coupling/maximize cohesion* design principles. That is, to obtain fragments that encapsulate well-defined computations with their own set of dependencies (high cohesion) and that are also independent of the original method (low coupling) [90]. The highest ranked candidates are then suggested to the developer, who can easily mark and extract them as new methods by means of the refactoring support typically provided by modern IDEs.

Concerning candidate selection, in [90] the authors rely on a hierarchical model, representing the *block structure* of the method source code, to select all possible

sequences of contiguous statements (fragment) satisfying three kind of preconditions: *syntactical* preconditions, which means that the extracted code must be a syntactically valid; *behaviour-preservation* preconditions, thus not leading to an extracted method with multiple return values (as explained in Subsection 2.5.1); *quality* preconditions, such as thresholds on the extraction size, e.g. avoiding poor-quality recommendation because of its small size, or because it is too large, thus covering almost all statements of the original method.

As ranking function, they used the *Kulczynski* set similarity coefficient [29], measuring the distance between two set of dependencies: the ones used by the candidate fragment and the ones used by the remaining statements of the method. Dependency set for a sequence of statements have been determined by considering used variables, types, and packages.

In [118] the authors use a candidate selection strategy that is similar to the one seen in [90], e.g. considering only well-sized fragments, but using also *blank lines* as fragments separation, since these are typically used by the developers to separate relatively independent code portions.

They defined a ranking function to balance between benefit and cost of a candidate fragment extraction. The cost of extraction is given by the coupling, in terms of data, between the fragment and the rest of the method, here given by counting how many parameters are needed by the new method (both input and output). The benefit of the extraction is given by the reduction in the length of the original method. All candidate fragments are sorted according to their benefit/cost ratios in a descending order, thus suggesting the one with the highest ratio.

Independently of the way code fragments are recommended, in case of a long method more than one extraction may be suggested. In Chapter 5 it will be shown how to manage multiple extractions using the *Replace Method with Method Object* refactoring, while optimising the number of fields to be added to the new class.

2.6 Data leak detection in Android applications

Data flow analysis, and in particular *taint analysis*, have been shown to be beneficial in several applications including detecting and preventing data leaks [37, 75, 104, 123], defending against injection vulnerabilities (SQL injection, cross-site

scripting, shell injection etc.) [15, 28, 61, 77, 81, 83, 109, 116], and assisting in the identification of configuration errors [8, 11, 49, 51, 76, 79, 97]. Special attention has been given to mobile devices [5, 9, 34, 37, 48, 59, 115, 122] and to the problem of data security and privacy.

In the context of data security, taint analysis can be performed either statically or dynamically (see Section 2.1.7 for details). The former is sometimes referred in the literature about data leak detection simply with *Static Data Flow Analysis (SDFA)*, where the latter is also known as *Dynamic Data Flow Tracking (DDFT)*.

However, both static and dynamic taint analysis are liable to some limitations (see Section 2.1.7). Among the widespread tools that use a static approach, such as [53], [27], sparta[6], etc., results tend to have a high rate of false positives, requiring a further human check. As of August 2017, more than 3 Millions apps are available on Google Play store[7] therefore a manual check is unfeasible. On the other hand, a dynamic approach [37] may introduce a substantial monitoring overhead, typically by customising the execution environment (such as the OS), significantly reducing the program performances, or even making the the monitoring unfeasible for small devices.

In Chapter 6 the data leak monitoring problem [75, 104, 123] has been investigated proposing an *Hybrid Taint Analysis (HTA)* based approach. This takes advantage of static taint analysis performed offline, and dynamic taint analysis, devising a hybrid approach that selects the part of an Android app that need to be monitored during runtime to avoid data leak. Moreover, it does not need to customise the execution environment (i.e. the Android OS), since all the monitoring capabilities will be automatically injected inside the Android application. These changes will be made at the bytecode level, i.e. without any modifications of the source code, similarly to the work in [11, 49, 51].

In this section, the taint analysis approaches will be introduced in the context of data leak detection. We will describe both the static and the dynamic approaches used in this field, also introducing the hybrid ones. Finally, related work on data leak detection will be given.

[6] http://www.cs.washington.edu/sparta
[7] http://www.appbrain.com/stats/number-of-android-apps

2.6.1 Taint Analysis for data security

STA tools [5, 54, 99] aim at detecting the paths of executions that may propagate data from untrusted or sensitive sources to undesirable sinks, by statically analysing the code. Although static analysis exhaustively evaluates a program, it is prone to produce false positives, since it is impossible to establish whether a discovered path will ever be actually executed, and hence whether data to be protected will reach the sink at some point.

DTA tools [37, 38] monitor the execution of a program and track the variable that contains (in a raw or elaborated form) information coming from a given source. Tracking can be performed by means of different approaches, including system emulators [30, 58, 81], API modification [28], code injection [62, 83, 116] and ad-hoc hardware [33, 95, 108]. The main disadvantage of dynamic taint analysis is that the tracking logic introduces an overhead that significantly reduces the performances of monitored programs.

Even though dynamic taint analysis performs better than the static one for what concerns false positives, tools that use this approach, such as [38], [117], [111] and [4], often require either a custom Android version that introduces low level detection mechanisms or to modify the source code of apps to call external APIs with the same purpose, making the analysis itself inapplicable on a large scale. For instance, DroidScope [117] exports three tiered APIs that refer to three different layers (hardware, OS and Dalvik VM) of an Android-based device. Specific plugins analyse different aspects of the app to check potential information leaks.

2.6.2 Related Works

A few recent approaches combine static and dynamic analysis to overcome the limits of both approaches. Chang et al. [25] proposed a method that combines static and dynamic analysis for efficiently monitoring C programs. It first performs static analysis to filter out data that need not dynamic tracking, then modifies the C source program to add the appropriate tracking code. The tracker monitors tainted data at byte-level granularity. Our approach is similar in spirit in that we employ static analysis to minimize the amount of tracking code needed.

A major difference is that we target compiled Java bytecode in place of C

source code, hence making our approach viable when source code is unavailable. Moreover, Java programs are intrinsically different from C programs and present new challenges, e.g. the management of the operand stack. On the other hand, Java has a more rigorous semantics that simplifies tracking and therefore allows us to greatly reduce the overhead. E.g., we avoid byte-level tracking in favour of a more efficient instruction-based tracking that implicitly tracks data flow at a variable level.

Jee et al. [60] proposed a technique that can be used in combination with other DDFT tools in order to reduce redundant tracking logic and thus optimizing the target program. After a DDFT tool has introduced a tracking logic into the binaries of a program, their tool separates the program logic from the tracking logic and then applies known optimization techniques on the tracking logic. In contrast, our approach generates the minimum amount of tracking code because of the reasoning performed on the results of static analysis.

Zhang et al. [120] proposed to optimize DDFT by analyzing the documentation and source code of APIs to find taint propagation data. They then associate the input parameters of a function with the corresponding outputs, and determine the APIs that need not be tracked. Their tools can be used for x86 binaries or C programs. Our approach performs a wider spectrum optimization on both APIs and application tracking code and makes no assumption on the behaviour of APIs. Moreover, our tool targets Java bytecode.

In the context of Android applications, AppCaulk [88] can identify data leaks at run-time by a dynamic taint analysis based on a code injection inside the target app. Mobile Sandbox [93] scans permissions and manifest file during the static analysis, while using an emulator at runtime to look for suspicious calls and network traffic. Andrubis [112] performs static analysis on the app bytecode and the manifest file and uses the results at runtime by performing several analyses like taint tracing, method tracing, on both Dalvik VM and system level.

COMBdroid [32] is a security tool allowing users to enforce fine-grained policies to prevent Android applications from starting phone calls, sending SMS messages, and accessing web URLs without the user awareness, prompting the users to choose whether to trust the recipient number or URL. Fine-grained polices are dynamically defined by the user by means of a *white list* and a *black list* for recipient numbers

and URLs, hence letting COMBdroid to automatically take further restriction decisions.

The monitoring behaviour is introduced in the Android application by replacing the `Activity` class with a modified one, thus only requiring the application to be instrumented instead of the Android OS. The rational is that all the data flow related to the three considered scenarios (sending SMS messages, starting phone calls, accessing web URLs) pass through the methods of the `Activity` class. Therefore, by overriding this class it is possible to intercept these data flows from a centralised manager.

This approach is similar to the one proposed in Chapter 6 in that both do not require the Android OS to be customised. However, in COMBdroid data flows have to pass through the `Activity` methods to be intercepted, which is an assumption that holds for the considered scenarios, but in general data leaks may occur using also different paths inside the application. Our approach lets the user to define which API usages to consider as sources of sensitive data and which to consider as sinks for untrusted destinations without making any assumption on the data path that will connect them, thus causing the data leak.

2.7 Conclusions

In this chapter all the fundamentals and related works, to understand and evaluate the contributions presented in further chapters, have been presented. A great importance has been given to concepts related to data dependence, taint analysis and refactoring, since needed to support the work presented in Chapter 5 for suggesting complex refactoring opportunities, that is the main objective of this thesis.

Chapter 3

Platform Dependence and Portability assessment

Code portability is a desirable non-functional requirement. The most established metric evaluating it consists of counting the number of platform dependent instructions, i.e. that use platform specific APIs. Generally, instructions using APIs are preceded or followed by related code that e.g. prepares some input for a call or analyses the return value. In this chapter a taint analysis approach will be proposed, that is used to identify code portions that are related by means of data dependence and are connected to a given platform. By considering both direct calls to platform APIs and the data dependence, the proposed approach provides a more precise and realistic indication of the code that should be considered in case of platform migration. Results about the analysis of an industrial application will be also given, thus proving the validity of proposed metric and tool for the portability assessment.

3.1 Assessing code portability

Applications depend on an underlying platform in that they use platform specific APIs to interact with their environment. In order to hit the heterogeneous and fast changing market of digital devices and IT infrastructures, often application code has to be *ported* to different platforms. Hence, it is important for the developers to leverage both meaningful metrics to measure the degree of code portability, and

practical tools assisting them in the platform migration process, by spotting the application points that are platform dependent.

Code portability is broadly recognised as a desirable *non-functional* requirement for software products, and there are a few contributions in the literature providing a way to assess it. The most practical and established portability metric consists of counting the number of instructions that use types defined in platform specific libraries or APIs [73, 92], such as e.g. method calls to access the file system or parse an XML data stream, etc.

Further portability metrics have been proposed, stated as "the percentage of target dependent statements" [92], and as "how much modifying and adapt the code to a new environment is cheaper than redevelop it" [73]. The above metric definitions suggest that the first step in assessing code portability is to identify the program points that are *bound* to an underlying environment. However, the traditional counting of program points using platform APIs would only provide an under estimation of the real number of program points that can be affected by changes in case of migration. Typically, API calls are followed, or preceded, by a code portion using some *supporting libraries* preparing data to be passed or that have been collected from the platform dependent method calls. Therefore, such code portions are likely to be changed should an API interaction be removed due to platform migration. Only by considering the relevant *data dependent* application code we can have an actual indication of the code percentage that is platform dependent.

This chapter will describe a solution aiming at assisting the developer during the porting of an application from a platform to another. The proposed approach considers a list of APIs that the developer denotes as platform dependent, i.e. a set of packages, classes or methods that are used to interact with the environment s/he wants to move from. The approach is based on *data dependence* [119] and *taint analysis* [87] (both introduced in Section 2.1 and used in Chapter 6) to automatically identify the program points referring such APIs, then using each of them as a *source* of a *data flow* we can find all the other instructions that are having a *supporting* role for the platform usage, and all other platform-related connections. The resulting set of program points is the portion of the application that has to be adapted in case of platform migration.

The proposed approach is innovative in that, on one hand, it leverages the developer point of view, asking him/her to define the target APIs to consider as a source of platform dependence. On the other hand, it automatically finds all the lines of application code that are using target APIs or related ones. Therefore, the approach additionally provides a tool that accordingly measures the portability of an application, and gives exact indications on where the dependent code is. Moreover, the said analysis can promote the use of other tools supporting a platform migration [41] or suggesting *refactoring opportunities* to enhance portability [44], e.g. by moving platform dependent code inside more *concern specific* modules using appropriate design patterns, such as the *Abstract Factory* or *Bridge* [45, 79]. Furthermore, it can be used to find portions of code that being dependent on partially trusted external types would be better confined and checked, by means e.g. of a defensive approach, in order to improve security, privacy, robustness, etc.

3.2 Limits of the traditional approaches

The approach described in this chapter mainly consists of two steps: (i) *finding* direct usages of a platform APIs that *bind* the code to the platform; (ii) finding the program parts affected by data related to the use of APIs, hence propagating the API dependence to other *supporting* instructions due to a chain of *data dependence*. Such a propagation achieves a more accurate and realistic evaluation of the platform dependent lines of code.

The proposed approach is more refined than other estimation methods, such as counting the lines of code binding given APIs, or considering whole portions of code related to a control flow path. By using data dependence we find all the lines of code that are participating into the preparation of a call to a given API, e.g. for setting up input parameters or the appropriate context for the call; and the lines of code that are using the results that the API call has provided. Hence, such lines of code are dependent on the API as if it was a call to the API itself. Ultimately, such other lines of code would have to be changed should the call to the API be removed. Moreover, data dependence let us find all (and only) the lines of code having such a relation with a given API, independently of the number of lines that can be in between the preparing code (or the code using the result) and the actual

call to the API.

Conversely, counting all the lines of code (e.g. in the same method) surrounding the call to a given API would not distinguish the instructions that are unrelated to the preparation of the call. Moreover, counting all the lines where e.g. the return type of the API calls appear would result in including all the instructions that use it in another context, unrelated with the given API.

3.3 Taint Analysis to assess platform dependence

Before talking about portability, we need to provide the definition of *platform* that will be used for the rest of this chapter.

Definition 2. *Platform* (or *environment*) refers to the complete collection of external components an application interacts with. This may include software components, operating systems, hardware, remote systems or data storage.

Generally, an application has some portions of code that call platform APIs. For an object-oriented language, such as Java, APIs refer to either classes or methods. Our aim is to assess the dependence of an application on specific parts of the platform, i.e. the ones involved in a given *migration process*. This calls for a definition of *migrating platform*.

Definition 3. Given a *platform* and a *migration process*, with *migrating platform* we refer to the subset of platform APIs that have to be replaced during the migration.

For the sake of our analysis, we are only interested on classes or methods used to interact with the migrating platform, introducing the definition of *Binding Set*.

Definition 4. Given a *migrating platform*, with *Binding Set* we refer to the APIs that can be used by an application to interact with the migrating platform. This can be defined by using elements having different granularity, such as an entire package or group of packages (*library*), a single class (*type*) or even a single method (*operation*).

```
1  Binding Set: java.nio.file.Files.newInputStream(Ljava/nio/file/Path;
2      [Ljava/nio/file/OpenOption;)Ljava/io/InputStream;
3
4  public static void useFile() throws IOException {
5   Path p = FileSystems.getDefault().getPath("dir", "f.csv"); // -1 [java.nio.file] *
6   InputStream in = Files.newInputStream(p);                  // source *
7   BufferedReader reader =                                    // +2 [java.io] *
8    new BufferedReader(new InputStreamReader(in));            // +1 [java.io] *
9
10  String line = null;
11  while ((line = reader.readLine()) != null) {               // +3 [java.io] *
12   // input format: id,username,income,outcome
13   String[] tokens = line.split(",");                        // +4 [java.lang] sink
14   String id = tokens[0];                                    // +5 []
15   String uname = tokens[1];                                 // +5 []
16   int income = Integer.parseInt(tokens[2]);                 // +5 [java.lang]
17   int outcome = Integer.parseInt(tokens[3]);                // +5 [java.lang]
18   // process data
19   String uname_up = uname.toUpperCase();                    // +6 [java.lang]
20   int balance = income - outcome;                           // +6 []
21   // output
22   System.out.println(id+","+uname_up+","+balance);          // +6 [java.lang,java.io]
23  }
24   in.close();                                               // +1 [java.io] *
25
26 }
```

Figure 3.1: An example of platform dependence assessment: the method uses a file based storage to access the user data. Tags have been added to show the data dependence from the instruction directly accessing the *migrating platform* (line 6). Lines marked with a '*' symbol are considered by the proposed approach as platform dependent.

3.3.1 Tracing the data flow

Let us consider a simple use case: the left side of Figure 3.1 shows the code of a method, dubbed `useFile()`. This firstly loads some data from a file, where each line contains the bank account information for a specific user, i.e. income and outcome. Then, it processes such data and for each user prints the balance to the standard output. Similar portions of code can be found in many applications and other methods on the same application can have a similar behaviour, i.e. they use the data loaded from a file-based storage.

Now, we analyse the scenario where a change on the underlying platform has occurred, and data have been moved from files to a *database*. In this case, the developer needs to spot where the methods have to be modified, changing the data loading code, while preserving the behaviour.

For the said example, the *migrating platform* will be the *file-based storage*, and the *Binding Set* will consider the `Files.newInputStream(path)` as a platform dependent API. Given such a Binding Set, a typical approach will only indicate line 6 as platform dependent, but this is not accurate. In fact, looking at the code we can identify mainly two parts having different *concerns*: lines from 5 to 11 and line 24 are implementing the concern of loading data from an input file, which will be modified to use a database instead; and lines from 12 to 23 (ignoring both comments and brackets) implementing the specific application concern that we want to preserve.

The first part of the code is actually platform dependent, and will be probably modified during the migration process. The objective of the proposed approach is to automatically highlight these program points, thus not limiting our platform dependence assessment to line 6 like a traditional approach. Instead, we will use this *explicit* platform dependent program point as a *source* of dependence that can be extended to the surrounding lines due to their *data dependence* (see Definition 1 in Section 2.1), which we say having an *implicit* platform dependence.

Definition 5. Given a *Binding Set* B, a program point has an *explicit* platform dependence if it directly accesses an API in B, i.e.: it (i) calls a method in B, or (ii) a method whose return type is in B; (iii) uses a field defined in a class (type) in B, or (iv) a field whose type is in B;

Definition 6. Given a program point p having an explicit platform dependence, we say that a program point v has an *implicit* platform dependence if it uses *non-common* and *non-internal* APIs to either prepare data further used by p or to operate on data previously defined by p. That is, v is either on a backward or forward data flow path that has *source* in p. We call the set of APIs (i.e. methods and classes) used in this path *Supporting Set*.

An *internal* API is a type or method that does not belong to an external library. *Common* APIs consider types like `String`, `Integer` or `List`, which are available from the Java standard library and are too general to be considered platform dependent (see Section 3.3.2 for more details).

In the `useFile()` method (see Figure 3.1), we can see that the explicitly dependent instruction on line 6 *uses* (i.e. reads) variable `p` and *defines* (i.e. writes) variable `in`. Therefore, if we trace the *backward data flow* from this program point, we can see that only line 5 previously defines the `path` variable, using type `FileSystems` from *java.nio.file* package. Moreover, if we trace the *forward data flow* from the source, we can see that the next instructions using the `in` variable are on line 24 (which closes the stream), and on line 8, where data flows in sequence through the `InputStreamReader` and the `BufferedReader` types of *java.io* package. Then, at a distance of two, in terms of code lines being traversed by data flow (instead of control flow), we find line 7 which defines `reader` variable by storing the result of the previous step. We can continue the forward data flow tracing by following the *define-use* cascade of relations, giving the resulting labels shown in Figure 3.1 ($\pm$ distance [used packages]).

As a result of using data flow analysis we labelled all the lines that are *data dependent* from the selected *source* (thirteen lines), but only some of them are actually *implicitly dependent* program points. In fact, the labelled lines from 12 to 23, as we previously said, cover the application concern that the developer will need to preserve during the platform migration. Therefore, using only a data flow analysis without applying some stopping conditions the given results would not be accurate.

3.3.2 Taint analysis algorithm

Looking at the tagged lines (from 16 to 26 in Figure 3.1), we can see that all of them are actually using either *common types*, like `String` (13, 19) and `Integer` (16, 17), or array and arithmetic operations (14, 15, 20). Even though these instructions are data dependent from the source, they cannot be related to any platform, since they are too common. This is also true for most of the instructions that use types defined in *java.lang* or *java.util* packages (like `List` or other common data structures). Although, there could be some exceptions: line 22 uses `System` to access the standard output. This is actually a common type, however in some cases this can not be ignored, e.g. when the developer has to port the application to the web or for batch execution. Therefore, the analysis must consider the developer point of view to determine the contents of the *Common Set*.

To improve the precision of the data analysis, the idea is to stop tracing the data flow when we encounter a source line which can not be related to any specific platform, such as a line containing no API usages or calls to APIs defined in the Common Set. To solve this problem we have used a *taint analysis*-based approach.

In Section 2.1.7 the concept of *taint analysis* has been introduced. It is a data flow analysis used to solves problems where there are *sources* for *tainted* data that need to be traced until they flow into one of the possible *sinks*. As shown in Chapter 6, this particular kind of data flow analysis is typically used to solve problems related to *data security* [5, 7], where we want to see if data coming from sensitive sources can flow to specific unauthorised channels (i.e. sinks).

We can use the same approach to taint the implicit platform dependent program points. We use each explicit dependent program point as a *source*, and we define two separate classes of sinks, *active* and *passive*. The former class will always stop the data flow on a line containing a method call to an *Internal Type*, i.e. a type that is not defined in an external library. The latter class, *passive*, only stops the data flow on a line containing common operations, i.e. array or arithmetical instructions or method calls to APIs in the *Common Set* (see Section 3.3.4 for more details).

Using this taint analysis approach with sources and sinks so defined, in the said `useFile()` method (Figure 3.1) we identify line 13 as a passive sink to be activated (because only *java.lang*, i.e. a library in the *Common Set*, has been used) stopping the data flow tracing. The six tainted lines (marked with a '*' in the example) are

```
1  Method code ported to the target platform
2  using a database instead of files as data storage
3
4  public static void useDatabase() throws SQLException {
5   Path p = FileSystems.getDefault().getPath("input", "operations_db");
6   String db = "jdbc:derby:"+p.toAbsolutePath()+";user=u;password=p";
7   Connection conn = DriverManager.getConnection(db);
8   Statement stmt = conn.createStatement();
9
10  ResultSet row = stmt.executeQuery("SELECT * FROM OPERATIONS");
11  while (row.next()) {
12   //input format: id,username,income,outcome
13
14   String id = row.getString(1);
15   String uname = row.getString(2);
16   int income = Integer.parseInt(row.getString(3));
17   int outcome = Integer.parseInt(row.getString(4));
18   // process data
19   String uname_up = uname.toUpperCase();
20   int balance = income - outcome;
21   // output
22   System.out.println(id+","+uname_up+","+balance);
23  }
24  stmt.close();
25  conn.close();
26 }
```

Figure 3.2: This is the ported version of the *useFile()* method: *useDatabase()*. It uses a database storage instead of files to store user information. All the platform dependent lines selected by the taint analysis approach for *useFile()* (from line 5 to line 11 and line 24 in Figure 3.1) have been actually modified due to the platform migration, thus not only the line directly accessing the file based storage (line 6 in Figure 3.1, i.e. the only one with an *explicit* platform dependence). This confirms the need for considering also the *implicit* platform dependences in order to correctly assess the method portability.

the ones considered as platform dependent by the output of the analysis. As a result of the taint analysis for the said method we obtain the *Supporting Set*, which is composed by `FileSystems`, `BufferedReader`, `InputStream` and `InputStreamReader`. It does not consider types like `String` and `Integer` because they are listed in the *Common Set*.

Figure 3.2 shows the `useDatabase()` method, which is the ported version of the previous one using the database instead of files. We can see how all of the tainted lines were actually modified due to the porting process. More lines of code were changed, e.g. lines 13 to 17, which are the ones related to the parsing operations. Due to the differences between the two involved platforms, we can not predict which

```
Find lines using either a binding or a supporting API:
 (binding) java.nio.file.Files.newInputStream(Ljava/nio/file/Path;
      [Ljava/nio/file/OpenOption;)Ljava/io/InputStream;
 (supporting) InputStream, InputStreamReader, BufferedReader, FileSystems

public static void useSocket() throws IOException {
 Socket socket = new Socket("127.0.0.1", 33333);
 InputStream in = socket.getInputStream();               // tagged
 BufferedReader reader = new BufferedReader(             // tagged
    new InputStreamReader(in, StandardCharsets.UTF_8));  // tagged
 String line = null;
 while ((line = reader.readLine()) != null) {            // tagged
   processSingleLine(line);
 }
 socket.close();
}
```

Figure 3.3: A method that uses *supporting* types `InputStream` and `BufferedReader` but that has no dependence on the file storage platform. This is an example of how looking for supporting type usages without considering the data dependence could lead to false positives.

other lines could be added or modified to *adopt* the *specific target* one. Therefore, our assessment can be seen to give the *lower bound* of the number of lines to be changed, spotting the lines that the developer will have to modify to *leave* the current migrating platform, independently of *any target* one.

3.3.3 The Supporting Set

Let us suppose that the developer knows which supporting types form the *Supporting Set* (even though this can be a significant assumption in real cases), and decides to use a traditional approach to determine which lines of code have to be ported, i.e. searching for type usages instead of tracing the data flow. In this way the obtained results for the `useFile()` method are similar to ours, however some false positives will be detected in the `useSocket()`. Figure 3.3 shows the `useSocket()` method, belonging to the same application as `useFile()`, and using buffers to read data from a socket, which in our scenario is not part of the migrating platform. In this scenario, a simple search for a set of types cannot correctly assess platform dependence, since they are not related to file management, but used for stream and buffer handling. Then, such types should be considered as platform dependent only if they are supporting a data flow started from an access to the file

```
1 Active sinks example:              Passive sinks example:
2 d = binding()        S             d = binding();      S
3 x = internal(d)      K             x = common(d)       K
4 y = api(internal(d)) K             y = api(common(d))  T
5 z = internal(api(d)) T,K           z = common(api(d))  T
6 api(x)                             api(x)
7 api(y)                             api(y)              T
8 api(z)                             api(z)              T
```

Figure 3.4: Two examples showing how active and passive sinks are used to stop the data flow (S: source, K: sink, T: tainted)

storage.

3.3.4 Active and passive sinks

In a taint analysis approach, sinks are needed to stop the data flow coming from a data source. In our case, the flow will be stopped when we recognise that further instructions cannot be assumed as platform dependent. Given a program point p on source line l, we consider it:

- an *active sink* if it calls an API belonging to an *Internal Type*, i.e. a type that is not defined in an external library;
- a *passive sink* if it is the last program point traversed by the current data flow before living l, which is *activated* only if no other program point was tainted on this line. Otherwise, it can be traversed by the flow.

In the first case, if the data flow tries to taint an active sink, we must always stop our tracing. This because we cannot bind this sink and any further instruction to any platform, since data have been processed using an internal API that we must consider as *application dependent*.

In the second case, if by traversing a source line we met a common operation but its result is immediately passed to an API not in the *Common Set*, we can consider the former operation as a weak stop condition, and we should still tag this line as platform dependent. On the other hand, if we are trying to leave a source line composed only by common operations, we should stop our analysis since we have probably met the starting of a code portion related to an application-specific concern.

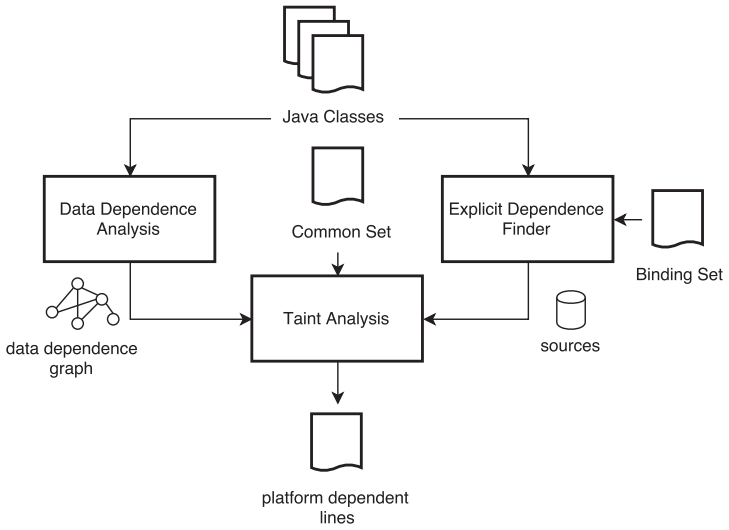

Figure 3.5: The taint analysis framework for portability assessment.

Figure 3.4 shows how active (on the left) and passive (on the right) sinks are identified and used to stop the data flow. In the example on the left, starting from the *binding* method (the source), the data flow will be stopped on line 3 and 4 since a call to an *internal()* method has been reached. This also happen on line 5, but just after the execution of an *api()* method, i.e. neither a common nor an internal method. In this case, the line will be considered as tainted, and then the flow will be stopped. In the example on the right, the flow is stopped by the *common()* method call because it is the last instruction on the line (i.e. the passive sink) and no other instruction has been tainted on the same line. On line 5 the flow is not stopped by the *common()* method call, because an *api()* call has been reached, and therefore tainted, on the same line. This is also true for line 4. Thus, also line 7 and 8 will be tainted by the data flow.

3.4 Framework implementation

Figure 3.5 shows the overall workflow used to implement the portability assessment approach described in Section 3.3, which is also formalised in Algorithm 1.

Starting from the Java application classes, the *explicit dependent* program points are identified, which will be used as sources by the taint analysis. This is

performed by the `Explicit Dependence Finder` component, which implements the *isSource(p)* procedure (shown in Algorithm 1) by inspecting the bytecode of input classes using the Byte Code Engineering Library (BCEL[1]). According to the *Binding Set*, that is provided by the developer, for each byte code instruction the component determines whether it is an explicit dependent program point, i.e. a source (see the definition in Section 3.3). Dependent program points are found when an opcode `INVOKE`, `GETFIELD` or `PUTFIELD` is used to directly refer a class or method in the *Binding Set*.

The identified bytecode instructions are considered as *sources* for the second step, that is performed by the `Taint Analysis` component. For each method of the input application, it executes the taint analysis algorithm by implementing the *taintAnalysis(m)* procedure shown in Algorithm 1.

To perform the analysis, a *Data Dependence Graph* (DDG) is needed, which is provided by the `Data Dependence Analysis` component. There are several frameworks for the static analysis of Java byte code (see Section 2.1.5), which can produce a DDG from an application, e.g. Soot [65] or Wala[2]. We used instead JavaPDG [89], a powerful tool that analyses Java bytecode to provide different types of program dependences representations [40] (e.g. the *control dependence graph*) given as a relational database. For the produced DDG, each node is a byte code instruction and each edge is a *definition-use* relation. For each node, other data are also given, such as the corresponding line position in the source file.

Having both the sources and the DDG, both the *backward* and the *forward* taint analysis can be performed, giving as a result a list of the *platform dependent lines* that can be used to tag the Java source code. To perform this last step, the definition of a *Common Set* is also needed, which by default considers all the types in *java.lang* and *java.util* packages, and which can be modified by the developer according to the specific analysis.

[1] https://commons.apache.org/proper/commons-bcel/
[2] http://wala.sourceforge.net

Algorithm 1 The Taint Analysis algorithm for platform dependence assessment

Require: init()
1: B := *BindingSet*
2: $S := \emptyset$

Require: tainAnalysis(m)
3: **for all** program point $p \in$ method m **do**
4: **if** isSource(p) **then**
5: $S := S \cup \{p\}$
6: **end if**
7: **end for**
8: G := getDataDependenceGraph(m)
9: backwardFlowAnalysis(G,S)
10: forwardFlowAnalysis(G,S)

Require: isSource(p)
11: **if** isInvoke(p) **then**
12: **if** calledAPI(p) $\in B$ **or** returnType(p) $\in B$ **then**
13: **return true**
14: **end if**
15: **else if** isFieldAccess(p) **then**
16: f := accessedField(p)
17: **if** type(f) $\in B$ **or** belongsTo(f,T) $\mid T \in B$ **then**
18: **return true**
19: **end if**
20: **end if**

Require: isActiveSink(p)
21: **if** isInvoke(p) **and** calledAPI(p) $\in$ *InternalTypes* **then**
22: **return true**
23: **end if**

Require: isPassiveSink(p)
24: **if** p is the last program point on current source line **and** no program point on current flow was tainted on this line **then**
25: **return true**
26: **end if**

Require: backwardFlowAnalysis(G,S)
1: $\overline{G}$:= revert(G)
2: forwardFlowAnalysis($\overline{G}$)

Require: forwardFlowAnalysis(G,S)
3: **for all** $s \in$ S **do**
4: *taint* program point s
5: **output** program point s is platform dependent
6: $V := \emptyset$
7: *Stack* := $\emptyset$
8: *Stack*.push(s)
9: **while** *Stack* is not *empty* **do**
10: p := *Stack*.pop()
11: **if** $p \notin V$ **then**
12: $V := V \cup \{p\}$
13: **if** isActiveSink(p) **or** isPassiveSink(p) **then**
14: stop tracing current data flow
15: **else**
16: **if** isInvoke(p) **and** calledAPI(p) $\notin$ *CommonSet* **then**
17: *taint* program point p
18: **output** program point p is platform dependent
19: **end if**
20: **for all** program point v adjacent to p in G **do**
21: *Stack*.push(v)
22: **end for**
23: **end if**
24: **end if**
25: **end while**
26: **end for**

3.5 Experiments

Let us consider an actual portability assessment use case by analysing *Clever* [102], an agent-oriented open source cloud middleware[3] implemented in Java, consisting of 33144 lines of code, 1645 methods, and 225 classes. Since it manages different aspects of a distributed cloud infrastructures, it is based on different external libraries, like *libvirt*[4] to interact with the hypervisors, *sedna*[5] to implement the

[3]https://github.com/clever-unime/clever-unime
[4]https://libvirt.org/java.html
[5]http://www.cfoster.net/sedna/

Java File	find Document	+ support methods	+ sup. types	taint analysis
HvlibVirt	7	49	127	15
ParserXML	7	18	31	11
CleverMessage	3	23	36	7
Guacamole	3	15	31	3
SensorBrain	2	5	10	5
ExecOperation	1	8	16	3
NotificationOperation	1	8	15	3
RequestInformation	1	8	17	3
ErrorResult	1	6	13	3
OperationResult	1	6	13	3
Result	1	2	10	3
ClusterCoordinator	0	3	7	0
HostCoordinator	0	2	5	0
ImageManagerAgent	0	2	2	0
DatabaseManagerAgent	0	1	1	0
VirtualizationManagerAgent	0	1	1	0
ModuleFactory	0	1	2	0
HyperVisorAgent	0	1	1	0
ServiceManagerAgent	0	1	1	0
DbSedna	0	0	5	0
VirtualizationManagerClever	0	0	1	0
HvVirtualBox	0	0	1	0
HvVMWare	0	0	11	0
Total	28	160	357	59

Table 3.1: Evaluation of the platform dependence of *Clever* [102] from the `org.jdom.Document` type.

persistence layer with an XML database, and *smack*[6] to handle the communication between the cluster nodes using the XMPP protocol. Therefore, several of its parts manage XML data streams, e.g. to load a description file of a virtual machine, to parse messages received from other nodes, or, simply, to load some configuration files. To handle XML formatted text, the *jdom*[7] library is used. For the developer to obtain a better control on how XML documents are loaded or stored, a limited direct use of type `org.jdom.Document` is desired, then we assess how many lines of code depend on it.

In this case, the *Binding Set* consists of the said `Document` class, whereas the *Common Set* is the default and consists of all the types in *java.lang* and *java.util* packages. Table 3.1 shows the results of our taint-based approach (fifth column) and a simple search (second column) for class `Document`, giving the lines directly using it, e.g. due to an attribute access or a method call (either as the target class

[6]https://www.igniterealtime.org/projects/smack/
[7]http://www.jdom.org

```
1  public String getStringedSubTree(String element)
2    throws JDOMException, IOException {
3   String s = "";
4   Element elem = rootElement.getChild(element);                    *
5   Iterator lst = elem.getDescendants();
6   int i;
7   Element e = document.detachRootElement();                     $ x *
8   e.removeNamespaceDeclaration(Namespace.NO_NAMESPACE);           x *
9   Element e2 = e.getChild(element);                               x *
10  XMLOutputter xout = new XMLOutputter();
11  Format f = Format.getPrettyFormat();
12  xout.setFormat(f);
13  return ((xout.outputString(e2).replaceAll("<"+element+">", ""))
        .replaceAll("</" + element + ">", ""));                     x *
14 }
```

Figure 3.6: A comparison of the tagging results given by the *find Document* approach ('$'), our *taint analysis* ('x') and the *find support methods* approach ('*').

or the return type). The simple search found 28 direct uses of `Document`, while our approach found a higher number of platform dependent source lines, i.e. 59 on twelve files.

By setting as a source for the taint analysis the lines using `Document`, we have found several code lines having an `Element` instance returned by the sources. Hence, when substituting `Document`, all the *data dependent* lines using `Element` to access the document content have to be changed as well. For example, Figure 3.6 shows a method in the *ParserXML.java* source file: only line 7 has been considered as platform dependent by the *find Document* approach (results are marked with '`$`' symbol), because it calls a method of the `Document` class[8]. By setting this line as a source for the taint analysis approach, we have found several code lines (marked with the '`x`' symbol in the figure) using the `Element` instance returned by the method call in line 7, thus exhibiting an implicit platform dependence.

Figure 3.7 shows the *Supporting Set* found during the taint analysis, which includes all the types and methods hit by the tainting data flow. It consists of 17 methods belonging to 6 types (other than `Document`), i.e. 4 in *jdom* library (`Element`, `SAXBuilder`, `XMLOutputter` and `Attribute`), 1 in *libvirt* (`Domain`) and 1 in *java.io* (`StringReader`). Only 4 of such 17 supporting methods show `Document` in their signatures, therefore they are the only ones that would be found by searching

[8]Of course the search has been performed by an appropriate tool on the byte code, which embeds for source line 7 the opcode for a method call and an indication of type `Document`.

```
java.io.StringReader
    .<init>(Ljava/lang/String;)V
org.jdom.Attribute
    .getValue()Ljava/lang/String;
org.jdom.Element
    .<init>(Ljava/lang/String;)V
    .addContent(Lorg/jdom/Content;)Lorg/jdom/Element;
    .detach()Lorg/jdom/Content;
    .getAttribute(Ljava/lang/String;)Lorg/jdom/Attribute;
    .getAttributeValue(Ljava/lang/String;)Ljava/lang/String;
    .getChild(Ljava/lang/String;)Lorg/jdom/Element;
    .getChildren(Ljava/lang/String;)Ljava/util/List;
    .removeNamespaceDeclaration(Lorg/jdom/Namespace;)V
org.jdom.input.SAXBuilder
    .<init>()V
    .build(Ljava/io/File;)Lorg/jdom/Document;
    .build(Ljava/io/Reader;)Lorg/jdom/Document;
org.jdom.output.XMLOutputter
    .output(Lorg/jdom/Document;Ljava/io/OutputStream;)V
    .outputString(Lorg/jdom/Document;)Ljava/lang/String;
    .outputString(Lorg/jdom/Element;)Ljava/lang/String;
org.libvirt.Domain
    .getXMLDesc(I)Ljava/lang/String;
```

Figure 3.7: Supporting Set (with types and methods) found during the taint analysis.

type `Document`.

By comparing the results, shown in Table 3.1, of the simple search and the taint-based analysis, we can see that the *recall* in spotting platform dependent lines has been greatly improved by the proposed approach. For evaluating *precision*, we consider that a simple search is performed for the additional classes in the *Supporting Set* but `StringReader`, since too generic and leading to many false positives. The fourth column of Table 3.1 gives the results, and shows that the number of found lines (357) is considerably higher than the previous two approaches. Code inspection reveals that this is mostly due to the use of class `Domain` in `HvlibVirt.java`, which manages configuration data for a virtual machine, however `Domain` is related to `Document` only in a few cases, e.g. when xml files are loaded from the file system. In other cases, found lines are only related to `Element`, since the `ParserXML` class accesses the xml document, retrieves the content, and passes it on. In our scenario, this behaviour will be kept unchanged.

As a final comparison let us suppose that the developer can be more precise in giving the supporting elements to be used by the *find* approach, i.e. indicating methods instead of types. Third column in Table 3.1 shows the corresponding

results. The number of found lines is still considerably higher than the taint-based approach, revealing lower precision compared to ours.

In the example shown in Figure 3.6 the lines tagged by the *find support methods* approach are marked with the '*' symbol. We can see that line 4 has been considered as platform dependent due to the call to `getChild()` method of `Element` class. However, the access to `Document` class in line 7 (which is the source of the platform dependence), and the access to `Element` class in line 4, are on two different data flows that are not related to each other. Therefore, marking line 4 as platform dependent would be a false positive, and we can see how this was correctly avoided by the the proposed taint analysis approach (in fact, line 4 is not 'x' marked).

The above analysis can suggest a possible refactoring operation, i.e. moving the responsibility of accessing the `Document` type only within `ParserXML` class, thus reducing the platform dependence, using the tagged lines to easily spot the parts to be modified.

3.6 Conclusions

In this chapter a taint analysis approach to improve platform dependence assessment was proposed. Once the target APIs to consider as a platform dependence are defined, the approach automatically inspects the static code and finds all (and only) the data dependent code portions, independently of how far apart are these. The efficacy of taint analysis for portability assessment was experimentally evaluated on an industrial application, exhibiting a better accuracy than a traditional approach.

Chapter 4

Concern Tagging and Refactoring

Modularity of a software system can be assessed once responsibilities of each method and class have been determined. Generally, developers attribute responsibilities to methods and classes manually. This can be problematic given that it relies on developers judgement and effort and as a result responsibilities can be coarse-grained, and imprecise. In this chapter an approach for automatically attributing concern tags to each instructions will be presented. The approach is based on taint analysis to determine which code lines are related to each other by data dependence. Moreover, indirectly Java APIs provide us the tags used to mark code lines. The devised automatic concern tagging approach is used to find out how responsibilities are spread over the code, and then to suggest refactoring activities in case tangling occurs.

4.1 Concern Tagging

Generally, developers strive to achieve modularity for their object-oriented systems, and make use of several solutions and techniques for helping them balance their design choices. One of the main problems arising when evaluating modularity is characterising the code portion with the concern it addresses. Some approaches let the developer name the concern for a code portion, then provide some assistance for modularity analysis, other approaches identify addressed concerns according to variable names [10, 82].

Naming a concern can be cumbersome for developers, since they have to provide

it as fine-grained as possible, and, as for commenting code, they can pay little or no attention to it. Of course, the more coarse-grained, i.e. a concern name given to a method, or a class, the less useful it is to assess modularity. Moreover, such a naming of concerns, as well as identifying concerns according to variable names, can be as precise as the naming convention followed by developers. An automatic and robust characterisation of each line of code with its concern would be greatly useful, since it would unload developers of such a task. Moreover, it would made it possible to have further automatic assessments, such as suggesting a more modular decomposition of a method, a class, etc.

A few previous works have attempted to automatically characterise code lines with concerns (see Section 2.2). One of the previous robust approaches is based on the use of the APIs, which name and encapsulate the support for handling some lower level concern [97]. However, only a portion of the lines of code, such as those using the APIs, can be characterised so far.

This chapter will describe *DeDuCT: Data Dependence based Concern Tagger*, that is an automatic approach aiming at characterising the single lines of code with proper concern tags. This in turn provides a high level representation that can ease code comprehension. The proposed approach is based on the list of used APIs giving both an indication of the concerns and the *concern seeds* that are then spread on the code according to data dependencies. APIs are the set of packages, classes or methods that platforms and standard libraries provide to support developers solving a problem related to a specific task, e.g. *io operations*, *string manipulation*, or *exception handling*. Then, by using data dependence [119] and a taint analysis [87], DeDuCT automatically identifies the lines of code related to APIs, since each of them is used as a *source* of a data flow that is traced to tag all the other instructions that have a supporting role, thus sharing the same concern as the one indicated by the API.

As a final result, the set of concern tags given to each line of code inside a method allows us to identify which code portions have a similar set of tags. This can be used for further modularity analyses, such as tangling and cohesion assessments, and for suggesting refactoring opportunities. I.e., code lines sharing a similar concern could be refactored by using techniques, such as e.g. *Extract Method* to separate concern specific code in a separate component, *Code Reordering* to move similar

```
1  public void initBookStore() throws IOException, SQLException {
2    Path logFilePath = FileSystems.getDefault().getPath("store","req.log");
3    File logFile = logFilePath.toFile();
4    reqLog = new BufferedWriter(new FileWriter(logFile));
5    Path dbPath = FileSystems.getDefault().getPath("store","store_db");
6    String absolutePath = dbPath.toAbsolutePath().toString();
7    String dburl = "jdbc:derby:" + absolutePath + ";user=usr;password=psw";
8    conn = DriverManager.getConnection(dburl);
9    int serverPort = 33333;
10   serverSocket = new ServerSocket(serverPort);
11 }
```

Figure 4.1: A method implementing some initialisation operations for a book store server, such as: (i) opening a stream to a logging file (line 2 to 4); (ii) opening a database connection (line 5 to 8); (iii) opening a socket on a specified port waiting for incoming requests (lines 9 and 10). Even in such a short method, it is hard to manually identify which lines belong to a specific concern, thus calling for an automatic concern tagging assistance.

code as near as possible to ease code understanding and endorse code reuse [44], or by resorting to other advanced separation of concern [50] or refactoring techniques.

In Section 5.4 the concern tags provided by DeDuCT will be used to drive a long method decomposition by means of the *Replace Method with Method Object* refactoring technique. Thanks to DeDuCT, *concern cohesive* fragments can be automatically identified, and then given to the method decomposition approach presented in Chapter 5, thus making the overall refactoring process fully automated.

4.2 DeDuCT

Let us consider a simple use case to describe the proposed approach that automatically tags code lines and finds clustering. Figure 4.1 shows method *initBookStore()* implementing initialisation operations for a book store server, such as: (i) open a stream to a text file used for logging (line 2 to 4); (ii) open the connection to a database (line 5 to 8); (iii) open a socket on a specified port to wait for incoming requests (lines 9 and 10).

Since the method is both short and performing only initialisation operations, the developer could be satisfied by naming for it a single concern (such as *initialisation*).

However, a more fine grained analysis, and concern attribution could improve code readability and modularity. Actually, by identifying and separating the three different initialisation subtasks stated above by using different, and well named, methods, code understanding and its future reuse could be improved.

Looking at the code in Figure 4.1, we can see how difficult it could be to manually identify which lines belong to a specific concern without any clue from the developer (like comments or empty lines), even in such a short method. Moreover, manually tagging with concern names each line of code quickly becomes an overwhelming task for large applications. Therefore, our approach aims at automatically performing the said task of concern identification.

4.2.1 Find API usages

In order to automatically *deduct*, i.e. extract, concern information from a method body, firstly we give a more precise and operative definition of concern. This can be seen as some heuristic (or rule) guiding a tagging agent and used for concern identification. We leveraged two concern definitions, providing different and complementary *semantic information*, i.e. behavioural clues, about a line of code.

Definition 7. *Instructions that happen to use the same APIs* explicitly *share the same concern (first concern tagging heuristic).*

To apply this heuristic on a fragment of code, we need to solve two different problems: (i) identifying which instructions use the same APIs, and (ii) deciding which concern the API usage suggests. Let us say, e.g., that we have two instructions using `java.lang.StringBuilder` type, then we can infer that such instructions address a concern closely related to string operations, or more generally, to the *string* concern. Similarly, if we have two instructions that use `java.net.ServerSocket` type, then they are dealing with a networking concern (*net*).

Accordingly, we use the standard Java libraries APIs as an indication of the concerns addressed by a line of code (see [97]). Given the huge number of classes in such APIs, they can be appropriately grouped together to define a (less specific) concern. Of course, such a grouping has to be performed only once, and can even simply be suggested by the packages of the Java APIs. Figure 4.2 shows an

API	Concern
java.nio.file.FileSystems	io
java.nio.file.FileSystem	io
java.nio.file.Path	io
java.io.File	io
java.io.FileWriter	io
java.io.BufferedWriter	io
java.lang.String	string
java.lang.StringBuilder	string
java.net.ServerSocket	net
java.sql.Connection	persistence

Figure 4.2: The *concern mapping* function for the Java types used in the *initBookStore()* method.

excerpt of this *API-to-concern* mapping, that is limited to the types used in the *initBookStore()* method. We can now provide a more formal definition used to assign concern tags to an instruction in line with the APIs it directly uses.

Definition 8. Given a set of APIs (i.e. types) T denoted as *tagging APIs*, a set of concerns C, and a *concern mapping* function $f_c : T \to C$, we say that a program point *explicitly* covers the concern $f_c(t)$ if it directly uses the type t, i.e.: (i) calls a method defined in t; (ii) uses a field defined in t; (iii) calls a method whose return type is t; (iv) uses a field whose type is t.

By using the above definition, we can inspect the *initBookStore()* method and find, for each line of code, which APIs are used. Then a tag can be attributed to the code. Figure 4.3 shows the results of this inspection, associating to each method line the related concern tags, according to the mapping function shown in Figure 4.2.

Line 2 gathers the path to the log file, and uses three types from the APIs, i.e. *FileSystem* (returned by *getDefault()* method of class *FileSystems*), *FileSystems* and *Path*, which are all related to the *io* concern. Therefore, a *0:io(2) concern tag* is added, where: *0* is the distance from the source of this tag (i.e. the current line in this case, since it directly uses *io* types); *io* is the concern; and *2* is the line providing the tag (again, the same line in this case).

Line 3 converts the obtained path to a file reference, thus calling a method of the *Path* class and using *File* as its return type. Both types are related to the *io* concern, so *0:io(3)* tag is added. Lines 4 opens a buffered stream to the log file, and it uses two *io* types (*BufferedWriter* and *FileWriter*), therefore we added the

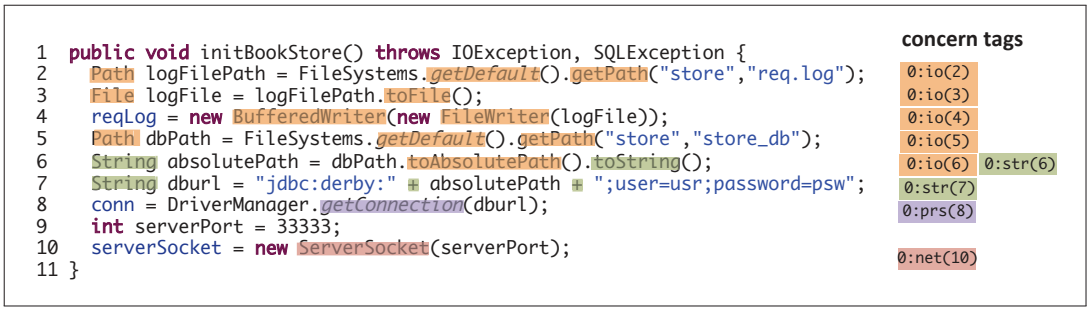

Figure 4.3: The first step of the concern tagging approach. Lines directly using a *tagging API* (i.e. a type defined in the Java Standard Library) have been tagged with the proper concern, according to the devised *concern mapping* function (see Figure 4.2).

0:io(4) concern tag. Line 5, similarly to line 2, gathers the path to the database, thus the *0:io(5)* tag is added. Similarly, we added *0:io(6)* *0:str(6)* to line 6 (where *str* stands for *string*), *0:str(7)* to line 7, *0:prs(8)* to line 8 (where *prs* stands for *persistence*), and *0:net(10)* to line 10.

By considering the API usages we tagged all but line 9, having lines from 2 to 5 with just *io* concern assigned, line 6 with both *io* and *string*, line 7 with *string*, line 8 with *persistence*, and line 10 with *net*.

However, while API usage tells us *which* category of operation is implemented, it can not explain *why*. For example, though line 2 and 6 perform exactly the same operation, i.e. defining a file path, this path will be further used for *different reasons*, i.e. for *io* operations in the first case, and for *persistence* ones in the second case. Therefore, to better understand the reason of an API call, thus the aim of the line performing it (its concern), we need to gather information about how the data produced by that call will be used by the other instructions inside the method, so explaining *why* that particular API was called.

4.2.2 Trace data dependence

To both enhance code coverage and provide more concern information for each line, we trace *data dependence*. The rational for this is suggested by the following heuristic.

Definition 9. *Instructions that are working on the same data* implicitly *share the same concern (second concern tagging heuristic).*

In the previous step (Section 4.2.1) we have found the instructions directly using a Java API, and tagged them with a specific concern according to a provided concern mapping function. Now, we consider each instruction as a *source* of a *data flow* that is related to a specific concern, i.e. the one given by the tag applied to the source instruction. Hence, a propagation algorithm is used to extend such a concern information to the other instructions that are *data dependent* on this *source*. To accomplish this task, we leverage a graph representation of data dependences among instructions, i.e. the *data dependence graph* (see Section 2.1.3 for definitions and details).

Given the data dependence graph and a set of sources, we use *taint analysis* to define the propagation algorithm that solves the concern tagging problem. Taint analysis [72] consists of static [5] and dynamic [87] data flow algorithms that take *sources* for *tainted* data and trace them until they reach one of the *sinks*. While flowing from one instruction to another, these become *tainted*. Typically, it is used to solve *data security* problems [5, 7, 38], and to check data flows within the code [43].

In Section 2.1.7 the concept of *taint analysis* has been introduced. It is a particular class of data flow algorithms that take *sources* for *tainted* data and trace them until they reach one of the *sinks*. While flowing from one instruction to another, these become *tainted*. Typically, it is used to solve *data security* problems (as shown in Chapter 6).

In this case, taint analysis is used to *taint* the dependent program points with a source concern tag, taking as *sources* the APIs referred into code lines, and as *sinks* the instructions that are a certain number of code lines far away from the source (*dependence range*). Moreover, we use the *undirected* data dependence graph. This enables us to consider complex data dependence situations, thus tracing both a *backward* and a *forward* data dependence, tagging the instructions that respectively *prepare* the input and *use* the output of the source instruction.

Figure 4.4 shows how the proposed taint analysis algorithm works, extending the results given by the previous step (Section 4.2.1). Line 6 has been previously tagged by the *find API usage* step with two concerns: *io* and *string*. These two

Figure 4.4: The second step of the concern tagging approach. By using a taint analysis approach, all the concern tags given by the previous step (i.e. directly accessing a Java API, thus having a '0' *depth* value) are now used as *sources* for a concern related data flow that we can trace to discover all the instructions *implicitly* sharing the same concern of the source. All the concern tags having a depth value greater then '0' have been discovered using this taint analysis propagation. For example, let us consider the concern tags generated by *line 6* (highlighted by the use of bold text): the *io* and *str* tags have been backward propagated to *line 5* due to the data dependence made by the `dbPath` variable, and forward propagated to *line 7* and *line 8* in turn due to the `absolutePath` and the `dburl` dependence.

tags are now extended to all data dependent lines within the chosen *dependence range* (here set to 4). This means revealing the instructions that are on a data path providing the input variable `dbPath` and the ones on a data path that uses `absolutePath` variable. For the former, we backward taint line 5 with both of the source concern tags, adding the *1:str(6)*, where *1* means that it is 1 source line distant from the source (depth), and *6* is the previous line on the data path. However, the expected *1:io(6)* is not listed, since line 5 already has an *io* tag with a lower depth value. By considering the minimum depth for a specific concern we associate the depth information to the *importance* of a concern for the source line: the more a source for that concern is near in the data path, the more important the concern will be.

Since from line 5 there are no other backward data dependent instructions, we consider forward propagation from line 6. The `absolutePath` output variable is forward used by line 7, causing the *1:io(6)* tag (and not the expected *1:str(6)* for the same reasons explained in the previous example). We can continue the tainting propagation to line 8 due to the use of the `dburl` variable, causing the *1:str(7)* and the *2:io(7)*. Since there are no other instructions on this data path, we stop the

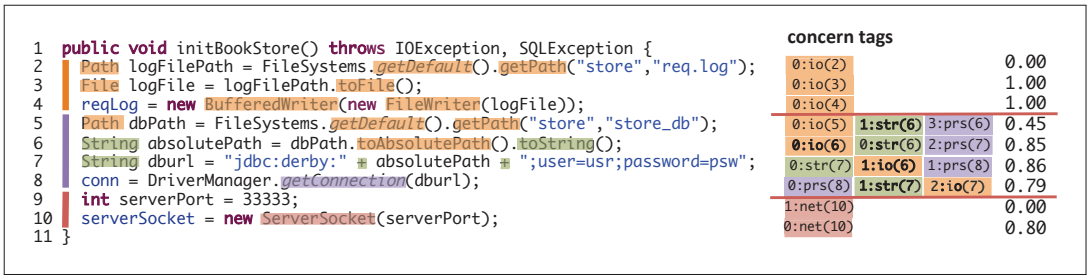

Figure 4.5: Concern tags given by the proposed approach can be used to suggest refactoring opportunities. Using a clustering algorithm, three blocks of source lines having similar concern tags have been found (i.e. three different *clusters*): line 2 to line 4 ($C0$, indicated by the orange vertical line); line 5 to line 8 ($C1$, indicated by the purple vertical line); and line 9 to line 10 ($C2$, indicated by the red vertical line). The identified clusters actually refer to the three different subtask performed by the method: open the log file, open the database connection, and open the server socket, thus automatically suggesting a method decomposition into smaller and concern cohesive methods.

propagation for the two source tags of line 6. The tainting procedure is executed for each source tag on each line, obtaining the final results shown in Figure 4.4.

Thanks to the high level of abstraction given by the concern tags, we can easily understand the responsibility of a source line before actually looking at the code itself. E.g., tags given to line 4 suggest that it is mostly dealing with *io* operations, which is confirmed by the actual Java code. Line 7 performs *string* manipulations using data *directly* given by the *previous* instruction, which performs *io* operations (since the *io* has a depth of 1, and the 6th source line is the previous line), and directly provided to a *persistence* related instruction. Moreover, line 8 performs *persistence* operations, and uses data that directly come from a *string* manipulation. It also depends on an *io* related instruction, having lower influence than the other two tags according to the depth information that comes with the tag. Line 9 does not use any API (since it has no sources), however it provides data further used for *networking* operations, hence it is tagged with *net*.

4.2.3 Clustering

We investigated how the proposed tags can indicate *concern cohesion* for a specific method, and further suggest refactoring opportunities, such as *Extract Methods* [44]. This will be also used in Chapter 5 to automate the decomposition of long methods. By defining a similarity function and a clustering algorithm, we can find clusters of instructions showing a similar *composite concern* (i.e. consisting of a set of concern tags given to source lines). As a similarity function, we used the *Jaccard Index* $J(A, B)$ where A and B are two sets of concern tags:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

To give more relevance to concern tags having a lower level of depth, for a specific set of tags, we consider each tag a number of times determined by the weight function $w(d) = 1 + (d' - d)$, where d indicated the tag depth and d' the considered dependence range (that is set to 4 in our case). E.g. for determining the similarity between line 7 (L_7) and line 8 (L_8) in Figure 4.5 the two sets are:

$$L_7 = \{str_i : i = 0..4\} \cup \{io_i : i = 0..3\} \cup \{prs_i : i = 0..3\}$$
$$L_8 = \{prs_i : i = 0..4\} \cup \{str_i : i = 0..3\} \cup \{io_i : i = 0..2\}$$

Then Jaccard Index is:

$$J(L_7, L_8) = \frac{|L_7 \cap L_8|}{|L_7 \cup L_8|} = \frac{11}{14} = 0.79$$

Given this similarity function, we *linearly* traverse the method code (from the first to the last line) and compute the similarity index of a line with the previous one (that is zero for the first line), using then a *similarity threshold* (that is *0.66* in our case) to determine whether a cluster has been formed.

Accordingly, for the above code (see Figure 4.5), we have found three blocks of source lines having similar concern tags, i.e. three different *clusters*, as: line 2 to line 4 ($C0$); line 5 to line 8 ($C1$); line 9 to line 10 ($C2$). More formally, we have identified three cluster of instructions having an high level of *similarity*, according to their concern tags. We can see how, in this simple example, the identified clusters

Figure 4.6: Apply extract method refactoring to separate clustered concern tags and enhance concern cohesion.

actually refer to the three different subtask performed by the method: open the log file, open the database connection, and open the server socket. The developer can use such cluster indications, automatically added to the source code by our tool, to reason about the extraction of each cluster into its own method.

Figure 4.6 shows the refactored version of the use case. This was obtained by using the Extract Method refactoring technique to each cluster suggested by DeDuCT. Now, the refactored *initBookStore()* method has only three lines of code, referring other well named, lower level methods. Running again the DeDuCT tagging and clustering steps we have tagged and clustered it, showing that the obtained methods have an higher *concern cohesion* since they all but *initBookStore()* consist of one cluster.

4.3 Implementation

This section describes the overall `DeDuCT` framework (depicted in Figure 4.7) used to implement the taint analysis approach for attributing concerns, and the clustering for the concern assessment both described in Section 4.2. The first part of the implemented framework (i.e. until the production of the *concern tags* list) is similar to the one implemented for portability assessment, described in Chapter 3 (see Figure 3.5). In fact, both of them perform a bytecode analysis to identify

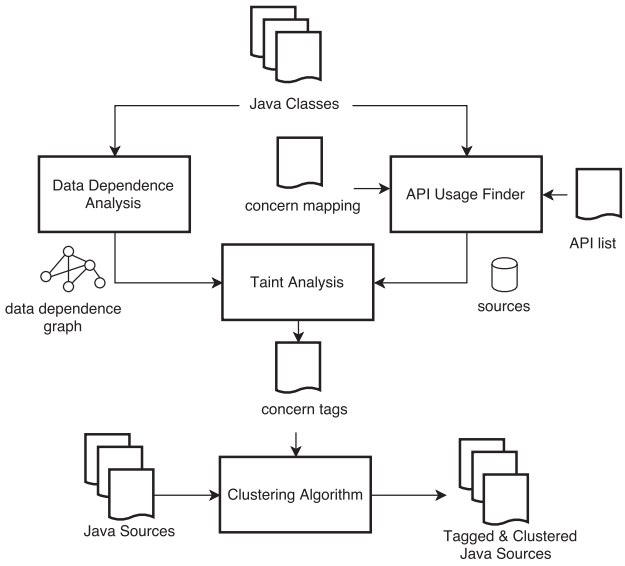

Figure 4.7: The DeDuCT framework.

usages of specific types, binding the code to a specific platform, or supporting specific concerns by means of well known Java APIs. However, they differs in the way the taint analysis has been defined. For example, for the portability assessment, a notion for both *active* and *passive* sinks was required (see Section 3.3.4), while for the concern propagation problem, sinks are instead defined using a *dependence range*. Moreover, differently from the portability assessment, in this case a lot of details are considered during the tainting propagation, e.g. the concern name and the tag depth, which are both needed during the concern similarity assessment.

Starting from Java application classes, the `API Usage Finder` component identifies the program points directly using a type defined in the *API list* (such as the Java APIs or any other external library that the developer thinks useful to characterise a concern, more generally a *tagging API*). This is performed according to Definition 8, thus selecting each `GETFIELD`, `PUTFIELD` and `INVOKE` opcodes referring a class in the *API list*, then determining the related concern according to the *concern mapping* function. The bytecode inspection of the input classes has been performed using the Byte Code Engineering Library (BCEL[1]).

As a second step, for each method of the input application, these sources are

[1]https://commons.apache.org/proper/commons-bcel/

Table 4.1: Analysed software systems

Systems	Methods	Method Body Lines	
		total	per method
JUnit 3.7	536	1876	3.50
JUnit 4.10	1231	2819	2.29
Clever	1645	10082	6.13
Only considering methods having at least 2 lines of code			
JUnit 3.7	251	1602	6.38
JUnit 4.10	464	2084	4.49
Clever	803	9260	11.53

Table 4.2: Analysed software systems

Systems	API Usage		Taint Analysis		Clusters	Cluster size
	tagged lines	per method	tagged lines	per method		
JUnit 3.7	766 (41%)	1.43	1175 (63%)	2.19	211	3.99
JUnit 4.10	1094 (39%)	0.89	1644 (58%)	1.34	296	3.15
Clever	4680 (46%)	2.84	6483 (64%)	3.94	1184	3.86
Only considering methods having at least 2 lines of code						
JUnit 3.7	674 (42%)	2.69	1055 (66%)	4.20	211	3.99
JUnit 4.10	820 (39%)	1.77	1276 (61%)	2.75	296	3.15
Clever	4489 (48%)	5.59	6291 (68%)	7.83	1184	3.86

used by the `Taint Analyser` component to drive the concern tag propagation algorithm described in Section 4.2. To perform such an analysis, a data dependence graph has been achieved by using JavaPDG [89]. Having the sources and the *undirected* data dependence graph, the taint analysis can be performed, giving as a result a list of *concern tags*. Finally, the `Clustering Algorithm` component has been implemented to perform the computation described in Section 4.2.3.

4.4 Experiments

A few open source software systems have been analysed, showing the results of our approach in terms of tagged lines and tags clustering.

Table 4.1 shows the main characteristics of the analysed software systems, i.e. JUnit 3.7, JUnit 4.10 and Clever[2], and the related structural metrics (the first three rows consider all the methods, while the last three only consider the ones having at least two lines of code). The size of the analysed systems is shown by the

[2]https://github.com/clever-unime

number of methods (second column), and the lines of code (third column). The average number of lines per method (fourth column) has been computed to show the granularity of methods, and given that the average size is less than or equal to 6.13 lines for all analysed systems, we can say that the methods tend to be small enough to expect they address a single responsibility.

Table 4.2 shows the results collected by the implemented tool (also here, the first three rows show data collected on all the methods, while the following three rows consider only methods having at least two lines of code). The metrics collected by our tool are as follows. The number of lines that have been automatically tagged with at least one concern, by considering the direct APIs usage of the line, is shown on the second column. For both JUnit versions, only the Java APIs usage was considered. For Clever, which is a complex cloud middleware, other external APIs have been mapped to concerns, e.g. *libvirt*[3] (*virtualisation*) and *smack*[4] (*messaging*). We manage to tag between 39% and 46% lines of code (according to the system under analysis). By using the taint-based approach, as described in Section 4.2.2, we manage to tag all the lines of code that are related by data dependence, and we can tag between 58% and 64% of lines (fourth column). Once lines of code have been tagged, we determine clusters, as described in Section 4.2.3. Only meaningful clusters were considered, i.e. counting at least two lines, the number of clusters found are in the sixth column, while their average size in the seventh column.

Since JUnit 3.7 is well known for its modularity, it appears that to solve a specific problem only a few lines are needed (the average size of its method is 3.50). This consideration seems to be confirmed by the size of the clusters suggested by our tool, which is between 3 and 4 lines.

Table 4.3 (JUnit 3.7), Table 4.4 (JUnit 4.10), and Table 4.5 (Clever) show cluster distribution for each method having at least 2 lines of code. In every table, the first column shows the number of clusters found on each method, hence whether a method has zero, one or more clusters. Zero clusters indicates that no concern tags could be attributed to lines (as shown by the tables, not all the lines have been given tags).

[3]https://libvirt.org
[4]https://www.igniterealtime.org/projects/smack

Table 4.3: Clusters distribution for JUnit 3.7

Clusters per Method	JUnit 3.7			
	Methods		AVG(size)	
			method	cluster
0*	285	–	–	–
0	114	(45%)	3.54	–
1	95	(38%)	5.59	3.89
2	26	(10%)	11.50	3.31
3	8	(3%)	13.00	3.54
4-6	8	(3%)	33.00	5.38
7+	0	(0%)	–	–

* method size < 2 lines of code

Table 4.4: Clusters distribution for JUnit 4.10

Clusters per Method	JUnit 4.10			
	Methods		AVG(size)	
			method	cluster
0*	767	–	–	–
0	214	(46%)	3.45	–
1	215	(46%)	4.45	3.18
2	27	(6%)	10.07	3.02
3	6	(1%)	12.33	2.89
4-6	2	($\simeq$0%)	21.50	3.67
7+	0	(0%)	–	–

* method size < 2 lines of code

Table 4.5: Clusters distribution for Clever

Clusters per Method	Clever			
	Methods		AVG(size)	
			method	cluster
0*	839	–	–	–
0	257	(32%)	5.29	–
1	279	(35%)	7.74	4.46
2	131	(16%)	14.71	4.03
3	51	(6%)	19.92	3.75
4-6	67	(8%)	28.00	3.41
7+	18	(2%)	51.17	3.54

* method size < 2 lines of code

In every table, the first row shows the number of methods having less then 2 lines of code, thus unable to contain a cluster having at least 2 lines. The other lines refer to methods with at least two lines. For the two versions of JUnit, we can see that several of these methods (95 in 251 for JUnit 3.7 and 215 in 464 for JUnit 4.10) have only one cluster, this is a strong indication that such methods perform one well defined task, since all lines are data dependent and with the same set of tags. Moreover, we can see that the average size of the cluster is similar to the average size of the method for both of JUnit versions (70% and 71% respectively) and a little less for Clever (58%). For JUnit 3.7, we find some methods having more than one cluster, i.e. 26 with two clusters (10% of the number of methods with at least two lines), while only the 6% of the methods have more than 2 clusters. Similarly, for JUnit 4.10, we have 27 methods (6% of considered methods) with two clusters, and 8 methods with more clusters (slightly more than 1% of methods). Clever has 279 in 803 methods with one cluster, 131 methods with two clusters and 136 methods with more than three clusters (8% of methods).

Generally, the more clusters per method the more opportunities of refactoring to separate a well defined task. This is also supported by the not too small average size of the clusters.

4.5 Tagging examples

In this section, some of the tagging and clustering results, obtained during the experiments described in Section 4.4, are shown. The taint analysis was performed using a *maximal propagation depth* of 4, while clustering used a *similarity threshold* of 0.66. For each line, tags and similarity scores (with the previous line) were added through comments directly aside the code, while clusters id are shown at the start of the line. In Chapter 5 it will be shown how these clusters of concern cohesive lines can be used to effectively drive a long method decomposition using the *Replace Method with Method Object* refactoring technique.

Figure 4.8 shows the *rerunTest()* method of the *TestRunner* class in JUnit 3.7. It has 25 *lines of code* (LOC). For this method, DeDuCT found 4 clusters: *C78* (5 lines), *C79* (3 lines), *C81* (3 lines), and *C85* (2 lines). *C80*, *C82*, *C83*, and *C84* are ignored, since they are all composed by a single line, thus they are

```
554     private void rerunTest(Test test) {
555 C78     if (!(test instanceof TestCase)) { // sim:0.00 2:string(554) 2:reflection(554)
556 C78         showInfo("Could not reload "+ test.toString()); // sim:0.75 0:string(556)
      2:reflection(554)
557             return;
558         }
559 C78     Test reloadedTest= null; // sim:0.88 1:string(573) 2:reflection(573)
560         try {
561 C78         Class reloadedTestClass= getLoader().reload(test.getClass()); // sim:0.67
      0:reflection(561) 2:string(554)
562 C78         Class[] classArgs= { String.class }; // sim:0.88 0:reflection(562) 3:string(564)
563 C79         Object[] args= new Object[]{((TestCase)test).getName()}; // sim:0.60 0:string(563)
      1:reflection(565)
564 C79         Constructor constructor= reloadedTestClass.getConstructor(classArgs); // sim:0.70
      0:reflection(564) 2:string(565)
565 C79         reloadedTest=(Test)constructor.newInstance(args); // sim:0.89 0:reflection(565)
      1:string(563)
566         } catch(Exception e) { // untagged
567 C80         showInfo("Could not reload "+ test.toString()); // sim:0.40 0:string(567)
568             return;
569         }
570 C81     TestResult result= new TestResult(); // sim:0.25 2:reflection(571) 3:string(571)
571 C81     reloadedTest.run(result); // sim:0.71 1:reflection(565) 2:string(565)
572
573 C81     String message= reloadedTest.toString(); // sim:0.78 0:string(573) 1:reflection(565)
574 C82     if(result.wasSuccessful()) // sim:0.33 3:reflection(570) 4:string(570)
575 C83         showInfo(message+" was successful"); // sim:0.38 0:string(575) 2:reflection(573)
576 C84     else if (result.errorCount() == 1) // sim:0.38 3:reflection(570) 4:string(570)
577 C85         showStatus(message+" had an error"); // sim:0.38 0:string(577) 2:reflection(573)
578         else
579 C85         showStatus(message+" had a failure"); // sim:1.00 0:string(579) 2:reflection(573)
580     }
```

Figure 4.8: JUnit 3.7, class: `junit.swingui.TestRunner`.

```
153     protected String processArguments(String[] args) {
154 C41     String suiteName= null; // sim:0.00 2:string(170)
155 C41     for (int i= 0; i < args.length; i++) { // sim:0.75 1:string(156)
156 C41         if (args[i].equals("-noloading")) { // sim:0.80 0:string(156)
157                 setLoading(false); // untagged
158 C41         } else if (args[i].equals("-nofilterstack")) { // sim:1.00 0:string(158)
159                 fgFilterStack= false; // untagged
160 C41         } else if (args[i].equals("-c")) { // sim:1.00 0:string(160)
161 C42             if (args.length > i+1) // sim:0.60 2:string(155)
162 C43                 suiteName= extractClassName(args[i+1]); // sim:0.60 0:string(162)
163                 else
164 C44                 System.out.println("Missing Test class name"); // sim:0.00 0:io(164)
165 C45             i++; // sim:0.00 1:string(156)
166             } else {
167 C45             suiteName= args[i]; // sim:0.75 2:string(155)
168             }
169         }
170 C45     return suiteName; // sim:0.75 1:string(162)
171     }
```

Figure 4.9: JUnit 4.10, class: `junit.runner.BaseTestRunner`.

not real clusters, but lines having a similarity with the surrounding ones under the considered threshold. As we can see, all the lines from 555 to 565 are almost concerning reflection operations, but they were split in two separate clusters because of the similarity between lines 562 and 563. However, we can see that this is just under the considered threshold (0.60). Therefore, even if in this iteration two separate clusters were suggested, the developer may further decide to merge them in a single one as a result of a more focused, less restrictive clustering iteration (i.e. by relaxing the threshold from 0.66 to 0.60 just over this class or method).

Figure 4.9 shows the *processArguments()* method (17 LOC) of the *BaseTestRunner* class in JUnit 4.10. DeDuCT found 2 clusters: *C41* (7 lines), and *C45* (3 lines) (ignoring one-line clusters). As we can see, lines 157 and 159 were *untagged*, i.e. not suitable for any tag to be added, due to any API usages or data dependence. Although, these lines without tags have not caused cluster *C41* to be split into separate clusters (in fact, it goes from line 154 to line 160). This is because DeDuCT can properly handle these situations, avoiding unnecessary split of clusters due to isolated untagged instructions. The rational here is that: if an instruction, which was not tagged, is surrounded above and below by two instructions with a high similarity, it is possible that also the untagged one also shares the same concern, yet avoiding a set of similar lines to be split because of a single, untagged, one.

This behaviour can be enabled by configuring a *skip* factor (typically set to 1

line), ignoring a certain number of untagged lines and, if the similarity of the two surrounding ones is over the threshold, adding all of them to the same cluster. In fact, the *sim:1.00* value stated on line 158 is referred to its similarity with line 156 (i.e. skipping 157 as a single untagged line), and since it is over the threshold, the cluster was not interrupted, thus including also the skipped line.

Figure 4.10 shows the *sendFileSOC()* method (40 LOC) of the *ImageManager* class in Clever. Being a long method, several clusters have been suggested, some of them having a number of lines that is over the average size of clusters for the considered project (that is 3.86, see Table 4.2), i.e. *C128* (7 lines), *C129* (4 lines), *C131* (5 lines). Other smaller clusters have also been found (*C126*, *C126*, and *C132* with 2 lines of code).

As we can see, even though the fragment from line 685 to line 702 is a linear and uninterrupted sequence of instructions, i.e. without any comments or white spaces helping the developer in identifying lines related to different tasks, DeDuCT was automatically able to find three different sub-tasks: socket initialisation (*C128*), data streaming (*C129*), data receiving and closing operations (*C131*).

Finally, Figure 4.11 shows another method of the *ImageManager* Clever class, *sendFileFTM()* (47 LOC). A cluster having a big size was identified, *C118* (11 lines, including the untagged line 733), other than two smaller ones, i.e. *C116* (3 lines), and *C123* (2 lines).

It is important to note that even if a cluster is suggested, not always it is possible to *extract* it as a *valid fragment*, i.e. without violating *syntax rules*, like splitting a conditional statement (see Section 2.5 for more details about *fragment extraction*). For example, it is not possible to validly extract *C116* (lines 728-730) in a new method without violating the syntax rules, because of the condition of an *if* statement was selected but without covering also its body.

We need to decide whether to extract only a part of the cluster (lines 728 an 729) or to use it as a *concern seed*, i.e. as a starting point for a fragment to be extracted, and by automatically selecting surrounding lines according to the language syntax rules. In this case, if we want to extract the entire cluster, we need also to consider the *if* body, i.e. lines 731-734. In Section 5.4 a strategy for an automatic extension of clusters, as concern seeds, into valid fragments will be presented, showing how the clustering information given by DeDuCT can be used

```
675     private String sendFileSOC(String filePath, String destHost) {
676         try
677         {
678 C126    File f = new File(filePath); // sim:0.00 0:io(678) 1:string(685) 3:net(685)
679 C126    if (!f.exists()) // sim:0.82 0:io(679) 2:string(678) 4:net(678)
680         {
681 C127        logger.error("Can't read VM file for sending"); // sim:0.56 0:io(681)
682             return null; // untagged
683         }
684
685 C128    logger.info("Sending " + f.getName() + " to " + destHost); // sim:0.38 0:io(685)
        0:string(685) 2:net(677)
686 C128    Socket soc = new Socket(destHost, 9999); // sim:0.67 0:net(686) 1:io(692) 2:string(677)
687 C128    logger.info("Connected to " + destHost); // sim:0.67 0:io(687) 0:string(687) 2:net(677)
688 C128    byte[] data = new byte[(int) f.length()]; // sim:0.85 0:io(688) 2:string(678) 2:net(694)
689 C128    BufferedInputStream bis = new BufferedInputStream(new FileInputStream(f)); // sim:0.82
        0:io(689) 2:string(678) 4:net(678)
690 C128    bis.read(data, 0, data.length); // sim:0.80 0:io(690) 3:string(689) 3:net(688)
691 C128    bis.close(); // sim:0.78 0:io(691) 3:string(689)
692 C129    DataOutputStream dos = new DataOutputStream(soc.getOutputStream()); // sim:0.58 0:net(692)
        0:io(692) 3:string(686)
693 C129    dos.writeInt(data.length); // sim:0.92 0:io(693) 1:net(692) 3:string(688)
694 C129    dos.write(data, 0, data.length); // sim:1.00 0:io(694) 1:net(692) 3:string(688)
695 C129    dos.flush(); // sim:0.91 0:io(695) 1:net(692) 4:string(692)
696 C130    this.logger.info("VM file sent succesfully"); // sim:0.50 0:io(696)
697 C131    DataInputStream dis = new DataInputStream(soc.getInputStream()); // sim:0.42 0:net(697)
        0:io(697) 3:string(686)
698 C131    String path = dis.readUTF(); // sim:0.83 0:io(698) 1:net(697) 4:string(697)
699 C131    dis.close(); // sim:1.00 0:io(699) 1:net(697) 4:string(697)
700 C131    dos.close(); // sim:1.00 0:io(700) 1:net(692) 4:string(692)
701 C131    soc.close(); // sim:0.67 0:net(701) 2:io(686) 3:string(686)
702 C132    return path; // sim:0.55 1:io(698) 2:net(698)
703
704 C132    } catch (UnknownHostException ex) { // sim:0.78 1:net(697) 1:io(697) 4:string(697)
705 C133        this.logger.error("Unknown Host Error sending VM file : " + ex.getMessage()); //
        sim:0.26 0:io(705) 0:string(705) 0:exception(705)
706             return null; // untagged
707
708 C134    } catch (FileNotFoundException ex) { // sim:0.27 1:io(705)
709 C135        this.logger.error("File not found Error sending VM file : " + ex.getMessage()); //
        sim:0.40 0:io(709) 0:string(709)
710             return null; // untagged
711
712 C136    } catch (IOException ex) { // sim:0.40 1:io(709)
713 C137        this.logger.error("IO Error sending VM file : " + ex.getMessage()); // sim:0.40
        0:io(713) 0:string(713)
714             return null; // untagged
715         }
716     }
```

Figure 4.10: Clever, class: `org.clever.HostManager.ImageManagerPlugins.ImageManagerClever.ImageManager`.

```
727     private boolean sendFileFTM(String name, String filePath, String destHost) {
728 C116    File f = new File(filePath); // sim:0.00 0:io(728) 1:xmpp(741) 1:string(729)
729 C116    System.out.println("filesize: " + f.length()); // sim:0.86 0:string(729) 0:io(729)
      2:xmpp(728)
730 C116    if (!f.exists()) // sim:0.85 0:io(730) 2:xmpp(728) 2:string(728)
731         {
732 C117        logger.error("Can't read VM file for sending"); // sim:0.45 0:io(732)
733             return false; // untagged
734         }
735
736         try
737         {
738 C118        String nick = destHost.toLowerCase() + "@" + this.conn.getServer().toLowerCase() +
      "/Smack"; // sim:0.29 0:string(738) 1:xmpp(739) 1:io(739)
739 C118        System.out.println("OFT to " + nick + " from " + conn.getXMPP().getUser()); //
      sim:0.87 0:xmpp(739) 0:string(739) 0:io(739)
740 C118        OutgoingFileTransfer oft = ftm.createOutgoingFileTransfer(nick); // sim:0.87
      0:xmpp(740) 1:string(738) 1:io(746)
741 C118        oft.sendFile(f, name); // sim:0.92 0:xmpp(741) 1:io(728) 2:string(728)
742 C118        while (!oft.isDone()) // sim:0.92 0:xmpp(742) 2:string(740) 2:io(740)
743             {
744 C118            if (oft.getStatus().equals(Status.error)) // sim:1.00 0:xmpp(744) 2:string(740)
      2:io(740)
745                 {
746 C118                System.out.println("ERROR!!! " + oft.getError()); // sim:0.73 0:xmpp(746)
      0:string(746) 0:io(746)
747 C118                oft.cancel(); // sim:0.73 0:xmpp(747) 2:string(740) 2:io(740)
748                     return false; // untagged
749                 }
750
751 C118            System.out.println(oft.getStatus()); // sim:0.85 0:xmpp(751) 0:io(751) 2:string(740)
752 C118            System.out.println(oft.getProgress()); // sim:1.00 0:xmpp(752) 0:io(752)
      2:string(740)
753 C119            System.out.println("........"); // sim:0.38 0:io(753)
754 C120            Thread.sleep(1500); // sim:0.00 0:concurrency(754)
755             }
756
757 C121        if (oft.getStatus().equals(Status.complete)) // sim:0.00 0:xmpp(757) 2:string(740)
      2:io(740)
758             {
759 C122            System.out.println("Transfer done"); // sim:0.23 0:io(759)
760             return true; // untagged
761             }
762
763 C123        if (oft.getStatus().equals(Status.error)) // sim:0.23 0:xmpp(763) 2:string(740)
      2:io(740)
764 C123        System.out.println("Transfer failed: " + oft.getError()); // sim:0.73 0:xmpp(764)
      0:string(764) 0:io(764)
765             return false; // untagged
766
767         } catch (XMPPException e) { // untagged
768 C124        System.out.println("Error sending VM image file with the FTM : " + e.getMessage()); //
      sim:0.00 0:xmpp(768) 0:string(768) 0:io(768)
769             return false; // untagged
770
771         } catch (InterruptedException e) { // untagged
772 C125        System.err.println("Error sleeping during OFT : " + e.getMessage()); // sim:0.00
      0:string(772) 0:exception(772) 0:io(772)
773             return false; // untagged
774         }
775     }
```

Figure 4.11: Clever, class: `org.clever.HostManager.ImageManagerPlugins.ImageManagerClever.ImageManager`.

to effectively assist complex refactoring opportunities.

4.6 Conclusions

In this chapter an approach to attribute concerns names to code lines has been presented. Such a characterisation allows us to find out whether all the code lines in a method contribute to a single small task, therefore modularity can be automatically assessed. Taint analysis was used to spread the initial concern seeds, given by the standard Java libraries APIs. The approach is fully automatic and innovative in that it uses the APIs as a source of concern identification. Then, it finds all (and only) the data dependent code lines regardless of the number of lines that can be in between. By analysing three industrial open source systems, an experimental evaluation of the effectiveness of the proposed approach was also given, showing a very high granularity of the given results, great accuracy and effectiveness.

Chapter 5

Suggest Complex Refactorings

Decomposing long methods into smaller pieces is commonly carried out by applying a sequence of *Extract Methods*. However, local variable usages can make some code hard to be extracted. In the literature the *Replace Method with Method Object* was proposed, which consists in moving the whole long method in a new class, then creating a field for each variable, so letting the developer free to apply any extraction without worrying about variable dependences. However, this can lead to classes exhibiting low cohesion, having many fields used by a few of its methods. Moreover, the considered fields may not correctly describe the actual object state, making the class harder to understand.

In this chapter, data dependence analysis has been used to give an automatic approach for replacing a method with a *Method Object*, with the aim of optimising the number of variables to become fields while minimising the *lack of cohesion* of the resulting *Method Object*. An experimental use case will be deeply described, showing how the concern tagger proposed in Chapter 4 can be jointly used with the approach presented in this chapter, thus providing an automatic tool to find, and to effectively apply, complex refactoring opportunities.

5.1 Assist *Replace Method with Method Object*

One of the most important problems that developers have to face to improve the modularity of their code is to decompose a long method into smaller pieces, so that these are easier to move, comprehend, and maintain. Long methods are

troublesome because they often implement different concerns, which can be hard to identify due to the complex logic they come with.

This decomposition is commonly carried out by applying the Extract Method refactoring, which consists in moving a code portion into its own method, giving it a name that better explains its purpose. However, even assuming that the developer has correctly identified the code to be extracted (which is a different problem, discussed in Chapter 4, i.e. identifying meaningful portions of codes performing a specific task), this operation can be burdened by the use of local variables.

To move the selected code to another method, the developer needs to scan the extracted code to find references to any variables that are local in scope to the source method, thus determining which ones need to be passed either as input parameters or returned as output values. The use of local variables previously declared or further used outside the selected fragment is a *data dependence* between the code to be extracted end the rest of the method. Sometimes, this dependence can even make the extraction unfeasible, that is, when more then one output value need to be returned. In this case is up to the developer to decide whether to make one of these variables a field or to devise a different decomposition.

Considering that refactoring a long method usually requires more then one fragment to be extracted, dealing with local variables can be cumbersome for the developer without proper assistance.

In the literature [12, 44], to avoid problems dealing with variables, the *Replace Method with Method Object* was proposed, which consists in moving the whole long method in a new class, then creating a field for every declared variable. In this way, the developer is free to apply extract method any times without worrying about the visibility of every variable, now become fields. Decomposing the long method into many methods in a separate object is also a good decision to ease the extension of the method capabilities, without affecting the rest of the original class.

Although the effectiveness of this technique has been proved, replacing every variable with a field can lead to classes exhibiting low cohesion, having many fields used by a few of its methods.

Moreover, long methods usually convey different responsibilities, thus possibly requiring a large number of variables to accomplish many small tasks. Therefore, considering each of these as a field will produce a class with a large number of fields

which, taken as a whole, may not correctly describe the actual state of the object. On the other hand, there is no need to replace every variable with a field, such as the ones that appear to be used internally only in one of the methods, i.e. being temporary variables for one of the selected decomposition.

In the following sections, we will see how it is possible to replace a method with a *Method Object* while optimising the number of variables to become fields. This will require the analysis of the *data dependencies* among the different decompositions the developer has chosen, automatically suggesting, for each variable, if it has to be considered: (i) as an input parameter for a method to be extracted; (ii) as a return value; (iii) as an internal variable; (iv) as a field for the *Method Object*.

Moreover, according to the chosen decomposition, fields will be carefully selected, identifying which combination (*field set*) minimises the *lack of cohesion* of the resulting *Method Object*.

In Section 5.2 the first part of the assisting approach will be discussed, showing how data dependencies can be used to automatically identify the role each local variable will have, thus indicating both input parameters and possible return values for each of the method to be extracted.

In Section 5.3 the second part of the approach will be discussed: using a combinatorial approach, all the possible field sets are enumerated, ruling out the ones not satisfying proper constraints imposed by the chosen decomposition. These combinations are then ordered by a score based on the LCOM [18, 55] metric. As a final result, the approach will suggest which variables become fields both to let every extraction feasible (thus having only one return value) and maximise the class cohesion.

In Section 5.4 an experimental use case will be shown. In Chapter 4 it has been described how to automatically characterise (*tag*) the code with the concern it addresses, according to used APIs and data dependence. The obtained concern information is used to identify code fragments that can be extracted into smaller methods, focused on a single concern. For a long method, different fragments are commonly suggested, thus endorsing the use of *Replace Method with Method Object* refactoring technique. We will see how the fragments suggested by *concern tags* can be used as *seeds* for a set of methods to be extracted in a real world scenario, thus showing how the joint use of both the proposed approaches can help the developer

in effectively decompose long methods into a cohesive *Method Object*.

5.2 Determine fragment data dependences

To describe how the approach works, let us consider a simple use case. Figure 5.1 shows the code for `processRequest(request)`, which is a method of a *book store* web service used to process incoming user requests. This method, as commonly happens, was initially conceived by the developer to perform all the necessary steps to accomplish a task, but which turned into a complex and long method (which has 43 lines of code). Helped by the comments inside the code, we can see that the task performed by the method can be divided into different steps: (i) parse input request message; (ii) prepare a query for the book store database; (iii) execute the query; (iv) parse query results; (v) send the results to the client using a socket connection.

Let us suppose that the developer wants to split this long method into different fragments, possibly one for each of the listed steps, providing a *method decomposition* reducing method length and complexity (such as reducing the number of loops or other control statements that make the code harder to read).

5.2.1 Fragments selection

As described in Section 2.5, one of the main approaches to decompose a method is by identifying *fragments* to be later extracted using the *Extract Method* refactoring technique. A code fragment is a contiguous sequence of statements that can be extracted without breaking *syntax* constraints, i.e. being a *valid extraction*.

To determine if a fragment can be a valid extraction, we need a hierarchical model representing the *block structure* of the method source code. In this model, a block is a sequence of contiguous statements, i.e., statements that follow a linear control flow. Statements within a block are divided into two categories: *basic statements* and *complex control structures*. Complex control structures include: *for* statements, *if-then-else*, *while*, *try-catch*, *do* and *switch* statements [118]. Therefore, to be a valid extraction, a fragment must respect these syntax constraints: (i) selected statements are part of a single block; (ii) inside a block, only contiguous

```
1  Connection conn;
2  PrintWriter sockWriter;
3  private void processRequest(String request) throws
       SQLException {
4   // Parse request message: command,text
5   String[] tokens = request.split(",");
6   String command = tokens[0];
7   String text = tokens[1];
8   // Prepare query according to command
9   String searchField = null;
10  if (command.compareTo("AU") == 0) {
11   searchField = "AUTHORS";
12  } else if (command.compareTo("TT") == 0) {
13   searchField = "TITLE";
14  } else {
15   sockWriter.println("ER,request syntax error");
16  }
17  // Execute query
18  if (searchField != null) {
19   String query = "SELECT * FROM STORE WHERE "
20     + searchField + " LIKE '%" + text + "%'";
21   Statement stmt = conn.createStatement();
22   ResultSet row = stmt.executeQuery(query);
23   List<String> resultList = new ArrayList<String>();
24   // Parse query results
25   while (row.next()) {
26    String id = row.getString(1);
27    String authors = row.getString(2);
28    String title = row.getString(3);
29    int quantity = Integer.parseInt(row.getString(4));
30    if (quantity > 0) {
31     String result = id + ":" + authors + ":" + title;
32     resultList.add(result);
33    }
34   }
35   // Send results to the client
36   String response = getDefaultHeader(command);
37   response += getEmptyResponseSet();
38   if (resultList.size() > 0) {
39    response = getHeader(command, resultList.size());
40    for (String result : resultList)
41     response += result + ";";
42   }
43   sockWriter.println(response);
44   stmt.close();
45  }
46 }
```

Figure 5.1: The long method that needs to be replaced with a *Method Object*. The code blocks selected for extraction are also depicted.

statements are selected; (iii) if a statement is selected, then all statements in its children blocks of statements are automatically selected [90].

For example, in Figure 5.1 the code block that goes from line 8 to line 16 (hereafter simply block 8-16) does not break syntax rules, since it is composed of a sequence of (two) contiguous statements, i.e. a *basic statement* (line 9) and a complete *complex control structure* (lines 10-16). On the contrary, fragment 9-13 cannot be extracted in a valid way, since the *if-then-else* complex control structure, started at line 10, has been split. Therefore, to respect syntax rules, if a statement is selected, then all of its *children* statements must be selected.

Once determined if a code fragment *can be* extracted, another problem is to decide if it *should be* extracted to improve code quality. As a rule of thumb, if a code portion needs a comment to understand its purpose, then it should be extracted into a new well-named method [44]. However, to automatically select code to be extracted different approaches have been proposed in the literature, such as selecting the complete computation of a given variable [1, 100, 101], selecting code loosely coupled with the rest of the method, in terms of used variables and types [90, 118], or selecting statements related to the same or similar concern (as shown in Chapter 4).

All the above techniques provide suggestions to let the developer select fragments, each performing a single task. The set of fragments considered for extraction is called *method decomposition*.

As shown in Figure 5.1, the chosen method decomposition is so defined:

1. Fragment 4-7: the code parsing the input request;
2. Fragment 8-16: the conditional statement selecting the `searchField` value according to the `command` variable;
3. Fragment 19-23: the code executing prepared query;
4. Fragment 26-33: the body of the *while* statement, parsing each query result;
5. Fragment 39-41: the body of the *if* statement, preparing the `response` value in case of a non-empty result set.

To apply this method decomposition, instead of using five different Extract Method refactorings, thus adding the new method inside the original class, *Replace*

Method with Method Object refactoring technique is used. This decision may be motivated by different factors: (i) it makes every extract method straightforward, since every variable is replaced by a field inside the *Method Object*, thus avoiding any decision about input parameters or return value to be used; (ii) every added method and field does not mingle with the ones already present inside the original class, thus avoiding to make the class responsibility harder to understand; (iii) the developer may consider this decomposition as a first step for a further extension, that will be easily applied in the context of the new *Method Object*.

As described in [44] (here copied for completeness), to apply a *Replace Method with Method Object* the following steps need to be performed:

- create a new class (the *Method Object*);
- give the new class a *final* field as a reference to the *source object* (the ones that hosted the original method);
- give it also a field for *every* temporary variable and parameter of the original method;
- give the new class a constructor that takes the source object and each parameter;
- give it a method called *compute* without parameters and copy the body of the original method inside of it;
- use the source object field to refer any method and field of the original object;
- replace the old method with the one that creates the new object and calls compute.

Following these steps, the resulting *Method Object* has been created and each fragment of the devised decomposition has been extracted, without worrying about parameters to be passed or values to be returned, because variables have become fields. Figure 5.2 shows the final result for the *compute* method, which shows how every fragment has been replaced by a method extraction. The original method body has been replaced by the following statement: `new RequestProcessor(this, request).compute()`.

```
1  public class RequestProcessor {
2   BookServer bs;
3   // Connection conn;        --> bs.conn
4   // PrintWriter sockWriter; --> bs.sockWriter
5
6   // All variables as fields
7   String request;
8   String [] tokens;
9   String command;
10  String text;
11  String searchField;
12  String query;
13  Statement stmt;
14  ResultSet row;
15  List<String> resultList;
16  String id;
17  String authors;
18  String title;
19  int quantity;
20  String result;
21  String response;
22
23  public RequestProcessor(BookServer source, String request) {
24   this.bs = source;
25   this.request = request;
26  }
27
28  public void compute() throws SQLException {
29   parseRequest();
30   prepareQuery();
31   // Execute query
32   if (searchField != null) {
33    executeQuery();
34    // Parse query results
35    while (row.next()) {
36     parseResults();
37    }
38    // Send results to the client
39    response = getDefaultHeader(command);
40    response += getEmptyResponseSet();
41    if (resultList.size() > 0) {
42     getResponse();
43    }
44    bs.sockWriter.println(response);
45    stmt.close();
46   }
47  }
48
49  [...]
50 }
```

Figure 5.2: The compute method resulting by applying the *Method Object* refactoring, making all non-temporary variables a field (15 added fields).

5.2.2 Determine *input*, *output* and *hidden* variable sets

All the 15 variables of the original method have become fields into the *Method Object*. However, most of these fields are not required to support the data flow between the extracted methods and *compute*. For example, looking at the code on Figure 5.1 in fragment 4-7 the `tokens` variable is declared and then used to parse user request, but it is never used outside of this fragment. It would be better to let it be a temporary variable internal to the respective extracted method, instead of a field that would be accessed only by that method. Similarly, a `result` variable is declared twice but with different scopes, and both are never used or referred outside the respective fragment, i.e. fragment 26-33 and 39-41.

Similar considerations about internal, or *hidden*, variables can be made to determine *input* and *output* variables for each fragment. However, we cannot simply look for accessed and assigned variables respectively, because this will lead to false positives and unnecessary variable sharing, due to the fact that an assigned variable could be not used any more outside the fragment, or reassigned in another fragment. For example, in Figure 5.1 the `response` variable is defined on line 36 and assigned on line 37. Looking at fragment 39-41 it is accessed on line 41, however, there is no need to consider it as an input variable, since the value read on line 41 is the one assigned inside the same fragment on line 39, thus it does not depend on the external assignment made on line 37. In a similar way, in fragment 19-23 the variable `query` was defined (line 19) but never used outside the fragment, thus there is no need for it to be considered as an output variable. However, manually checking where a variable is referred outside a fragment and if there is an actual data dependence can be cumbersome and time consuming for the developer, thus error prone.

To automate this process, we used *data dependence analysis*. Figure 5.3 shows the *data dependence graph* for `processRequest()` method, obtained using the Soot framework[1] [105]. Nodes, which represent the numbered method lines, are grouped according to fragments. Every line that do not belong to any fragment, i.e. not selected for any extraction, e.g. line 3 and 18, will be considered as belonging to the same *external* fragment, yet it will not be considered for an extraction. Edges represent a data dependence between two lines by means of a specific variable,

[1] https://sable.github.io/soot/

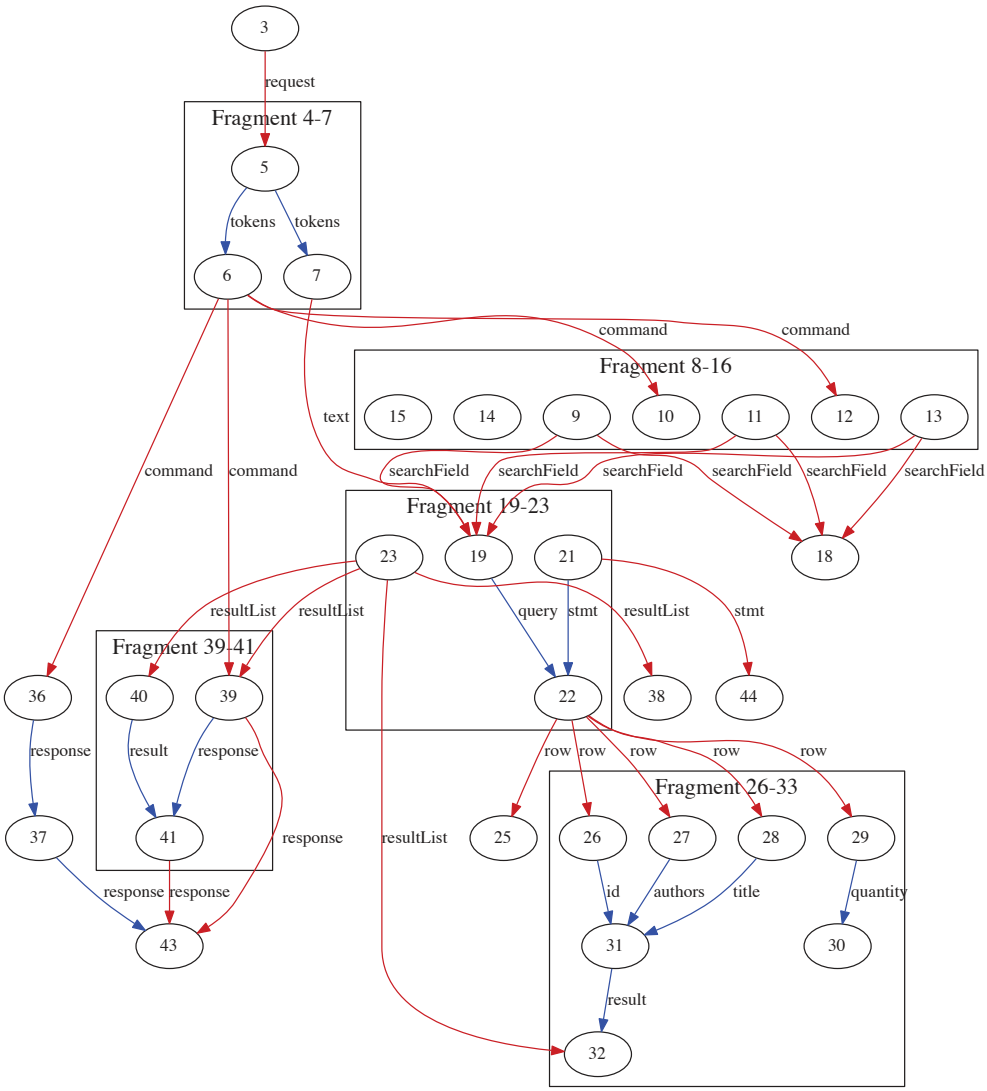

Figure 5.3: The data dependence graph of `processRequest()` (see Figure 5.1). Nodes represent method lines, and are grouped into *fragments*. Each edge represents a data dependence between two lines caused by the variable indicated by its label. Red edges refer to *external variable dependences*, i.e. dependences between nodes of different fragments. Blue edges refer to *internal variable dependences*, i.e. dependences between nodes of the same fragment.

given as edge label.

Definition 10. Line v is data dependent on line u by means of variable x if u accesses the value of x previously assigned by v. This is possible if (i) there exists an execution path p in the control flow graph from u to v, and if (ii) no other statements on p overwrite the value defined by u.

Some of these edges connect nodes belonging to different fragments (depicted in red). Other ones connect nodes belonging to the same fragment (depicted in blue). The former show variable dependences that need to be preserved after the extraction, i.e. using input parameters, return values or shared object fields. The latter indicate dependences that can be hidden inside the new method once the fragment has been extracted. In fact, not every variable used inside a fragment needs to be considered as input parameter, even if defined and assigned outside the fragment.

Definition 11. Given a set of fragments, we will say that an edge (u, v) refers to an *external variable dependence* if u and v do not belong to the same fragment.

Definition 12. Given a set of fragments, we will say that an edge (u, v) refers to an *internal variable dependence* if u and v belong to the same fragment.

Definition 13. Given a fragment F and its induced subgraph G_F in the dependence graph, the *input* variable set I_F is composed of every variable associated with external variable dependence edges coming into G_F.

Definition 14. Given a fragment F and its induced subgraph G_F in the dependence graph, the *output* variable set O_F is composed of every variable associated with external variable dependence edges going out from G_F.

Definition 15. Given a fragment F, the *hidden* variable set H_F is composed of every variable x associated with internal variable dependence edges as long as $x \notin I_F \cup O_F$

For example, in fragment 19-23, `query` is a hidden variable, since only on the (19,22) internal edge, while `stmt` is instead an output variable because both on the (21,22) internal edge and the outgoing (21,44) external edge. Variable `text`

```
F1 (4-7):         I={request} O={command, text} H={tokens}
F2 (8-16):        I={command} O={searchField} H={}
F3 (19-23):       I={searchField, text} O={stmt, row, resultList} H={query}
F4 (26-33):       I={row, resultList} O={} H={id, authors, title, quantity, result}
F5 (39-41):       I={resultList, command} O={response} H={result}
Original method: H={}
```

Figure 5.4: The I (input), O (output), and H (hidden) sets of variables found by using a data dependence analysis.

is an output variable for fragment 4-7 and an input variable for fragment 19-23 because of the incoming (7-19) external edge. Variable `searchField` is an input variable for fragment 19-23 because of (9,19), (11,19), and (13,19). Variables `row` and `resultList` are both output variables because of the external edges going out respectively from node 22 and 23.

Following these definitions, we have analysed the data dependence graph and obtained the input, output, and hidden set for every fragment. Results are shown in Figure 5.4. We can see that 7 different hidden variables (8, if we consider `result` twice) are not supporting data sharing, i.e. {`tokens`, `query`, `id`, `authors`, `title`, `quantity`, `result`}. All variables that are not used by any fragment are considered as *hidden* inside the original method. In this use case, every variable is used by at least one extracted fragment.

In conclusion, by applying the *Replace Method with Method Object* as described in the refactoring catalogue, we would have to consider all the 15 variables of the original method to become fields of the *Method Object*. However, by using the described data dependence analysis, we found that 7 of these fields refer to *hidden* variables, thus they will be used only by one of the methods. Moreover, these fields are not useful to ease method extraction, since they are not involved in any variable dependence. For these reasons, we can remove all the 7 hidden variables from the object fields without being worried of changing the program behaviour, so reducing the number of *Method Object* fields from 15 to 8.

5.2.3 Extend data dependence analysis

Using data dependence we were able to automatically identify input and output variables. However, sometimes this is not enough. Let us consider the example

```
1 String v = computeVar();
2 //-- Fragment to be extracted --//
3 if (cond()) {
4     v = newValue();
5 } else {
6     doSomethingElse();
7 }
8 //------------------------------//
9 use(v)
```

```
1 /* Fragment extraction */
2 int extracted(int v) {
3     if (cond()) {
4         v = newValue();
5     } else {
6         doSomethingElse();
7     }
8     return v;
9 }
```

Figure 5.5: Even if v is never accessed by the fragment to be extracted (see code on the left), it has to be considered as an input variable, due to the `return v` statement that will be eventually added after the fragment extraction (see code on the right).

shown in Figure 5.5 (left side). We want to extract fragment 3-7 from a method and determine input and output variables to define input parameters and return value. Only v is referred with an assignment, thus the only dependence that we can see into the data dependence graph is between line 4 and 9. Using the approach previously described, this fragment will have only v as an output variable and no input variables, thus suggesting that the extracted method will have no parameters and v as a return value. However, if the *else* branch is executed, the value of v will not be overwritten, thus requiring the previous value to be returned, and should be given as an input parameter. Figure 5.5 (right side) also shows the code of the validly extracted method.

This problem is caused by an *implicit* dependence that became *explicit* only after the method extraction, i.e. caused by the *return v* statement that will finally use the variable while the original fragment does not. To make our approach able to correctly behave in these situations we need to extend the static analysis with a further step.

Definition 16. Given a fragment F and its input variable set I_F and output variable set O_F, let s be the first instruction of the fragment and e a symbolic instruction added at the end of the fragment: for each variable $x \in O_F \setminus I_F$, if there exists a path p from s to e where x is never assigned (written), then x should be added to I_F.

In the example shown in Figure 5.5 (left side), the execution path causing the variable v to be added to the input set is in fact the sequence: *line 3*, *line 6*, *end of*

fragment e. This analysis will be executed after the data dependence analysis to possibly add some output variables to the input set in such situations.

5.3 Field Set selection

While hidden variable sets tell which variables are not shared among different fragments, thus not requiring any field to be added, input and output sets tell which input parameters or return variables will be needed in case of a method extraction, i.e. without considering the support given by the fields added for the *Method Object* refactoring.

Since we want to reduce the number of fields to be added to the *Method Object*, at a first attempt the developer may decide to limit fields usage as much as possible, thus using every input variable as a parameter, and use fields only when more than one output variable has been found for a fragment. Choosing which one may depend on different factors, such as the number of fragments that require it, either as input or output variable.

In fragment 4-7, both `text` and `command` are output variables. Typically programming languages do not allow multiple return values, thus forcing the developer to set at least one of them as a field.

Instead of avoiding any non mandatory field, in our approach we look for the field combination that maximise the class cohesion. This will require both a tool to enumerate all possible combination of field *candidates* respecting some enumeration *rules*, and a cohesion metric to evaluate the cohesion value provided for the class by each candidate.

5.3.1 Cohesion for a *Method Object*

Given a method decomposition in a set of fragments $D = \{F_i\}(i = 1, .., m)$, then the set of field candidates C is so defined.

$$C = \bigcup_{F_i \in D} I_{F_i} \cup O_{F_i} \tag{5.1}$$

To measure the class cohesion provided by a given a candidate combination

$K = \{C_1, ..., C_n\}$, we use the LCOM metric proposed by Henderson-Sellers [55]. Consider the decomposition $D = \{F_i\}(i = 1, ..., m)$, and the combination $K = \{C_j\}(j = 1, ..., n)$ of field candidates. Let $\mu(C_j)$ the number of fragments using candidate C_j (either as an input or an output variable). LCOM for a given field combination K is so defined.

$$LCOM(C_1, ..., C_n) = \frac{\frac{1}{n}\sum_{j=1}^{a}\mu(C_j) - m}{1 - m} \tag{5.2}$$

Figure 5.4 shows the input and output sets of all the 5 fragments (in order $F_1, ..., F_5$). By the union of all of those sets we obtain the following candidate set.

$$C = \{request, command, text, searchField, stmt, row, resultList\} \tag{5.3}$$

Let us suppose we want to evaluate how good for the final cohesion of the *Method Object* will be choosing the candidates combination $K = \{text, stmt\}$.

$$\begin{gathered} D = \{F_1, ..., F_5\}; K = \{text, stmt\}; \\ m = 5; n = 2 \\ \mu(text) = |\{F_1, F_3\}| = 2 \\ \mu(stmt) = |\{F_3\}| = 1 \\ LCOM(text, stmt) = \frac{\frac{1}{2}\left[(\mu(text) - 5) + (\mu(stmt) - 5)\right]}{1 - 5} = 0.87 \end{gathered}$$

5.3.2 Constrained enumeration of field sets

Not every field combination is *behaviour preserving*. In fact, since F_1 and F_3 have respectively two and three output values, only one for each fragment can be a return value, forcing the others to be a part of each combination. Therefore, $K = text, stmt$ is not a legal combination and needs to be ruled out. We need a tool for a *constrained enumeration* of all the *legal candidate combinations*.

In previous works [21, 22, 23] we used ATGT [46], that is a tool for automatically

generating test sequences from an Abstract State Machine (ASM) specification (using the AsmetaL[2] language). Based on the Yices SMT (Satisfiability Modulo Theories) solver[3], it was conceived to support t-wise constrained combinatorial interaction testing [24]. The ASM model is used both to describe enumeration domains and constrains to rule out illegal combinations. It finally uses a combinatorial approach to optimise the dimension of the resulting test suite, according to the chosen degree of interaction [24].

In the proposed approach, the enumeration domain is composed of the set of candidates C, while the *enumeration constraints* (or *rules*) are the followings:

1. if a fragment has more output variables, only one of them can be a variable and the other ones must be fields;

2. if a fragment has a *loop index* as an input variable, it cannot become a field.

While the first is a mandatory rule, since it is needed to preserve the behaviour of the code, the second one may be considered as optional, but it is useful to improve code readability and object state description. Looking again at the code of fragment 26-33 (see Figure 5.1) we can see that the body of a *while* statement has been selected for the extraction. The `row` variable is the index of this loop, and is necessary as an input for the selected fragment. Letting it be a field will make the code of the *compute()* method harder to read, because it will not make it explicit that the *variability* of the extracted fragment depends on the loop index. Nevertheless, the most important reason is that a loop index is a temporary variable that is only used to assist the loop statement, thus typically it is not useful to describe the object state because of its variability.

By automatically finding (i) which fragments have more than one output variable, and (ii) which fragments use an input variable as a loop index, we have automated the generation of the ASM, which will be used for the constrained enumeration of valid field combination, i.e. respecting the rules described above. In this way, the developer only needs to provide a set of fragments as a method decomposition, and the approach will automatically provide the field set maximising the *Method Object*

[2]http://asmeta.sourceforge.net/index.html
[3]http://yices.csl.sri.com

```
1  asm processRequest
2  import StandardLibrary
3  signature:
4
5  enum domain Candidate = {FLD|VAR}
6
7  dynamic monitored request :      Candidate
8  dynamic monitored command :      Candidate
9  dynamic monitored text :         Candidate
10 dynamic monitored searchField :  Candidate
11 dynamic monitored stmt :         Candidate
12 dynamic monitored row :          Candidate
13 dynamic monitored resultList :   Candidate
14 dynamic monitored response :     Candidate
15
16 definitions:
17 // F1
18 invariant inv_multiout_b1 over command,text : command=FLD or text=FLD
19
20 // F3
21 invariant inv_multiout_b3_stmt over stmt,row,resultList :
22          stmt=VAR implies row=FLD and resultList=FLD
23 invariant inv_multiout_b3_row over stmt,row,resultList :
24          row=VAR implies stmt=FLD and resultList=FLD
25 invariant inv_multiout_b3_resultList over stmt,row,resultList :
26          resultList=VAR implies stmt=FLD and row=FLD
27
28 // F4
29 invariant inv_loopindex_b4 over row : row=VAR
30
31 default init s1:
```

Figure 5.6: The ASM for the constrained enumeration of field sets. It was automatically generated by the analysis of the input and output sets of the given fragments.

cohesion. Moreover, by using any automatic fragment selection approach, such as the one proposed in Chapter 4, the refactoring process can be fully automated.

Figure 5.6 shows the ASM the obtained ASM. The advantages of using this representation is that new rules can be easily added for future extensions of the approach. Line 5 shows the enumeration domain for the *Candidate* elements (field of variable). Line 7 to 14 list the considered candidates, i.e. all the non hidden variables. All the other lines have been automatically generated to define the enumeration constraints (using a set of *invariants*) for all of the fragments. Line 18 applies *rule 1* for F_1: having two output variables (`command,text`), at least one of those must be a field. The same rule is applied to F_3 that having more then two output variables (`stmt,row,resultList`) we need an invariant for each of the three candidates: if a candidate is set to *VAR* (variable) the others must be set

```
INFO - violated: 24 covered: 0 infeasible: 232
INFO - 24 test results are generated
INFO - reducing test suite
INFO - test suite has not been reduced
1 : row=VAR request=VAR response=VAR text=VAR resultList=FLD command=FLD searchField=FLD stmt=FLD
2 : row=VAR request=FLD response=VAR text=FLD resultList=FLD command=FLD searchField=FLD stmt=FLD
3 : row=VAR request=VAR response=VAR text=FLD resultList=FLD command=FLD searchField=FLD stmt=FLD
4 : row=VAR request=FLD response=FLD text=VAR resultList=FLD command=FLD searchField=VAR stmt=FLD
5 : row=VAR request=VAR response=FLD text=VAR resultList=FLD command=FLD searchField=FLD stmt=FLD
6 : row=VAR request=FLD response=FLD text=FLD resultList=FLD command=VAR searchField=VAR stmt=FLD
7 : row=VAR request=FLD response=VAR text=FLD resultList=FLD command=FLD searchField=VAR stmt=FLD
8 : row=VAR request=FLD response=VAR text=FLD resultList=FLD command=VAR searchField=FLD stmt=FLD
9 : row=VAR request=FLD response=VAR text=FLD resultList=FLD command=VAR searchField=VAR stmt=FLD
10 : row=VAR request=FLD response=VAR text=VAR resultList=FLD command=FLD searchField=VAR stmt=FLD
11 : row=VAR request=FLD response=FLD text=VAR resultList=FLD command=FLD searchField=FLD stmt=FLD
12 : row=VAR request=VAR response=VAR text=FLD resultList=FLD command=FLD searchField=VAR stmt=FLD
13 : row=VAR request=VAR response=VAR text=FLD resultList=FLD command=VAR searchField=FLD stmt=FLD
14 : row=VAR request=VAR response=FLD text=FLD resultList=FLD command=VAR searchField=FLD stmt=FLD
15 : row=VAR request=FLD response=FLD text=FLD resultList=FLD command=VAR searchField=FLD stmt=FLD
16 : row=VAR request=VAR response=VAR text=VAR resultList=FLD command=FLD searchField=VAR stmt=FLD
17 : row=VAR request=FLD response=FLD text=FLD resultList=FLD command=FLD searchField=VAR stmt=FLD
18 : row=VAR request=FLD response=FLD text=FLD resultList=FLD command=FLD searchField=FLD stmt=FLD
19 : row=VAR request=FLD response=VAR text=VAR resultList=FLD command=FLD searchField=FLD stmt=FLD
20 : row=VAR request=VAR response=FLD text=FLD resultList=FLD command=FLD searchField=VAR stmt=FLD
21 : row=VAR request=VAR response=FLD text=VAR resultList=FLD command=FLD searchField=VAR stmt=FLD
22 : row=VAR request=VAR response=VAR text=FLD resultList=FLD command=VAR searchField=VAR stmt=FLD
23 : row=VAR request=VAR response=FLD text=FLD resultList=FLD command=VAR searchField=VAR stmt=FLD
24 : row=VAR request=VAR response=FLD text=FLD resultList=FLD command=FLD searchField=FLD stmt=FLD
```

Figure 5.7: Legal variable and fields combinations given by ATGT.

to *FLD* (field). In F_4 `row` is used as a loop index, thus requiring the *rule 2* to be applied, that is, `row` must be set to *VAR*[4].

The enumerated combinations over the 8 candidates that respected the described constraints are shown in Figure 5.7. Among all the 256 possible combinations (2^8) only 24 respected the devised constrains, thus 232 of them have been ruled out. Since we are not interested in the combinatorial optimisation provided by ATGT, we have set a 8-wise coverage, thus as the number of the considered candidates. For example, the first combination selected {`request,text,row`} as variables, and {`resultList,stmt,searchField,command`} as fields. As we can see, all of these combinations have either `command` or `text` as a field, due to the behaviour constraint on F_1. The same consideration holds for the constraint on F_3. Moreover, every combination has `row` as a variable due to the *loop index* constraint on F_4.

[4]For more details about how to define an ASM using the AsmetaL language refer to: http://fmse.di.unimi.it/asmeta/download/AsmetaL_guide.pdf

Table 5.1: Legal field sets ordered by score=LCOM(fs)

Field Set	score
command,resultList,stmt	0.67
command,resultList,stmt,text	0.69
command,resultList,searchField,stmt	0.69
command,resultList,searchField,stmt,text	0.70
resultList,stmt,text	0.75
command,resultList,searchField,stmt,text,request	0.75
command,resultList,stmt,text,request	0.75
command,resultList,stmt,request	0.75
command,resultList,searchField,stmt,request	0.75
command,resultList,searchField,stmt,response	0.75
command,resultList,searchField,stmt,response,text	0.75
command,resultList,stmt,response	0.75
resultList,searchField,stmt,text	0.75
command,resultList,stmt,response,text	0.75
command,resultList,searchField,stmt,response,text,request	0.79
command,resultList,stmt,response,text,request	0.79
command,resultList,searchField,stmt,response,request	0.79
command,resultList,stmt,response,request	0.80
resultList,searchField,stmt,text,request	0.80
resultList,searchField,stmt,response,text	0.80
resultList,stmt,text,request	0.81
resultList,stmt,response,text	0.81
resultList,searchField,stmt,response,text,request	0.83
resultList,stmt,response,text,request	0.85

5.3.3 Select best field set

For each of the legal combinations listed in Figure 5.7 we can use Equation 5.2 to determine the cohesion of the *Method Object*. Note that we limit the evaluation of this metric to the method obtained by the extraction of the selected fragments, thus not considering both the *compute()* method and the class constructor. The LCOM values obtained for each fields set are shown in ascending order in Table 5.1.

We can see that the field set providing the lowest lack of cohesion is {`command, resultList, stmt`} with a LCOM value of 0.67. Therefore, this is the preferred combination to be used inside the *Method Object*. Once fields have been selected and added to the class, the fragment extraction from the *compute()* method can be easily performed using the assistance of the *Extract Method* refactoring tools commonly provided by advanced IDEs. This operation will be performed without any error of multiple output values, since thanks to our approach we have already added necessary fields to avoid this problem. Figure 5.8 shows the final version

Table 5.2: Comparison of the number of fields for the resulting class using different approaches

Field selection approach	number of fields
All variables	15
Only non-hidden	8
Minimising LCOM	3

of the class after fragment extractions, using `command`, `resultList`, and `stmt` as class fields.

Maximising the cohesion is not the only advantage we obtained. By preserving the use of the right variables in the *compute()* method, i.e. the original method after the fragment extraction, we better understand the data flow between the different fragments, by the use of parameters and return values.

Another important difference is that now *compute()* may preserve some of the parameters of the original method. In this way, the constructor will take only the method parameters that were selected for describing the *object state*, i.e. the selected fields. The other ones, selected as variables, will be given to the *compute()*, like the `request` parameter in this use case.

In conclusion, using this approach we were able to automatically select only useful fields. As summarised in Table 5.2, the traditional refactoring approach tells us to use all variables as fields to ease method decomposition (15 fields). However, once decided the fragments to be extracted, we can use data dependence analysis to determine which variables are actually hidden inside the extracted method, and reducing the number of required fields (8 fields). Moreover, not every field set provides a high level of object cohesion. By using a constrained enumeration and a cohesion metric we can determine the field set that provides the lowest lack of cohesion, thus reducing again the number of required fields (3 fields).

5.4 Experiments

To assess the effectiveness of the described approach, let us consider a real world use case, by refactoring a long method taken from the Clever cloud middleware used for experiments in Section 4.4; `ImageManager.storageIM()` (the complete method listing is shown in Appendix A.1).

```
1  public class RequestProcessor {
2   BookServer bs;
3   // Connection conn;        --> bs.conn
4   // PrintWriter sockWriter; --> bs.sockWriter
5
6   // Fields selected by the tool
7   String command;
8   Statement stmt;
9   List<String> resultList;
10
11  public RequestProcessor(BookServer source) {
12   this.bs = source;
13  }
14
15  private void compute(String request) throws SQLException {
16   String text = parseQuery(request);
17   String searchField = prepareQuery();
18   // Execute query
19   if (searchField != null) {
20    ResultSet row = executeQuery(searchField, text);
21    // Parse query results
22    while (row.next()) {
23     parseResults(row);
24    }
25    // Send results to the client
26    String response = getDefaultHeader(command);
27    response += getEmptyResponseSet();
28    if (resultList.size() > 0) {
29     response = getResponse();
30    }
31    bs.sockWriter.println(response);
32    stmt.close();
33   }
34  }
35
36  [...]
37 }
```

Figure 5.8: The compute method of the resulting *Method Object* making as field only the variables selected by our approach (3 added fields).

The `ImageManager` class has 853 LOC (*lines of code*). It has 18 fields and 36 methods. The `sorageIM()` method has 152 LOC and 134 NCNB (*non comment non blank*. It has 16 local variables (17 if we consider separately the different scoped `mgr` variables), 2 input parameters and no return value. It also accesses to 3 of the 18 class fields.

5.4.1 Fragment extractions guided by concern tags clusters

To extract fragments with a high *concern cohesion* we use the concern tagging approach proposed in Chapter 4: *DeDuCT*. It leverages API usages and taint analysis to characterise each line of code with the addressed concern, by applying a set of concern tags. For example, a line tagged with this set of concern tag $\{(0, io); (1, string); (3, persistence)\}$ is described as a line directly using (*depth=0*) *io* APIs and depending on data produced by other lines by means of *string* and a *persistence* APIs. Contiguous lines have been clustered according the similarity of their set of tags, thus grouping together lines performing similar operations (such related to the *io*) for similar purposes (such using manipulated *string* coming from a *persistence* layer). Finding two contiguous lines having a similarity under a give threshold (such as 0.66) determines the end of current cluster and the start of a new one.

Information about clusters and similarity can be used by the developer to identify fragment extraction opportunities related to concern cohesive code (Appendix A.1 also shows clustering information on each line). In `sorageIM()`, by using DeDuCT, 12 clusters with at least 2 lines have been found, 7 if we consider only clusters with at least 3 lines. We will consider the latter ones in order to have bigger extracted methods, thus more chances for hidden variables. Each of these clusters can be used as a *seed* for a fragment to be extracted, i.e. it can be extended by considering other contiguous lines. For example, a seed can be extended to extract all the code of a conditional branch almost totally covered by a single concern cluster, to merge contiguous clusters that are trivially working on the same object (even if for different reasons), or simply to respect syntax constraints (see Subsection 5.2.1 for details about syntax constraints).

In this long method, 7 clusters with at least 3 lines have been detected, thus 7 fragments will be selected and considered for extraction. Figure 5.9 shows fragment

```
10 C27:0.62       List params = new ArrayList();
11                //-----------------------------------------//
12 C28:0.50       VirtualFileSystem vfs = new VirtualFileSystem();
13 C28:1.00       vfs.setURI(vfsD);
14 C28:1.00       FileObject file_s = vfs.resolver(vfsD, vfs.getURI(), vfsD.getPath1());
15                //-----------------------------------------//
16 C29:0.43       UUID id = UUID.randomUUID();
```

Figure 5.9: Fragment 12-14. Replaced by the instruction: `VirtualFileSystem vfs = getVFS(vfsD)`.

```
20            } else {
21                //-----------------------------------------//
22 C32:0.58       FileContent content = file_s.getContent();
23 C32:0.77       String lastMod = "";
24 C32:0.67       if (file_s.getType().equals(FileType.FILE)) {
25 C33:0.07           DateFormat dateFormat = DateFormat.getDateTimeInstance(
26                        DateFormat.MEDIUM, DateFormat.MEDIUM);
27 C34:0.33           lastMod = dateFormat.format(new Date(content
28                        .getLastModifiedTime()));
29                //-----------------------------------------//
30            }
```

Figure 5.10: Fragment 22-28. Replaced by the instruction: `FileContent content = getContent()`.

12-14. In this case, no other lines but the cluster ones have been added. In each line information about clusters and similarity with previous line has reported. For example, line 12 has a similarity of 0.50 with the previous code line (line 10), and since it is under the considered threshold (0,66) it has been added to another cluster ($C28$).

For the second fragment shown in Figure 5.10 cluster $C32$ covers the condition of an *if* statement. Since removing this line from the selection only two lines will be considered for extraction (we want fragments of at least three lines), all the *if* statement will be considered for extraction.

Figure 5.11 shows fragment 48-52. which has selected code that is duplicated in different parts of the method, with a few changes. In this case, $C45$ has been used as a seed (it has at least 3 lines) but it is preceded by a cluster $C44$ that is very similar to the former. In fact, the similarity between bordering lines (lines 49 and 50) is just under the threshold (0.60), yet suggesting that can be both added to the same fragment. Other 2 of the 7 fragments have been selected for the same reason, i.e. fragment 38-42 and fragment 105-109.

```
44              } else {
45  C43:0.53        response = this.localRepository + id + "."
46                      + file_s.getName().getExtension();
47                  //-----------------------------------------//
48  C44:0.47        params.add("new");
49  C44:0.70        params.add(response);
50  C45:0.60        params.add(lastMod);
51  C45:0.67        params.add(content.getSize());
52  C45:0.69        params.add(lock);
53                  //-----------------------------------------//
54              }
```

Figure 5.11: Fragment 48-52. Replaced by the instruction: `setNewParams(lock, response, content)`.

Cluster $C52$ is shown in Figure 5.12. It covers almost the entire *while* statement, which internally has other nested conditional statements that make the code hard to read. However, it cannot be extracted due to the *return* statement on line 95. If a sequence of statements contain a *return* statement but not all possible execution flows end in a *return* statement, the behaviour may not be preserved (i.e. with the following execution sequence: *line 71*, *line 97*). However, *return* statements have a meaning only in the context of the original method, thus it is better to avoid their extractions. Therefore, we decide to select for extraction only the body of the *if* statement on line 76.

5.4.2 Optimise Replace Method with Method Object

By using the concern tag clusters found by DeDuCT, here used as seeds, we have found 7 fragments to be extracted, thus identifying a method *decomposition* that can be applied using the *Replace Method with Method Object* refactoring technique. We can now use the proposed optimising approach to select the proper variables to become fields. Figure 5.13 shows the *input*, *output*, and *hidden* variable sets found using data dependence analysis. Over the 18 variables (i.e. possible fields) 9 are *hidden* variables. For example, `mgr, file_d, l, respLock, file_d, e, id` are all variables that are not used by any fragment, thus considered as hidden inside the original method.

The remaining 9 variables are used as *candidates* for becoming fields. To perform the constrained enumeration the ASM has been produced. Three constraints will be considered: two for multiple output variables (F_1 and F_2) and one for the `rs`

```
71 C52:0.38       while (!rs.isAfterLast()) {
72 C52:1.00           LockFile l = new LockFile();
73 C52:1.00           respLock = l.checkLock(
74                           LockFile.getLockEnumType(rs.getString("lock")),
75                           lock);
76 C52:1.00           if ("SI".equals(respLock)) {
77                       //-----------------------------------------//
78 C52:1.00               LockFile.lockMode b1l = LockFile.getLockEnumType(rs
79                               .getString("lock"));
80 C52:0.70               if (b1l.ordinal() >= lock.ordinal()) {
81 C52:0.73                   params.add("notUpdate");
82 C52:0.80                   params.add(rs.getString("response"));
83                       } else {
84 C52:0.80                   params.add("update");
85 C52:0.80                   params.add(rs.getString("response"));
86 C52:0.80                   params.add("");
87 C52:1.00                   params.add("");
88 C52:0.90                   params.add(lock);
89 C53:0.58                   this.sqlite.updateElementInMap("response='"
90                                   + rs.getString("response") + "'",
91                                   "lock='" + LockFile.getLockType(lock)
92                                   + "'");
93                       }
94                       //-----------------------------------------//
95 C54:0.50              return params;
96                   }
97 C55:0.60           rs.next();
98               }
```

Figure 5.12: Fragment 78-93. Replaced by the instruction: `setParamsSiLock(lock, rs)`.

```
F1 (12-14):       I={vfsD} O={vfs, file_s} H={}
F2 (22-28):       I={file_s} O={content, lastMod} H={dateFormat}
F3 (38-42):       I={params, response, lastMod, lock} O={} H={}
F4 (48-52):       I={params, response, lastMod, content, lock} O={} H={}
F5 (78-93):       I={rs, lock, params} O={} H={b1l}
F6 (105-109):     I={params, response, lastMod, lock} O={} H={}
F7 (120-124):     I={params, response, lastMod, content, lock} O={} H={}
Original method: H={mgr, file_d, l, respLock, file_d, e, id}
```

Figure 5.13: The I (input), O (output), and H (hidden) sets of variables found by using a data dependence analysis.

```
asm storageIM
import StandardLibrary
signature:

enum domain Candidate = {FLD|VAR}

dynamic monitored vfsD :     Candidate
dynamic monitored vfs :      Candidate
dynamic monitored file_s :   Candidate
dynamic monitored content :  Candidate
dynamic monitored lastMod :  Candidate
dynamic monitored params :   Candidate
dynamic monitored response : Candidate
dynamic monitored lock :     Candidate
dynamic monitored rs :       Candidate

definitions:
// F1
invariant inv_multiout_b1    over vfs, file_s : vfs=FLD or file_s=FLD

// F2
invariant inv_multiout_b2    over content,lastMod : content=FLD or lastMod=FLD

// F5
invariant inv_loopindex_b5   over rs : rs=VAR

default init s1:
```

Figure 5.14: The generated ASM according to the considered constraints.

loop index (F_5).

Over the 512 (2^9) possible field combinations only 144 satisfied the enumeration constraints (368 were ruled out). For each of these, the LCOM value has been given (see Table 5.3). The field combination that minimises the lack of cohesion is $\{file\_s, params, lock, lastMod\}$ with $LCOM = 0.46$, while the worst solution is given by $\{vfsD, content, vfs\}$ with $LCOM = 0.89$.

Identified the 4 fields to be used inside the method object, the extraction of the 7 fragments can be performed, following the steps of the Replace Method with Method Object refactoring technique. A new `StorageIM` has been created, giving it a reference to the original `ImageManager` object. Instead of creating a field for every variable, only 4 of them have been selected. A constructor has been defined, giving it as input the reference of the *source* object and only one of the two original method parameters (`lock`). The second one (`vfsD`) has been given instead to the *compute()* method. This because, since `lock` has been selected as a field, it is a part of the description of the new class, and therefore it is better to give it to the constructor. For the opposite reason, since `vfsD` has been used as a variable, it is

Table 5.3: Legal field sets ordered by LCOM

Field Set	LCOM
file_s,params,lock,lastMod	0.46
file_s,params,lock,lastMod,response	0.47
params,lock,lastMod,vfs	0.50
file_s,params,lastMod,response	0.50
file_s,lock,lastMod,response	0.50
file_s,params,lastMod	0.50
file_s,params,lock,lastMod,content,response	0.50
params,lock,lastMod,response,vfs	0.50
file_s,lock,lastMod	0.50
file_s,params,lock,lastMod,content	0.50
⋮	⋮
vfsD,file_s,lock,content,vfs	0.77
vfsD,lastMod,vfs	0.78
vfsD,content,response,vfs	0.79
vfsD,file_s,lastMod,vfs	0.79
vfsD,file_s,content,response,vfs	0.80
vfsD,file_s,content	0.83
file_s,content,vfs	0.83
content,vfs	0.83
vfsD,file_s,content,vfs	0.88
vfsD,content,vfs	0.89

Table 5.4: Comparison of the number of fields for `StorageIM` using different approaches

Field selection approach	number of fields
All variables	18
Only non-hidden	9
Minimising LCOM	4

required by the *compute()* method as an input parameter.

Table 5.4 summarises the field optimisation results. The complete listing for the `StorageIM` class after the fragment extractions from *compute()* is show in Appendix A.2.

5.4.3 Fragment extractions guided by code complexity

The final version of the *compute()* method after the fragment extractions has 76 NCNB, thus it is easier to read but still a long method. Since we have opted for the creation of a separate class, thanks to the *Method Object*, we are free to make other

```
F1 (35-57): I={id, lock, content, vfs} O={} H={response, mgr, file_d}
F2 (72-87): I={id, content, lock} O={response} H={}
```

Figure 5.15: The I (input), O (output), and H (hidden) sets for the second step of fragment extractions.

extractions without being worried of overwhelming the original `ImageManager` class. Using concern tags we have extracted 7 methods showing an high concern cohesion. We can now consider other indicators to find new fragments to be extracted, such as *reducing the number of nested conditional statements*. The *compute()* method has a maximal nesting level of 4 and we want to reduce it by one level. We have so selected from the *compute()* code, shown in Appendix A.2, fragment 35-37 (i.e. a branch body of a 3-nesting level *if* statement, which has other two nested conditional statements) and fragment 72-87 (to extract a 4-nesting level *if-else* statement).

By using data dependence analysis we determined the input, output and hidden set for these two new fragments (see Figure 5.15), finding that none of them has more than one output variable, thus it is not mandatory to consider other fields to be added. We can therefore apply the two extractions, finally obtaining a *compute()* method of 38 NCNB. The final version of the `StorageIM` class is shown in Appendix A.3.

Figure 5.16 shows the obtained call graph for the `StorageIM` class. Gray coloured nodes represent the 7 concern cohesive method extracted using the indications provided by DeDuCT, while white coloured ones represents the *compute()* and the two method extracted to reduce the number of nested conditional statements. As we can see, by using both of the fragment selection heuristics, and supporting them with the use of the approach showed in this chapter, we were able to considerably reduce the size and the complexity of the original method, moving from 134 to 38 NCNB. Thanks to DeDuCT, small, low level, concern cohesive methods have been extracted, then grouped by higher level methods used to reduce code complexity. Finally, thanks to our analysis, we are also sure that the selected combination of fields is both behaviour preserving and providing a high level of class cohesion.

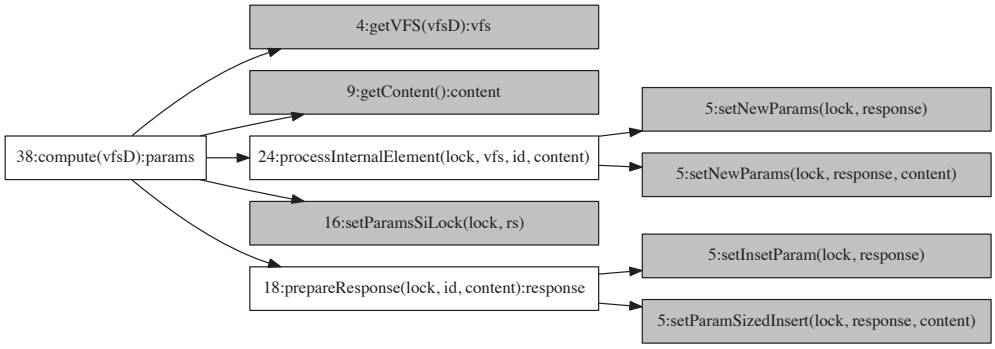

Figure 5.16: The call graph of the `StorageIM` class, showing the final method decomposition. Each node label starts with the method NCNB size. Gray coloured nodes are the ones extracted using concern clusters as seeds. The two, white coloured, nodes called by *compute()*, are the methods extracted to reduce the number of nested conditional statements. In this way, we have small, concern cohesive methods at a lower level, which are then grouped in higher level methods to reduce the code complexity of the original method. The *compute()* size has been reduced from 134 to 38 NCNB.

5.5 Conclusions

In this chapter, a data dependence based approach to effectively decompose a long method with a *Method Object* has been given. The approach automatically selects the variables to become fields, in a way that the *lack of cohesion* of the resulting *Method Object* is minimised. The proposed approach is independent from the way the method decomposition has been given by the developer, i.e. from the strategy used to select the code fragments to be extracted. However, we have shown how using the concern tagger presented in Chapter 4 let us able to automatically decompose a long method into *concern cohesive* pieces, thus finally giving to the developers an effective and automatic tool to suggest and assist complex refactoring opportunities.

Chapter 6

Data Leak detection

Some support is needed in order to shun the possibility that sensitive data handled by applications are sent to improper destinations. Although applications running on Android OS declare the accessed services, once the user accepts, the application receives complete permissions and may use sensitive data improperly.

This is a relevant problem not only from a user point of view, i.e. to preserve personal data from been leaked by an installed application, but also from a developer point of view, i.e. to assess the robustness of a developed application from data leak attempts which may be caused by exploiting particular execution flows.

Some tools have emerged to check data access and flow, either based on static or dynamic taint analysis. However both of the approaches have limits. *Static Taint Analysis (STA)* often detects false data leaks, whereas the more precise *Dynamic Taint Analysis (DTA)* introduces a significant overhead in monitored applications.

In this chapter a novel Hybrid Taint Analysis (HTA) is described, which combines static and dynamic taint analysis for detecting data leaks in Java and Android applications. The proposed approach minimizes the overhead by computing a minimal set of application points that need to be monitored and injects control code on the target application.

The proposed approach is innovative in that it takes advantage of STA and then monitors at run-time only data paths that potentially give sensitive data out, with no loss in quality with respect to DTA. We show the feasibility of our approach by providing a tool that is tailored to Android environment, tool-chain, libraries, and requirements that Android applications have to satisfy. To show the validity of the

proposed approach, a case study on some example Android applications will be finally discussed.

6.1 Preventing data leak in Android applications

Nowadays, we rely more and more on applications that handle sensitive data, which can be very critical, such as health-related information or bank details, or personal, such as address books, visited locations, etc. It is therefore an important concern for users to make sure that their applications keep as such private data. Users trusting an application can let it access, i.e. read or write, sensitive data, however the same application could improperly send sensitive data to a remote server. Therefore, users need some further assurance, e.g. from an external trusted authority, in order to shun the possibility that what they consider sensitive data are sent to improper destinations.

Most of the data protection is performed by defining security policies and avoiding untrusted applications. These can prevent programs from accessing resources, including sensitive data. However, security policies could force the user to trade between functionality and privacy. Indeed, applications running on Android OS have to declare their requirements in terms of accessed OS services, then the user is presented with such a list and decides whether to use the application. Such a mechanism falls short of the above desired assurance, since granting the application access to sensitive data makes the user at the risk of possible unaware data leaks.

More advanced systems monitor Internet traffic and deny access to certain URLs that are considered untrusted. However, a list of untrusted URLs is unlikely to be exhaustive, thus still exposing to security issues, and needs to be kept up-to-date, thus requiring a significant effort.

Some tools have emerged to check data access and flow on applications, such as, e.g., FlowDroid [6] and TaintDroid [38]. These tools are either based on *Static Taint Analysis (STA)* or *Dynamic Taint Analysis (DTA)* (which are particular classes of static and dynamic data flow analysis, see Section 2.1.7 and 2.6 for more details and fundamentals). The former brings no overhead at running time, however can have false positives, since whether a control path leaking sensitive data will be executed or not is unknown before running the application, whereas the latter can

bring a costly overhead during execution, having to monitor any access to sensitive data and the whole flow through the destinations.

In this chapter a novel *Hybrid Taint Analysis (HTA)* based tool will be presented, JADAL, for JAva DAta Leak analyser, that combines static and dynamic taint analysis to detect data leaks in Java and Android applications. The proposed approach achieves the precision of DTA without introducing the overhead of state-of-the-art dynamic approaches. The underlying idea is that static analysis aids in selecting a minimal set of program points that need to be tracked at runtime to detect data leaks.

The proposed approach is innovative in that it takes advantage of a static analysis and then monitors at runtime only data paths that potentially give sensitive data out. This approach is both light at runtime and precise. Moreover, it lets users take control on data flow analysis, and let them define which data or sources are considered as sensitive and which destinations have to be avoided for such sensitive data. E.g. a user could define a ban for an application organizing a to-do list to read their geographical coordinates, another user could instead ban the application to send their to-do list or address book to a remote service. The devised tool will check an application according to the said defined settings.

However, looking for data leak attempts is not only useful from a *user point of view*, i.e. to determine whether to trust an application to be installed, but also from a *developer point of view*. In fact, data leak analysis is also important to check the *robustness* of a developed application before it is released, validating its behaviour with respect of some data security policies. The developer may want to determine if it is possible to exploit a vulnerability in their application which can be used to leak sensitive data. By using the proposed approach, the developer can see if there is possible for an attacker to generate an execution flow making sensitive data to be sent into untrusted channels, thus having an effective tool to assess the robustness of their application from data leak attempts.

To let JADAL able to analyse Android applications, several issues related to Android environment have been addressed. Specifically, a solution to analyse and store results for the Android libraries, commonly used by many apps, have been proposed. This reduces the overhead since when processing an app we need not analyse known called libraries. As far as technical aspects are concerned, the

devised tool is tailored to Android environment, tool-chain, libraries, and typical requirements that apps have to satisfy, properly integrating the provided runtime components.

6.2 Hybrid Taint Analysis

With taint analysis, we can reduce the problem of detecting *data leaks* into Android applications to the problem of detecting whether there is a data flow between a set of program points (bytecode instructions) called *sources* and a set of program points called *sinks*. A data leak is a data flow between an instruction that gets sensitive data and an instruction that sends such data to an unauthorized channel. Typical examples of sources are calls to APIs that retrieve sensitive data, such as phone contacts or the GPS location (in a mobile phone) or the Personal Identification Number (in a smart card). Sinks can be API calls that send data over the Internet or display something on an output device.

In this section the overall hybrid JADAL approach, which combines static and dynamic taint analysis for detecting data leaks in Android application, will be described. We first describe the static taint analysis, which allows us to detect potential data leaks and suggests points of the program that we need to monitor to detect data leaks at run time. Then, the monitoring algorithm used to implement the dynamic taint analysis, i.e. determining whether a data leak occurred at runtime, will be described.

6.2.1 STA: determine points to be monitored

Given a set of program points to be used as *sources* (i.e. APIs that retrieve sensitive data) and a set of program points to be used as *sinks* (i.e. API calls that send data over untrusted channels), the first problem to detect all the potential data leaks in an Android application is to determine any possible *data flow* from any source to any sink, i.e. a sequence of instruction that cause the data coming from a source to flow into a sink.

As described in Section 2.1, the data flow in a program can be described by data dependence. Therefore, we need to construct the *Data Dependece Graph (DDG)*

of the application under analysis (see Section 2.1 for details). DDG will allow us to track any possible data flows that involve working memory (local variables, member variables, global variables, operand stack). To simplify exposition, we do not consider any flows that involve external devices or storage (writing to and reading from files, etc.). Moreover, we consider the program as sequential, and thus we consider data flows that involve one thread only. These restrictions are not intrinsic limitations since our method can be integrated with existing techniques for secondary storage taint propagation [37] and inter-thread dependence tracking [52].

We take advantage of two properties of a DDG: (1) any data flow between two instructions u and v in the program corresponds to a path between u and v in the DDG; (2) given a path $p = [v_1, \ldots, v_k]$ in the DDG and an execution σ of the program, if there is a sequence of instructions in σ that contains $v_1, \ldots, v_k$ in the specified order and for every adjacent instructions (v_i, v_{i+1}) there is a data flow between v_i and v_{i+1}, then there is a data flow between v_1 and v_k.

These properties tell us that we can reduce the problem of detecting data flows between any two instructions in the program, i.e. between a source and a sink, to the problem of checking whether in the DDG there exists a path p from the source to the sink. The existence of such a path in the DDG needs to be accessed for all the possible $(source, sink)$ pairs, i.e. determining all possible data leaks.

To efficiently find all the possible data leak, we first discard from the DDG all nodes that are not in paths between sources and sinks. To do this, we run a *Depth-first search (DFS)* starting from each source and discard all nodes that are not visited by any of the DFSs. Then we start a *reverse* DFS (following edges in reverse direction) from every sink and again discard all non-visited nodes. By removing all those *unnecessary* nodes, we obtain a *restricted DDG (rDDG)* where every node is on a path between a source and a sink.

6.2.2 DTA: the monitoring algorithm

We can now modify the input program in order to monitor the execution of every instruction corresponding to nodes in the *rDDG*, and packing the application with a *monitoring component* that will look for their execution at runtime to detect data leaks. Every time an instruction corresponding to a node in the rDDG is detected, a monitoring component will call a procedure *notifyNode(v)*. We also

Algorithm 2 Injected code for dynamic taint analysys (data flow monitoring)

Require: init()
1: ActiveFlows = $\emptyset$
Require: notifyNode(v,γ)
2: tainted=false
3: **if** $v \in$ Sources **then**
4: tainted=true
5: **end if**
6: **for all** $e \in$ ActiveFlows | target(e) = v and compatibleContext(e, γ) **do**
7: tainted=true
8: **end for**
9: **if** tainted=true **then**
10: ActiveFlows = ActiveFlows $\cup\{(v, w, \gamma)|(v, w) \in E\}$
11: **if** $v \in$ Sinks **then**
12: **output** data leak detected
13: **end if**
14: **end if**
Require: notifyIntermediate(s, γ)
15: **for all** $e \in$ ActiveFlows $|s \in DR(e)$ and compatibleContext(e, γ) **do**
16: ActiveFlows = ActiveFlows $\setminus\{e\}$
17: **end for**

insert a call *init()* at the program starts, to initialize the data structures. The pseudo-code of such two procedures is in Algorithm 2.

Procedure *init*() makes a list of *active edge flows* empty. An active edge flow corresponds to an edge of the DDG. There can be more than one active edge flow on the same edge, e.g. associated to different stack frames. Every time an instruction corresponding to a DDG node v is executed, procedure $notifyNode(v, \gamma)$ is called. Parameter γ is the current context (object reference and stack frame). $notifyNode$ (v, γ) checks if v is tainted (i.e. there is a data flow between a source and v). If v is a source, then we set variable *tainted* to true (lines 3-5). If v is the target of some active edge flows with compatible context (we discuss context later), then we have found a data flow between a tainted node and v. Therefore, we set the variable *tainted* to true (lines 6-8). At the end, if v has been marked as tainted, then its outgoing edges are added to the list of active edge flows. If v is also a sink, then a data leak is signalled.

Function "$compatibleContext(\gamma, e)$" (line 6) checks if the current context is compatible with the context of the active edge flow e. The context is important

for edges that represent intra-method data dependences and member variable data dependences. Indeed in the case of intra-method data dependences the data flow can occur only within the same stack frame. This check is necessary because two instructions in the same method may be executed in two different stack frames (e.g. in recursive calls) and hence would not produce any data flow. In the case of member variable data dependences, $compatibleContext()$ checks if the active edge flow refers to the current object instance (see Section 2.1.6 for more details about the different types of data dependence).

6.2.3 Checking edge triggering

When we detect, at runtime, a data flow involving an edge e of the DDG we say that e is *triggered*. Checking *edge triggering* is not trivial since monitoring the two endpoints is not enough. Indeed, a target instruction s_2 of a data dependence executed after the source s_1 would not necessarily result into a data flow between s_1 and s_2. E.g., a variable written by s_1 may have been modified before s_2 is executed. We need a way to remove edge flows that are not triggered from the active edge flow list, before s_2 is notified.

Informally, we say that an edge flow (u, v, γ) is triggered at a certain point of the execution of the input program, if instruction v is being executed and a data flow compatible with the data dependence (u, v) is detected. Checking edge triggering requires monitoring a set of instructions in the program (not only u and v).

To better describe the concept, let us consider a data dependence (u, v) corresponding to a global variable λ. We want to check if (u, v) is triggered during the execution. That means we want to check if the value written on λ by u is read by v. For this purpose, we monitor u, v and all other instructions that write on λ. When u is executed, (u, v, γ) become an *active* edge flow: the value written in λ may be read from v at some point in the future, but we cannot say that this will happen. If at a later stage we reach another instruction that writes on λ, then (u, v, γ) is *dropped*: the value written by u cannot be read by v, since it has been replaced. Therefore, (u, v, γ) is not active any more. Otherwise, if edge (u, v) is still active and instruction v is reached, then the edge flow is considered *triggered*: we detected a data flow compatible with (u, v).

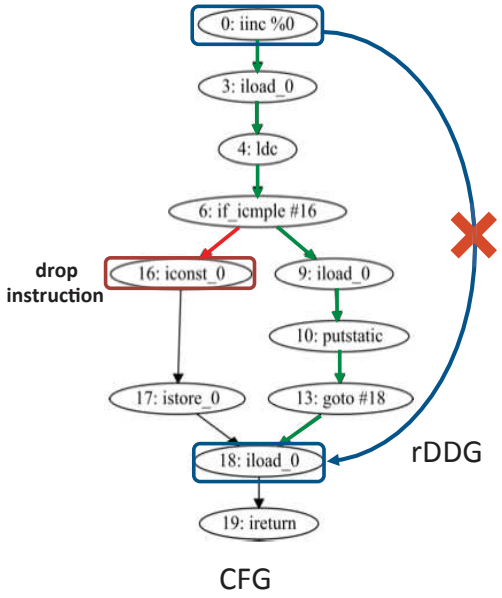

Figure 6.1: An example of dropping edge for an intra-method data dependence.

During the static analysis we compute the sets of dropping instructions $DR(e)$ for every edge e in the restricted DDG. We then inject code into the program to receive notification every time an instruction in $DR(e)$ is executed. When the execution of an instruction $w \in DR(e)$ is notified, we remove every affected active edge flow with compatible context from the active edge flow list (see $notifyIntermediate()$ procedure in Algorithm 2).

6.2.4 Determine Dropping Instructions

The computation of $DR(e)$ depends on the specific case. We now describe how to compute $DR(e)$ (set of dropping instructions) and check the context, by means of function $compatibleContext(e, \gamma)$. These procedures depend on the kind of data dependence represented by the edge e in the DDG.

6.2.4.1 Intra-method data dependences

To monitor if an intra-method data dependence (u, v) between u and v is dropped we consider the set P of instruction causing the data flow involving edge (u, v). We want to determine if the program execution *leaves* P with the minimum numbers of

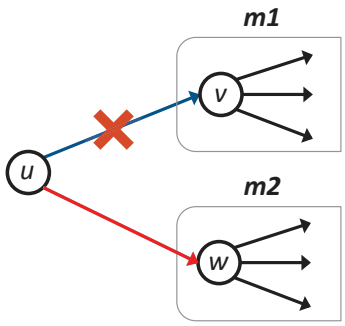

Figure 6.2: An example of dropping edge for an inter-method data dependence.

monitoring instructions. To solve this problem, we will add to $DR((u, v))$ all nodes that are endpoints of edges starting from P and ending in $G \setminus P$ (set difference), where G is the data dependence graph.

As an example, Figure 6.1 shows the *Control Flow Graph (CFG)* of a simple method. As explained in Section 2.1.3, since instruction 0 writes on a memory cell (a local register) that is further read by instruction 18, and there is a path π from 0 to 18 (depicted with green edges in Figure 6.1) where no other instruction overrides the written value, then we can say that there is a data dependence from 0 to 18, thus the edge $(0, 18)$ will be in the DDG. Let us suppose that this edge was selected to be a part of the *restricted DDG (rDDG)* (depicted with a blue edge in Figure 6.1): to determine if edge $(0, 18)$ is *triggered*, we cannot just monitoring and notify 0 and 18, because we need to be sure that all the sequence of instructions on π in the CFG have been also executed, i.e. we stayed on the path causing the data flow involving edge $(0, 18)$ in the rDDG (causing the edge triggering). However, instead of monitoring all the nodes on π, we can just notify the monitoring component when the execution *leaves* the path, i.e. on instruction 16. In fact, if the node 16 is notified at runtime, we are sure that the execution leaved path π thus causing the data flow on the edge $(0, 18)$ to be *dropped*.

A flow on an intra-method data dependence can occur only within the same stack frame. Therefore, if e is an intra-method data dependence, $compatibleContext(e, \gamma)$ checks that the stack frame of edge flow e is the same as the one of γ.

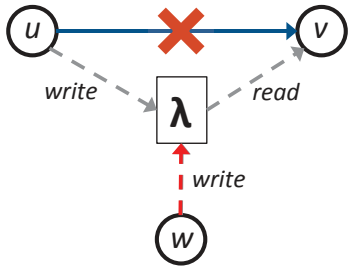

Figure 6.3: An example of dropping edge for a member variable dependence.

6.2.4.2 Inter-method data dependences

Inter-method data dependences represent information flow during a call to or the return from a method. In correspondence with a call, instructions of the caller that prepare call parameters are connected to instructions of the callee that utilize such parameters. If the call is static, then the inter-method data dependences corresponding to parameter transfers are always triggered. If the call is virtual or dynamic, then there may be more than one inter-method data dependence starting from an instruction. Only one call among the ones considered in the dispatching table is dispatched, therefore only one edge flow is triggered. Return data dependencies are managed similarly. Indeed a return instruction can have many data dependencies (one per caller) but only one of these will be dispatched.

Given a set of inter-method data dependences $\{(u, v_i)|i = 1, \ldots, k\}$ corresponding to a method call or a method return and that start from the same instruction u, the dropping instructions are assigned for each i as $DR((u, v_i)) = \{v_j|j = 1, \ldots, k; j \neq i\}$. That is, once we know which of the methods in the dispatching method has been called, all the active edge flows related to the other methods in the dispatching table are dropped (as depicted in Figure 6.2).

Note that the flow in inter-method data dependences does not depend on the context. Therefore, if e is an inter-method data dependence, $compatibleContext(e, \gamma)$ returns true.

6.2.4.3 Member variable dependences

For member variable data dependences $e = (u, v)$ on a member variable λ, if $W(\lambda)$ and $R(\lambda)$ are the set of instructions that write to and read from λ, respectively,

then $DR(e) = W(\lambda)\backslash\{u\}$. That is, if any other instruction w, different from u, writes on the member variable λ before v can read it, the active edge flow (u, v, γ) is dropped (as depicted in Figure 6.3).

For non-static member variable dependences (u, v) there is a data flow between u and v only if u and v refer to the same object instance. Therefore, in this case $compatibleContext(e, \gamma)$ checks that the object instance of the context of edge flow e is the same as the one of γ. For static member variable dependences $compatibleContext(e, \gamma)$ always returns true.

6.3 JADAL implementation

The JADAL approach described in Section 6.2 has been implemented by a set of separate Java services, tailored with a set of third party tools that are needed, e.g. to unpack and repack Android APK files (the Android application package), gathered from the official Android SDK. Figure 6.4 shows the entire workflow, that starting from the Android input application uses static analysis techniques to identify a minimal set of points to be monitored to enable dynamic taint analysis, i.e. the dynamic data leak detection during execution. JADAL will finally generate a signed and executable Android package that includes the components handling leak detection. As a result, modifications to the underneath Android OS are not required and the data-leak self-checking application can be executed as any other app.

As Section 6.2 described, JADAL needs both the CFG and the DDG to determine the program points to be monitored at runtime to intercept data leak attempts. Thus, starting from the application class files, a third part static analyser, namely JavaPDG [89], is used to obtain a graph representation of the input application, called Program Dependence Graph (PDG) [40] that combines CFGs and DDGs of each class method, and uses the call graph to link them together.

The PDGs for the classes of the input application are then given to the first JADAL component, the `Inspector`. This is responsible for the initialization of the internal data used by JADAL algorithms. Since building the PDGs is a costly operation, in terms of CPU time and memory consumption, we have to reduce as much as possible the program parts we need to analyse each time a new input

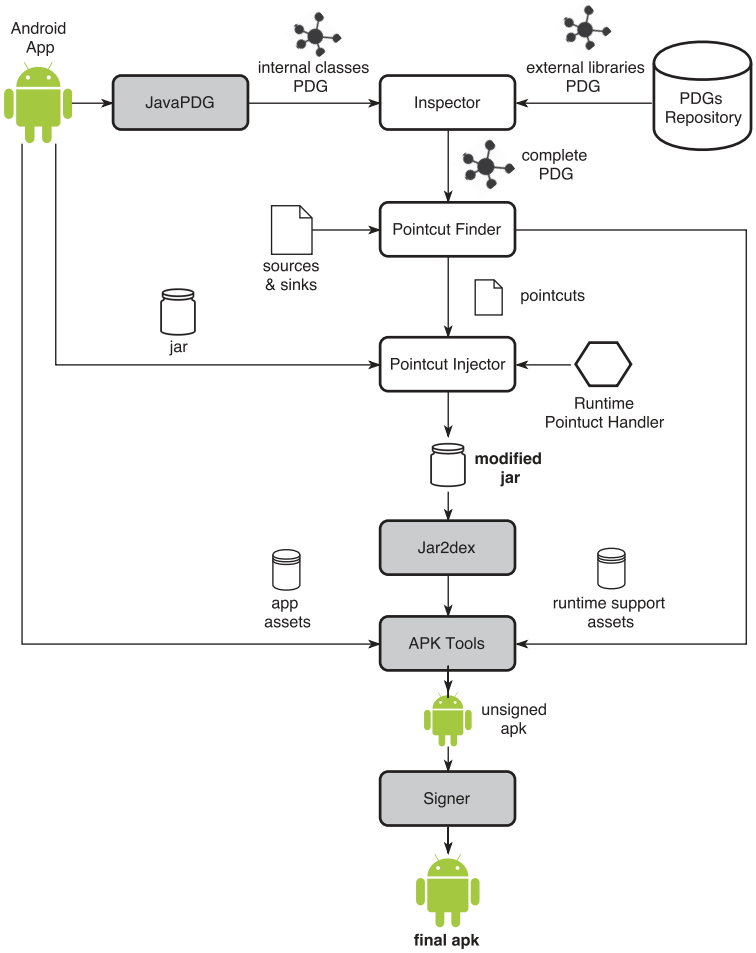

Figure 6.4: The workflow of the JADAL Framework; gray elements are external components, while white ones are internal.

application is given. Thus, we have to reuse as much as possible previously computed PDGs to shorten response time of the JADAL system. Therefore, commonly used external libraries (such as the Android SDK) are analysed by JADAL only once, storing their computed graph representations in a `PDGs Repository`. Such graphs are retrieved from the repository every time they are needed. Inspector links together the PDGs of the *internal* classes (newly computed by JavaPDG) with the external library representations (previously computed and stored into the repository).

Once the complete PDG has been built, the `Pointcut Finder` runs the JADAL statical analysis described in Section 6.2. We define a list of sources and sinks. For example, methods that read the GPS position are considered as sources whereas methods to send data through the network are considered as sinks. This list is taken as input by the `Pointcut Finder` to determine the minimal set of program points to be monitored. The algorithm produces both the list of *pointcuts*, one for each of the selected program points, and the subgraph induced by these points, including the *restricted DDG (rDDG)*. Since this graph is further required to assist the monitoring component, it is saved and staged for inclusion in the final APK as an external asset.

The actual modification of the input application is performed by the `Pointcut Injector`, that uses BCEL[1] to manipulate the Java bytecode of the composing class files. A *pointcut* is a program point, i.e. a bytecode instruction, where a method call to the monitoring component need to be injected (just before the instruction). Other than the pointcuts list, this component takes as input the class files of the input application, as a JAR archive, and the monitoring component that will be invoked every time a pointcut is reached at running time.

Since both JavaPDG and BCEL only work with the bytecode of a canonical JVM (that differs from *Dalvik*[2]), JADAL modifies the JAR class package, instead of the *DEX* file (*Dalvik Executable Format*), of the input application. The modified JAR produced by `Pointcut Injector` is converted to an Android package by an external tool that converts the JAR output to the DEX format and inserts it in the final APK, together with the original assets (images, datasets and other

[1] https://commons.apache.org/proper/commons-bcel/
[2] The VM used by Android OS

Table 6.1: Computation time for the Android SDK DDGs

Package	Computation Time(s)	Classes	Methods	Instructions
java.io.*	257	80	356	3969
javax.xml.*	431	54	355	1578
android.service.*	456	19	189	937
android.hardware.*	809	67	504	2896
android.telephony.*	899	33	399	2216
android.text.*	1649	141	977	5120
android.app.*	2609	130	1648	8213
java.lang	2670	135	1199	7147
android.view.*	3045	190	2480	11629
java.util.*	4333	244	2771	14393
...	...	...	...	...
Complete Android SDK	>14 h	3673	30574	164555

non-code components of the input application) and the data structures created in the previous step to support the monitoring activity. The workflow is completed by signing the produced APK with a keypair associated to JADAL.

6.3.1 Dealing with external libraries

An input application can have several method calls to external libraries. Even a simple Java application that manipulates string contents needs to import at least the *java.lang* standard library. This is also true for the Android applications, e.g. every time we want to popup a message using the *Toast*[3] class of the Android Standard Library (also called *Android API*), we will have to import the related package (i.e. the *android.widget*).

In this case, data can flow inside external methods; therefore, to correctly apply the JADAL static analysis we have to consider the DDG of both the internal classes and all the external packages that are used by the input application. This is also necessary every time we want to consider as source and sink methods that belong to external APIs.

This leads to three issues and considerations: (i) standard libraries are not shipped with the Android applications (since they are part of the runtime environment), but they need to be considered; (ii) although third-part libraries (such as the Facebook SDK[4] or external Ads libraries) might be included in the analysed

[3]https://developer.android.com/reference/android/widget/Toast.html
[4]https://developers.facebook.com/docs/android

```
public class MainActivity extends AppCompatActivity {
 ...
 triggerbtn.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View v) {
   Location location = lm.getLastKnownLocation(LocationManager.GPS_PROVIDER);
   double longitude = location.getLongitude();// source
   double longitudeMod = longitude + 100;
   UntrustedChannel.send(longitudeMod);   // sink
  }
 });
 ...
}
```

Figure 6.5: Java code of the callback method containing the data leak attempt. This will be actually executed only by pressing the button inside the Activity at runtime.

```
API.class                MainActivity.class       R$bool.class
R$drawable.class         R$layout.class           R$style.class
BuildConfig.class        R$anim.class             R$color.class
R$id.class               R$mipmap.class           R$styleable.class
MainActivity$1.class     R$attr.class             R$dimen.class
R$integer.class          R$string.class           R.class
```

Figure 6.6: List of the classes composing the input Android app. We expect that only the MainActivity$1.class will be modified, since it is the only internal class involved in a data leak attempt (according to selected source and sink, shown in Figure 6.7).

app, usually only a few methods are actually called, making the static analysis of the whole library unnecessary; (iii) both standard and third-part libraries can be used by different applications, and we should analyse them only once.

External libraries typically have many classes and methods, hence building the related DDGs is a resource consuming task. Table 6.1 shows the computation times needed to build the DDGs for each of the Android SDK packages (API level 21) using a host with two 6-core Intel(R) Xeon® E5-2620 CPUs (2.00GHz) and 32GB of DDR3 memory (1333 MHz). As we can see, more than fourteen hours are needed for the whole SDK. To save computation time, JADAL uses a `Repository` that can be queried with the name of the required package and returns the related PDG, if available. While building the PDG the `Inspector` looks for each method call that points to external libraries and asks the `Repository` to retrieve the related representations. It then links them together with the PDG of the input application.

```
# sources.txt
android.location.Location.getLongitude()D

# sinks.txt
com.example.simpleapp.API.show(D)V
```

Figure 6.7: Content of sources and sinks file for this simple example. Both an external (as source) and internal (as sink) method have been chosen to show the DDG completion done by the `Inspector` component.

Figure 6.5 shows an excerpt of the code of a simple Android app, that we used as a case study. The application consists of 16 classes, as showed in Figure 6.6, We introduce a data leak attempt within the *onClick()* callback method of the *MainActivity* class. To produce the complete PDG corresponding to the listed method, we analyse both the bytecode of the *getLastKnownLocation()* method in the *LocationManager* class and the *getLongitude()* method in the *Location* class, both provided by the *android.location* of the Android standard library. This causes the `Inspector` to query the `Repository`, for the missing package's PDG.

6.3.2 Pointcut Finder

Once the complete PDG has been built by the `Inspector`, JADAL proceeds with the `Pointcut Finder` step, which searches for candidate data flow paths linking a source with a sink, and then provides a reduced set of application points (*pointcuts*) to be monitored at runtime to avoid data leaks (as explained in Section 6.2).

First part of Figure 6.8 shows the bytecode of the callback method previously introduced (see Figure 6.5) and Figure 6.7 shows the source and the sink we want to use for this simple example. We consider one source from package *android.location* as method *Location.getLongitude()D* and one sink from package *com.example.simpleapp* as method *UntrustedChannel.send(D)V*. Both an external (as source) and internal (as sink) method have been chosen to test the actual DDG completion with external libraries done by the `Inspector` component.

The said method *send()* attempts a data leak, as it tries to send the GPS longitude position (trivially modified by adding 100 to the value) on an untrusted channel (here given by a method of the *UntrustedChannel* internal class). JADAL is able to recognise this threat and provides a list of application points to be

```
BEFORE INJECTIONS:
public void onClick(android.view.View);
0:  getstatic    MainActivity.lm:LocationManager
3:  ldc          "gps"
5:  invokevirtual LocationManager
8:  astore_2     .getLastKnownLocation(String)Location
9:  aload_2
10: invokevirtual Location.getLongitude()D
13: dstore_3
14: dload_3
15: ldc2_w       100.0d
18: dadd
19: dstore       5
21: dload        5
23: invokestatic UntrustedChannel.send(D)V
26: return

AFTER INJECTIONS:
public void onClick(android.view.View);
0:  new          <java.util.Random>
3:  dup
4:  invokespecial java.util.Random.<init>()
7:  invokevirtual java.util.Random.nextInt()
10: istore       %7
12: getstatic    MainActivity.lm
15: ldc          "gps"
17: invokevirtual getLastKnownLocation(String)
20: astore_2
21: aload_2
22: invokevirtual Location.getLongitude()D
25: ldc          1389
27: iload        %7
29: invokestatic handler.AppObserver.handlePointcut(II)V
32: dstore_3
...
```

Figure 6.8: Bytecode of the callback method before and after pointcuts injection. *LocationManager* and *Location* belong to the *android.location* external package, while *UntrustedChannel* to the *com.example.simplapp* internal package.

```
1389:com.example.simpleapp.MainActivity$1.onClick(Landroid/view/View;)V:13
1390:com.example.simpleapp.MainActivity$1.onClick(Landroid/view/View;)V:14
1392:com.example.simpleapp.MainActivity$1.onClick(Landroid/view/View;)V:18
1393:com.example.simpleapp.MainActivity$1.onClick(Landroid/view/View;)V:19
1394:com.example.simpleapp.MainActivity$1.onClick(Landroid/view/View;)V:21
```

Figure 6.9: Pointcuts automatically selected by the `Pointcut Finder`. As expected, they are all located inside the callback method.

monitored at runtime (shown in Figure 6.9). Each pointcut is defined by an *ID*, a *method signature* and an *instruction offset*, which refer to the method and the bytecode instruction where the code that calls the monitoring component must be injected. E.g., the first entry in the pointcut list (Figure 6.9) refers to the *dstore_3* instruction having offset 13 in the *onClick()* method.

6.3.3 Pointcut Injection

For each selected pointcuts, the `Pointcut Injector` introduces into the bytecode a call to the *notifyNode(v, γ)* method of the monitoring component (see Algorithm 2). This requires both the pointcut ID to identify the node and some information about the current context of the method execution. This consists in (i) the identification of the stack context and (ii) the object reference. In the second part of Figure 6.8 we can see how the method has been automatically modified. The instructions from 0 to 10 generate a random number as a unique reference to the current stack context, while the instructions from 25 to 29 are injected just before the `1389:dstore_3` pointcut execution to notify the monitoring component (`AppObserver`). The call to AppObserver gives as parameters both the pointcut id and the stack context reference (here the object reference is not required since the method is static).

It is worth noting that among all of the 16 classes composing the analysed Android application (see Figure 6.6) only the `MainActivity$1.class` has been modified due to pointcut injection. In fact, looking at the produced pointcut list, shown in Figure 6.9, we can see that all of them refer to the *MainActivity$1* internal class, since it is the only one involved in a data leak attempt. This is an important result, since only the application parts that are actually involved in a potential data leak will be monitored at runtime, thus reducing the introduced overhead if compared with traditional dynamic approaches.

6.3.4 Producing the final APK

One of the last steps needed to run the output application on an Android device, is the generation of a signed APK, starting from the resources of the original application and the data collected during the static analysis. This process

is performed through a script that makes use of a set of tools to: (i) reverse engineering the Android APK file (*apk-tool*[5]); (ii) retrieve existing assets and jars from DEX files (*dex2jar*); (iii) inject the data leak handling logic and the results obtained from static analysis; (iv) create new DEX files (*jar2dex*); (v) produce a new APK (again *apk-tool*); (vi) sign the produced APK (*jarsigner*).

To enable the `Runtime Pointcut Handler` it is necessary to update the *AndroidManifest.xml* file. This file contains data that describe the application, such as basic information, available activities, required permissions, etc. Among the others, section `application` is quite relevant: it contains definitions of components/subclasses. Whenever a subclass is defined, it is declared through a name attribute that must include the full package designation. The name gives the proper context to load the restricted DDG computed in the previous steps.

6.4 Experiments

In order to demonstrate the JADAL effectiveness, a toy app, `LeakingAppAndroid`, has been built to simulate access to sensitive and unsensitive data, and stream to authorised and unauthorised channels. We apply the JADAL tool to LeakingAppAndroid to detect whether there is a data leak, i.e. sensitive data sent to unauthorised channels. Figure 6.11 (on the right) shows the user interface of the app, which has two buttons on the top of the main activity interface, allowing the user to load sensitive or unsensitive data, respectively, and keep them in main memory. The user can select the output channel through the two radio buttons in the center of the interface and then send previously loaded data to it, by using button "SEND" in the center. If the "Authorized" channel is set, data are sent to the "Authorized Channel" edit box right below the send button, and a sequence of password digits is displayed (e.g. "654653"). If the "Unauthorized" channel is set, data are visualized in the "Unauthorized Channel" edit box. A data leak occurs if the user loads sensitive data and then sends them to the unauthorized channel.

Figure 6.10 shows the class diagram of LeakingAppAndroid. Class "App" holds the status of the program, which comprises loaded data and output channel. The output channel is a reference to the interface "ISend" that contains a method for

[5]http://ibotpeaches.github.io/Apktool/

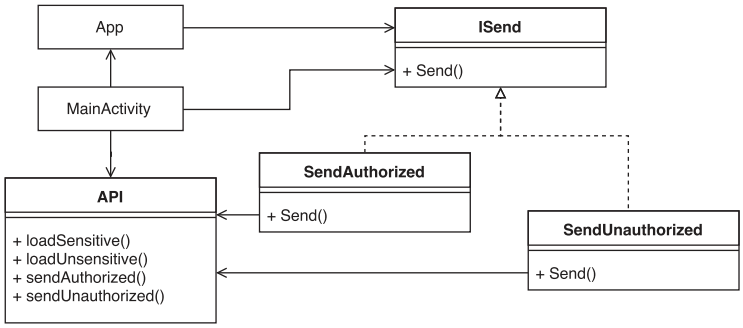

Figure 6.10: Class diagram of the LeakDetectionApp.

sending data. This interface has two implementations: "SendAuthorized" and "SendUnauthorized". The user can switch between these two implementations through the radio buttons in the main activity. Class "MainActivity" controls the user interface by managing buttons and edit controls. Class "API" represents the interface with the device, hence it contains methods for loading and sending data. A data leak in LeakingAppAndroid corresponds to a data flow from the return value of "loadSensitive()" to the input parameter of "sendUnauthorized()".

Using the static analysis described in Section 6.2 JADAL setected only 5 program points to be monitored at runtime to detect data leaks (see the table on the top-left of Figure 6.11),

Figure 6.11 (on the bottom-left) shows a part of the LeakingAppAndroid DDG with a potential data leak. It involves 3 vertices (4872, 5070, and 5086) and 2 edges were found in paths between the source (return value of "loadSensitive()") and the sink (parameter of "sendUnauthorized()"). Instruction 4872 is executed when the user presses button "GET SENSITIVE". It takes the return value of loadSensitive() and writes it on the variable `App.data`. Instruction 5070, called when button "SEND" is pressed, gets the value of App.data and puts it in the operand stack to be used as a parameter for method Send() of the output channel. Instruction 5086 loads the method parameter and passes it to sendUnauthorized().

The data leak shown in Figure 6.11 can be detected by statically analysing the DDG of the app. However, a data leak identification by means of static analysis does not mean that it will take place. In this case, the data leak occurs only if the user performs a certain *sequence of operations* (i.e. (i) load sensitive data, (ii) select an unauthorized channel, and (iii) send data). In general, data leaks detected by

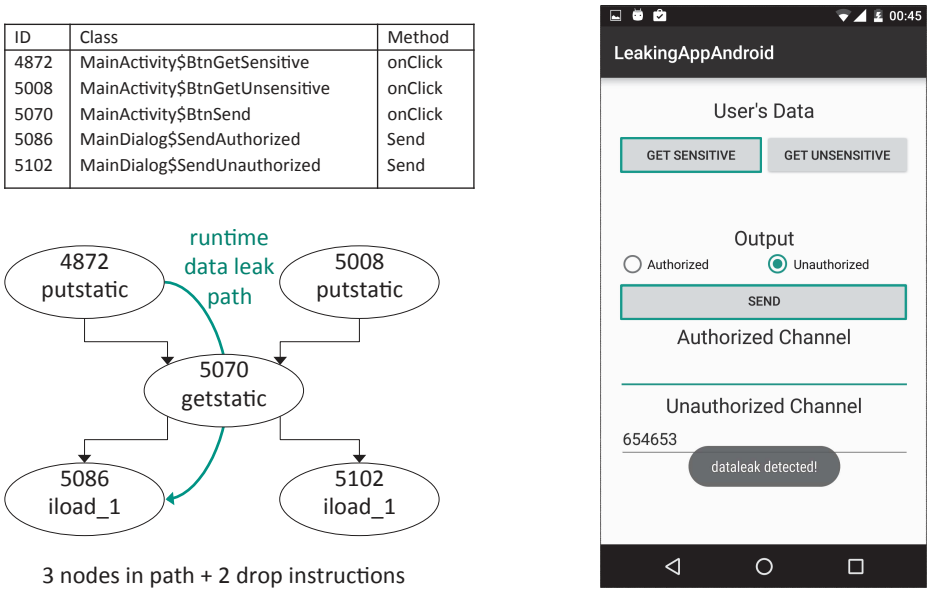

Figure 6.11: A part of the DDG of LeakingAppAndroid (on the bottom-left) limited to the 5 selected program points to be monitored at runtime (see table on the top-left), and a screenshots of the app (on the right) showing a detected data leak.

static analysis may never occur. E.g. the app could implement a business logic that sends sensitive data to authorized channels and non-sensitive data to unauthorized channels, hence a false data leak would be detected by static analysis.

JADAL injected control code on LeakingAppAndroid to monitor the two edges of the leaking path and notify a data leak when it actually occurs during runtime (as shown on in the app screenshot in Figure 6.11). A manual verification confirmed that a data leak is signalled if and only if it really occurs. Specifically, a data leak is advised if the user presses "GET SENSITIVE" and then "SEND" after selecting an unauthorized channel in the modified Android app produced by the JADAL tool.

If the data flow is somehow broken between the source and the sink, no data leak is notified. E.g., if the user presses "GET UNSENSITIVE" after pressing "GET SENSITIVE and before pressing "SEND", then no data leak is notified.

6.5 Conclusions

In this chapter, a novel hybrid taint analysis approach for detecting data leaks in Android apps was proposed, which combines static and dynamic taint analysis in order to perform reliable monitoring while introducing a low overhead in target apps. In addition to its efficiency, the proposed approach provably detects all data leaks that can be found by dynamic taint analysis.

An implementation of the JADAL tool was also described, showing how to automatically modify an Android application making it "data leak aware" and able to be run in an normal Android device (i.e. without any customisation due to monitoring needs, which are instead embedded by JADAL inside the app).

We showed the feasibility of the approach by presenting the results obtained on a sample Android app that simulates loading sensitive data and sending them to an unauthorized channel. The combined taint analysis shows that JADAL can correctly manage use cases that cannot be handled by static analysis alone, e.g. in presence of polymorphism, and it has a low overhead since it requires monitoring only a minimal set of program points.

Chapter 7

Conclusions

This thesis has presented a support to assist evolving software based on data usage, showing how information about data dependence and propagation can give useful insights about different aspects concerning software modularity, portability and robustness. The devised data dependence and taint analysis approaches have been applied on real world scenarios, demonstrating how data can give important clues about code behaviour and responsibility, which can be used to devise automatic tools to support software evolution.

Data dependence and taint analysis were helpful to automatically assess the dependence on a particular platform. By proposing a taint analysis approach it was possible to understand which instructions depend on platform specific APIs, either explicitly or implicitly, thus proposing a portability metric that is more accurate than traditional ones. In fact, the latter consider only explicit interactions with the underlying platform, thus not taking into account the instructions having an implicit platform dependence, i.e. using data coming from, or direct to, platform APIs, which need to be considered while measuring *code portability*.

By extending the concept of tracing data coming from specific APIs to assert portability, we presented DeDuCT, which is an automatic tool tagging code instructions with concern information. It is innovative in that it uses the APIs as a source of concern identification. Then, it finds all (and only) the data dependent code lines, regardless of the number of lines that can be in between, giving them the concern tag suggested by the used APIs.

Concern tags have been used to suggest *Extract Method* refactoring opportunities,

identifying fine grained code fragments exposing a high concern cohesion, i.e. addressing a specific task, and moving them in separate methods, thus helping in improving *code modularity*.

Moreover, the provided concern tagger offered the possibility to suggest *complex* refactoring opportunities, by automatically identifying a long method decomposition into concern cohesive pieces. This was achieved by providing an automatic tool to apply the *Replace Method Method Object* refactoring technique.

The approach automatically selects the variables to become fields, in a way that the *lack of cohesion* of the resulting *Method Object* is minimised. The effectiveness of the proposed refactoring tools has been proved and explained by analysing real world scenarios of long methods, whose complexities has been reduced by decomposing the code in concern cohesive pieces, easier to understand and maintain.

Another important property addressed in this thesis was *code robustness* against sensitive data disclosure. It has been shown how using a novel hybrid approach, JADAL, combining static and dynamic taint analysis, it is possible to assist developers in assessing if the application they developed can be exploited to leak user data, i.e. by forcing the execution of a particular sequence of instructions eventually leading to a data leak.

It was demonstrated how JADAL can handle situations that cannot be addressed by using only a static approach, e.g. because of polymorphism. Moreover, dynamic analysis overhead has been lowered by selecting only a minimal set of program points to be monitored at runtime.

In conclusion, different aspects related to software evolution have been addressed by leveraging similar concepts related to data dependence and taint analysis. By means of properly devised approaches and tools, it was experimentally demonstrated how portability, modularity and robustness can be better investigated by following data usages and propagation inside the application.

Assisting the developers in evolving modern software systems means providing tools which timely address design issues, hence making developers able to focus on fast and frequent changes rather then on a detailed and fixed design. By means of the proposed approaches, the deterioration of code structural quality can be automatically addressed either during the development of new software components or to analyse and improve legacy applications.

Appendix A

Listings

A.1 The `storageIM()` method before applying the *Replace Method with Method Object*

```
1     public class ImageManager implements ImageManagerPlugin {
2         private Logger logger;
3         private SQLite sqlite;
4         private String localRepository;
5         [...]
6         public List storageIM(VFSDescription vfsD, LockFile.lockMode lock)
7                 throws FileSystemException, CleverException, Exception {
8 C26:0.00     String response = "";
9             String respLock;
10 C27:0.62    List params = new ArrayList();
11            //-----------------------------------------//
12 C28:0.50    VirtualFileSystem vfs = new VirtualFileSystem();
13 C28:1.00    vfs.setURI(vfsD);
14 C28:1.00    FileObject file_s = vfs.resolver(vfsD, vfs.getURI(), vfsD.getPath1());
15            //-----------------------------------------//
16 C29:0.43    UUID id = UUID.randomUUID();
17            try {
18 C30:0.43        if (!file_s.exists()) {
19 C31:0.50            params.add(response);
20                } else {
21                    //-----------------------------------------//
22 C32:0.58            FileContent content = file_s.getContent();
23 C32:0.77            String lastMod = "";
24 C32:0.67            if (file_s.getType().equals(FileType.FILE)) {
25 C33:0.07                DateFormat dateFormat = DateFormat.getDateTimeInstance(
26                            DateFormat.MEDIUM, DateFormat.MEDIUM);
27 C34:0.33                    lastMod = dateFormat.format(new Date(content
```

```
28                                  .getLastModifiedTime()));
29                          //-----------------------------------------//
30                          }
31  C35:0.35                logger.debug("element searched : " + file_s.getName().getURI());
32  C36:0.56                ResultSet rs = this.sqlite.retrieveElementsInMap(file_s
33                                  .getName().getURI());
34  C37:0.64                if ((!rs.isAfterLast() && !rs.isBeforeFirst())) {
35  C38:0.46                    if (file_s.getType().equals(FileType.FOLDER)) {
36  C39:0.47                        response = "" + this.localRepository + id;
37                                  //-----------------------------------------//
38  C40:0.46                        params.add("new");
39  C41:0.55                        params.add(response);
40  C42:0.45                        params.add(lastMod);
41  C42:0.70                        params.add("");
42  C42:0.90                        params.add(lock);
43                                  //-----------------------------------------//
44                              } else {
45  C43:0.53                        response = this.localRepository + id + "."
46                                          + file_s.getName().getExtension();
47                                  //-----------------------------------------//
48  C44:0.47                        params.add("new");
49  C44:0.70                        params.add(response);
50  C45:0.60                        params.add(lastMod);
51  C45:0.67                        params.add(content.getSize());
52  C45:0.69                        params.add(lock);
53                                  //-----------------------------------------//
54                              }
55  C46:0.42                    FileSystemManager mgr = VFS.getManager();
56  C46:1.00                    FileObject file_d = mgr.resolveFile(response);
57  C47:0.58                    vfs.cp(file_s, file_d);
58  C47:1.00                    if (file_s.getType().equals(FileType.FOLDER)) {
59  C48:0.15                        logger.debug("Image1");
60  C49:0.08                        this.sqlite.insertElementAtMap(file_s.getName()
61                                          .getURI(), response, "", lastMod, LockFile
62                                          .getLockType(lock));
63                              } else {
64  C50:0.08                        logger.debug("Image2");
65  C51:0.08                        this.sqlite.insertElementAtMap(file_s.getName()
66                                          .getURI(), response,
67                                          new Long(content.getSize()).toString(),
68                                          lastMod, LockFile.getLockType(lock));
69                              }
70                          } else {
71  C52:0.38                    while (!rs.isAfterLast()) {
72  C52:1.00                        LockFile l = new LockFile();
73  C52:1.00                        respLock = l.checkLock(
74                                          LockFile.getLockEnumType(rs.getString("lock")),
75                                          lock);
76  C52:1.00                        if ("SI".equals(respLock)) {
77                                      //-----------------------------------------//
```

```
78  C52:1.00                    LockFile.lockMode b11 = LockFile.getLockEnumType(rs
79                                      .getString("lock"));
80  C52:0.70                    if (b11.ordinal() >= lock.ordinal()) {
81  C52:0.73                        params.add("notUpdate");
82  C52:0.80                        params.add(rs.getString("response"));
83                              } else {
84  C52:0.80                        params.add("update");
85  C52:0.80                        params.add(rs.getString("response"));
86  C52:0.80                        params.add("");
87  C52:1.00                        params.add("");
88  C52:0.90                        params.add(lock);
89  C53:0.58                        this.sqlite.updateElementInMap("response='"
90                                          + rs.getString("response") + "'",
91                                          "lock='" + LockFile.getLockType(lock)
92                                          + "'");
93                              }
94                              //-----------------------------------------//
95  C54:0.50                    return params;
96                          }
97  C55:0.60                    rs.next();
98                      }
99  C56:0.46                if (file_s.getType().equals(FileType.FOLDER)) {
100 C57:0.47                    response = this.localRepository + id;
101 C58:0.50                    this.sqlite.insertElementAtMap(file_s.getName()
102                                     .getURI(), response, "", lastMod, LockFile
103                                     .getLockType(lock));
104                             //-----------------------------------------//
105 C59:0.43                    params.add("insert");
106 C60:0.55                    params.add(response);
107 C61:0.45                    params.add(lastMod);
108 C61:0.70                    params.add("");
109 C61:0.90                    params.add(lock);
110                             //-----------------------------------------//
111                         } else {
112 C62:0.53                    response = this.localRepository + id + "."
113                                     + file_s.getName().getExtension();
114 C63:0.07                    logger.debug("Image4");
115 C64:0.08                    this.sqlite.insertElementAtMap(file_s.getName()
116                                     .getURI(), response,
117                                     new Long(content.getSize()).toString(),
118                                     lastMod, LockFile.getLockType(lock));
119                             //-----------------------------------------//
120 C65:0.46                    params.add("insert");
121 C65:0.70                    params.add(response);
122 C66:0.60                    params.add(lastMod);
123 C66:0.67                    params.add(content.getSize());
124 C66:0.69                    params.add(lock);
125                             //-----------------------------------------//
126                         }
127 C67:0.42                FileSystemManager mgr = VFS.getManager();
```

```
128 C67:1.00              FileObject file_d = mgr.resolveFile(response);
129 C68:0.58              vfs.cp(file_s, file_d);
130 C69:0.50              return params;
131                   }
132               }
133           } catch (Exception e) {
134 C70:0.00      logger.error(e.getLocalizedMessage());
135           }
136 C71:0.00   return params;
137       }
138       [...]
139   }
```

A.2 Refactored version of the `storageIM()` method by means of a *Method Object*

```
1     public class StorageIM {
2         private final ImageManager im;
3         //private Logger logger; --> im.logger
4         //private SQLite sqlite; --> im.sqlite
5         //private String localRepository; --> im.localRepository
6
7         // Fields selected by the tool
8         private LockFile.lockMode lock;
9         private String lastMod;
10        private List params;
11        private FileObject file_s;
12
13        public StorageIM(ImageManager source, LockFile.lockMode lock) {
14            this.im = source;
15            this.lock = lock;
16        }
17
18        public List compute(VFSDescription vfsD)
19                throws FileSystemException, CleverException, Exception {
20            String response = "";
21            String respLock;
22 /*MOD*/    params = new ArrayList();
23 /****/    VirtualFileSystem vfs = getVFS(vfsD);
24            UUID id = UUID.randomUUID();
25            try {
26                if (!file_s.exists()) {
27                    params.add(response);
28                } else {
29 /****/               FileContent content = getContent();
```

```
30 /*MOD*/            im.logger.debug("element searched : " + file_s.getName().getURI());
31 /*MOD*/            ResultSet rs = im.sqlite.retrieveElementsInMap(file_s
32                          .getName().getURI());
33                    if ((!rs.isAfterLast() && !rs.isBeforeFirst())) {
34                        //-----------------------------------------//
35                        if (file_s.getType().equals(FileType.FOLDER)) {
36                            response = "" + this.localRepository + id;
37 /*****/                    setNewParams(lock, response);
38                        } else {
39 /*MOD*/                    response = im.localRepository + id + "."
40                                  + file_s.getName().getExtension();
41 /*****/                    setNewParams(lock, response, content);
42                        }
43                        FileSystemManager mgr = VFS.getManager();
44                        FileObject file_d = mgr.resolveFile(response);
45                        vfs.cp(file_s, file_d);
46                        if (file_s.getType().equals(FileType.FOLDER)) {
47 /*MOD*/                    im.logger.debug("Image1");
48 /*MOD*/                    im.sqlite.insertElementAtMap(file_s.getName()
49                                  .getURI(), response, "", lastMod, LockFile
50                                  .getLockType(lock));
51                        } else {
52 /*MOD*/                    im.logger.debug("Image2");
53 /*MOD*/                    im.sqlite.insertElementAtMap(file_s.getName()
54                                  .getURI(), response,
55                                  new Long(content.getSize()).toString(),
56                                  lastMod, LockFile.getLockType(lock));
57                        }
58                        //-----------------------------------------//
59                    } else {
60                        while (!rs.isAfterLast()) {
61                            LockFile l = new LockFile();
62                            respLock = l.checkLock(
63                                  LockFile.getLockEnumType(rs.getString("lock")),
64                                  lock);
65                            if ("SI".equals(respLock)) {
66 /*****/                        setParamsSiLock(lock, rs);
67                                return params;
68                            }
69                            rs.next();
70                        }
71                        //-----------------------------------------//
72                        if (file_s.getType().equals(FileType.FOLDER)) {
73 /*MOD*/                    response = im.localRepository + id;
74 /*MOD*/                    im.sqlite.insertElementAtMap(file_s.getName()
75                                  .getURI(), response, "", lastMod, LockFile
76                                  .getLockType(lock));
77 /*****/                    setInsetParam(lock, response);
78                        } else {
79 /*MOD*/                    response = im.localRepository + id + "."
```

```
80                                  + file_s.getName().getExtension();
81  /*MOD*/                    im.logger.debug("Image4");
82  /*MOD*/                    im.sqlite.insertElementAtMap(file_s.getName()
83                                  .getURI(), response,
84                                  new Long(content.getSize()).toString(),
85                                  lastMod, LockFile.getLockType(lock));
86  /*++++*/                   setParamSizedInsert(lock, response, content);
87                      }
88                      //-------------------------------------------//
89                      FileSystemManager mgr = VFS.getManager();
90                      FileObject file_d = mgr.resolveFile(response);
91                      vfs.cp(file_s, file_d);
92                      return params;
93                  }
94              }
95          } catch (Exception e) {
96  /*MOD*/       im.logger.error(e.getLocalizedMessage());
97          }
98          return params;
99      }
100
101     private void setParamSizedInsert(LockFile.lockMode lock, String response,
102             FileContent content) {
103         params.add("insert");
104         params.add(response);
105         params.add(lastMod);
106         params.add(content.getSize());
107         params.add(lock);
108     }
109
110     private void setInsetParam(LockFile.lockMode lock, String response) {
111         params.add("insert");
112         params.add(response);
113         params.add(lastMod);
114         params.add("");
115         params.add(lock);
116     }
117
118     private void setParamsSiLock(LockFile.lockMode lock, ResultSet rs)
119             throws SQLException {
120         LockFile.lockMode b1l = LockFile.getLockEnumType(rs
121                 .getString("lock"));
122         if (b1l.ordinal() >= lock.ordinal()) {
123             params.add("notUpdate");
124             params.add(rs.getString("response"));
125         } else {
126             params.add("update");
127             params.add(rs.getString("response"));
128             params.add("");
129             params.add("");
```

```
130                 params.add(lock);
131 /*MOD*/          im.sqlite.updateElementInMap("response='"
132                         + rs.getString("response") + "'",
133                         "lock='" + LockFile.getLockType(lock)
134                         + "'");
135 
136             }
137         }
138 
139         private void setNewParams(LockFile.lockMode lock, String response,
140                 FileContent content) {
141             params.add("new");
142             params.add(response);
143             params.add(lastMod);
144             params.add(content.getSize());
145             params.add(lock);
146         }
147 
148         private void setNewParams(LockFile.lockMode lock, String response) {
149             params.add("new");
150             params.add(response);
151             params.add(lastMod);
152             params.add("");
153             params.add(lock);
154         }
155 
156         private FileContent getContent() {
157             FileContent content = file_s.getContent();
158 /*MOD*/     lastMod = "";
159             if (file_s.getType().equals(FileType.FILE)) {
160                 DateFormat dateFormat = DateFormat.getDateTimeInstance(
161                         DateFormat.MEDIUM, DateFormat.MEDIUM);
162                 lastMod = dateFormat.format(new Date(content
163                         .getLastModifiedTime()));
164             }
165             return content;
166         }
167 
168         private VirtualFileSystem getVFS(VFSDescription vfsD) {
169             VirtualFileSystem vfs = new VirtualFileSystem();
170             vfs.setURI(vfsD);
171 /*MOD*/     file_s = vfs.resolver(vfsD, vfs.getURI(), vfsD.getPath1());
172             return vfs;
173         }
174         [...]
175     }
```

A.3 The *Method Object* after other two *Extract Methods* on `compute()`

```
1    public class StorageIM {
2        private final ImageManager im;
3        //private Logger logger; --> im.logger
4        //private SQLite sqlite; --> im.sqlite
5        //private String localRepository; --> im.localRepository
6
7        // Fields selected by the tool
8        private LockFile.lockMode lock;
9        private String lastMod;
10       private List params;
11       private FileObject file_s;
12
13       public StorageIM(ImageManager source, LockFile.lockMode lock) {
14           this.im = source;
15           this.lock = lock;
16       }
17
18       public List compute(VFSDescription vfsD)
19               throws FileSystemException, CleverException, Exception {
20           String response = "";
21           String respLock;
22           params = new ArrayList();
23           VirtualFileSystem vfs = getVFS(vfsD);
24           UUID id = UUID.randomUUID();
25           try {
26               if (!file_s.exists()) {
27                   params.add(response);
28               } else {
29                   FileContent content = getContent();
30                   im.logger.debug("element searched : " + file_s.getName().getURI());
31                   ResultSet rs = im.sqlite.retrieveElementsInMap(file_s
32                           .getName().getURI());
33                   if ((!rs.isAfterLast() && !rs.isBeforeFirst())) {
34                       //-------------------------------------------//
35/*****/                 processInternalElement(lock, vfs, id, content);
36                       //-------------------------------------------//
37                   } else {
38                       while (!rs.isAfterLast()) {
39                           LockFile l = new LockFile();
40                           respLock = l.checkLock(
41                                   LockFile.getLockEnumType(rs.getString("lock")),
42                                   lock);
43                           if ("SI".equals(respLock)) {
44                               setParamsSiLock(lock, rs);
45                               return params;
```

```
                        }
                        rs.next();
                    }
                    //-------------------------------------------//
/*++++*/              response = prepareResponse(lock, id, content);
                    //-------------------------------------------//
                    FileSystemManager mgr = VFS.getManager();
                    FileObject file_d = mgr.resolveFile(response);
                    vfs.cp(file_s, file_d);
                    return params;
                }
            }
        } catch (Exception e) {
            im.logger.error(e.getLocalizedMessage());
        }
        return params;
    }

    private String prepareResponse(LockFile.lockMode lock, UUID id,
            FileContent content) {
        String response;
        if (file_s.getType().equals(FileType.FOLDER)) {
            response = im.localRepository + id;
            im.sqlite.insertElementAtMap(file_s.getName()
                    .getURI(), response, "", lastMod, LockFile
                    .getLockType(lock));
            setInsetParam(lock, response);
        } else {
            response = im.localRepository + id + "."
                    + file_s.getName().getExtension();
            im.logger.debug("Image4");
            im.sqlite.insertElementAtMap(file_s.getName()
                    .getURI(), response,
                    new Long(content.getSize()).toString(),
                    lastMod, LockFile.getLockType(lock));
            setParamSizedInsert(lock, response, content);
        }
        return response;
    }

    private void processInternalElement(LockFile.lockMode lock,
            VirtualFileSystem vfs, UUID id, FileContent content) {
        String response;
        if (file_s.getType().equals(FileType.FOLDER)) {
            response = "" + im.localRepository + id;
            setNewParams(lock, response);
        } else {
            response = im.localRepository + id + "."
                    + file_s.getName().getExtension();
            setNewParams(lock, response, content);
```

```
96              }
97              FileSystemManager mgr = VFS.getManager();
98              FileObject file_d = mgr.resolveFile(response);
99              vfs.cp(file_s, file_d);
100             if (file_s.getType().equals(FileType.FOLDER)) {
101                 im.logger.debug("Image1");
102                 im.sqlite.insertElementAtMap(file_s.getName()
103                         .getURI(), response, "", lastMod, LockFile
104                         .getLockType(lock));
105             } else {
106                 im.logger.debug("Image2");
107                 im.sqlite.insertElementAtMap(file_s.getName()
108                         .getURI(), response,
109                         new Long(content.getSize()).toString(),
110                         lastMod, LockFile.getLockType(lock));
111             }
112         }
113
114         [...]
115     }
```

Bibliography

[1] A. Abadi, R. Ettinger, and Y. A. Feldman. Fine slicing. In *Fundamental Approaches to Software Engineering: 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24-April 1, 2012, Proceedings*, page 471. Springer, 2012.

[2] A. Abran, K. T. Al-Sarayreh, and J. J. Cuadrado-Gallego. A standards-based reference framework for system portability requirements. *Computer Standards & Interfaces*, 35(4):380–395, 2013.

[3] B. Adams, Z. M. Jiang, and A. E. Hassan. Identifying crosscutting concerns using historical code changes. In *Proceedings of International Conference on Software Engineering*, pages 305–314. IEEE, 2010.

[4] V. Arena, V. Catania, G. La Torre, S. Monteleone, and F. Ricciato. Securedroid: An android security framework extension for context-aware policy enforcement. In *Proc. of IEEE International Conference on Privacy and Security in Mobile Systems (PRISMS)*, pages 1–8, 2013.

[5] S. Arzt et al. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of PLDI*. ACM, 2014.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proc. of ACM SIGPLAN Notices*, volume 49, pages 259–269, 2014.

[7] G. Ascia, V. Catania, R. Di Natale, A. Fornaia, M. Mongiovì, S. Monteleone, G. Pappalardo, and E. Tramontana. Making android apps data-leak-safe by data flow analysis and code injection. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2016 IEEE 25th International Conference on*, pages 205–210. IEEE, 2016.

[8] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of OSDI*, 2010.

[9] M. Backes et al. Appguard, fine-grained policy enforcement for untrusted android applications. In *Proceedings of DPM workshop*, 2013.

[10] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. In *Proceedings of OOPSLA*, volume 43, pages 543–562. ACM, 2008.

[11] F. Banno, D. Marletta, G. Pappalardo, and E. Tramontana. Tackling consistency issues for runtime updating distributed systems. In *Proceedings of IPDPSW*. IEEE, 2010.

[12] K. Beck. *Smalltalk best practice patterns*. Prentice-Hall, 1997.

[13] K. Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[14] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82, 1994.

[15] E. Bosman, A. Slowinska, and H. Bos. Minemu: the world's fastest taint tracker. In *Proc. of RAID*. Springer, 2011.

[16] P. Bottoni, F. Parisi-Presicce, and G. Taentzer. Coordinated distributed diagram transformation for software evolution11partially supported by the ec under research and training network segravis. *Electronic Notes in Theoretical Computer Science*, 72(4):59–70, 2003.

[17] S. Breu and T. Zimmermann. Mining aspects from version history. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 221–230. IEEE, 2006.

[18] L. C. Briand, J. W. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[19] L. C. Briand, J. W. Daly, and J. K. Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on software Engineering*, 25(1):91–121, 1999.

[20] G. Cabri, L. Ferrari, and L. Leonardi. Enabling mobile agents to dynamically assume roles. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 56–60. ACM, 2003.

[21] A. Calvagna, A. Fornaia, and E. Tramontana. Assessing the correctness of jvm implementations. In *WETICE Conference (WETICE), 2014 IEEE 23rd International*, pages 390–395. IEEE, 2014.

[22] A. Calvagna, A. Fornaia, and E. Tramontana. Combinatorial interaction testing of a java card static verifier. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, pages 84–87. IEEE, 2014.

[23] A. Calvagna, A. Fornaia, and E. Tramontana. Random versus combinatorial effectiveness in software conformance testing: a case study. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1797–1802. ACM, 2015.

[24] A. Calvagna and A. Gargantini. T-wise combinatorial interaction test suites construction based on coverage inheritance. *Software Testing, Verification and Reliability*, 22(7):507–526, 2012.

[25] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of CCS*. ACM, 2008.

[26] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[27] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proc. of International Conference on Mobile Systems, Applications, and Services*, MobiSys, pages 239–252. ACM, 2011.

[28] E. Chin and D. Wagner. Efficient character-level taint tracking for java. In *Proceedings of workshop on Secure web services*, pages 3–12. ACM, 2009.

[29] S.-S. Choi, S.-H. Cha, and C. C. Tappert. A survey of binary similarity and distance measures. *Journal of Systemics, Cybernetics and Informatics*, 8(1):43–48, 2010.

[30] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of USENIX Security Symposium*, pages 321–336, 2004.

[31] A. Cockburn. *Agile software development*, volume 177. Addison-Wesley Boston, 2002.

[32] K. Cotterell, I. Welch, and A. Chen. An android security policy enforcement tool. *International Journal of Electronics and Telecommunications*, 61(4):311, 2015.

[33] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of MICRO*, pages 221–232. IEEE, 2004.

[34] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *Proceedings of NDSS*, 2011.

[35] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Trans. on Software Engineering*, 29(3):210–224, 2003.

[36] A. D. Eisenberg and K. De Volder. Dynamic feature traces: Finding features in unfamiliar code. In *Proceedings of International Conference on Software Maintenance (ICSM)*, pages 337–346. IEEE, 2005.

[37] W. Enck et al. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of OSDI*, 2010.

[38] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[39] R. Ettinger. Refactoring via program slicing and sliding. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 505–506. IEEE, 2007.

[40] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[41] A. Fornaia, C. Napoli, G. Pappalardo, and E. Tramontana. An ao system for oo-gpu programming. In *Proc. of the 16th Workshop From Object to Agents (WOA)*, volume 1382, Napoli, Italy, June 2015. CEUR-WS.

[42] A. Fornaia and E. Tramontana. Deduct: a data dependence based concern tagger for modularity analysis. In *Computers, Software, and Applications (COMPSAC), 2017 IEEE 41th International Conference on*. IEEE, 2017.

[43] A. Fornaia and E. Tramontana. Is my code easy to port? using taint analysis to evaluate and assist code portability. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2017 IEEE 26th International Conference on*, pages 269–274. IEEE, 2017.

[44] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[45] E. Gamma, R. Helm, R. Johnson, and R. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, MA, 1994.

[46] A. Gargantini, E. Riccobene, and S. Rinzivillo. Using spin to generate tests from asm specifications. In *Abstract State Machines 2003*, pages 263–277. Springer, 2003.

[47] M. Gawinecki, G. Cabri, M. Paprzycki, and M. Ganzha. Wscolab: Structured collaborative tagging for web service matchmaking. In *WEBIST (1)*, pages 70–77, 2010.

[48] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Proceedings of TRUST*. Springer, 2012.

[49] R. Giunta, G. Pappalardo, and E. Tramontana. Aspects and annotations for controlling the roles application classes play for design patterns. In *Proc. of APSEC*. IEEE, 2011.

[50] R. Giunta, G. Pappalardo, and E. Tramontana. AODP: refactoring code to provide advanced aspect-oriented modularization of design patterns. In *Proceedings of SAC*, pages 1243–1250. ACM, 2012.

[51] R. Giunta, G. Pappalardo, and E. Tramontana. Superimposing roles for design patterns into application classes by means of aspects. In *Proceedings of ACM Symposium on Applied Computing (SAC)*, pages 1866–1868, March 2012.

[52] M. L. Goodstein et al. Butterfly analysis: Adapting dataflow analysis to dynamic parallel monitoring. In *Proceedings of SIGARCH Computer Architecture News*, volume 38. ACM, 2010.

[53] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: Scalable and accurate zero-day android malware detection. In *Proc. of ACM International Conference on Mobile Systems, Applications, and Services*, MobiSys, pages 281–294, 2012.

[54] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable javascript. In *Proceedings of ISSTA*. ACM, 2011.

[55] B. Henderson-Sellers. *Software metrics*. Prentice Hall, Hemel Hempstaed, UK, 1996.

[56] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE transactions on Software Engineering*, (5):510–518, 1981.

[57] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. 1995.

[58] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *Proc. of SIGOPS/EuroSys*, volume 40. ACM, 2006.

[59] S. Holavanalli et al. Flow permissions for android. In *Proceedings of ASE*. IEEE/ACM, Nov. 2013.

[60] K. Jee et al. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *Proc. of NDSS*, 2012.

[61] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of NDSS*, 2011.

[62] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: practical dynamic data flow tracking for commodity systems. In *Proceedings of SIGPLAN Notices/VEE*, volume 47. ACM, 2012.

[63] U. Khedker, A. Sanyal, and B. Sathe. *Data flow analysis: theory and practice*. CRC Press, 2009.

[64] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP*, pages 220–242. Springer, 1997.

[65] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[66] Y. Lee, B. Liang, S. Wu, and F. Wang. Measuring the coupling and cohesion of an object-oriented program based on information flow. In *Proc. International Conference on Software Quality, Maribor, Slovenia*, pages 81–90, 1995.

[67] J. Lenhard and G. Wirtz. Portability of executable service-oriented processes: metrics and validation. *Service Oriented Computing and Applications*, 10(4):391–411, 2016.

[68] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proc. of Working Conf. on Reverse Engineering*, pages 214–223. IEEE, 2004.

[69] M. Marin, A. V. Deursen, and L. Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(1):3, 2007.

[70] K. Maruyama. Automated method-extraction refactoring by using block-based slicing. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 31–40. ACM, 2001.

[71] M. Mattsson, H. Grahn, and F. Mårtensson. Software architecture evaluation methods for performance, maintainability, testability, and portability. In *Proc. of Conf. on Quality of Software Architectures*, 2006.

[72] M. Mongiovì, G. Giannone, A. Fornaia, G. Pappalardo, and E. Tramontana. Combining static and dynamic data flow analysis: a hybrid approach for detecting data leaks in java applications. In *Proc. of Symposium on Applied Computing (SAC)*, pages 1573–1579. ACM, 2015.

[73] J. D. Mooney. Strategies for supporting application portability. *Computer*, 23(11):59–70, 1990.

[74] J. D. Mooney. Issues in the specification and measurement of software portability. In *Proc. of Conf. on Soft. Engin.*, 1993.

[75] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM TOSEM*, 9(4), 2000.

[76] C. Napoli, G. Pappalardo, and E. Tramontana. Using modularity metrics to assist move method refactoring of large systems. In *Complex, Intelligent, and Software Intensive Systems (CISIS), 2013 Seventh International Conference on*, pages 529–534. IEEE, 2013.

[77] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of NDSS*, 2005.

[78] C. Nunes, A. Garcia, E. Figueiredo, and C. Lucena. Revealing mistakes in concern mapping tasks: an experimental evaluation. In *Proceedings of CSMR*, pages 101–110. IEEE, 2011.

[79] G. Pappalardo and E. Tramontana. Suggesting extract class refactoring opportunities by measuring strength of method interactions. In *Software Engineering Conference (APSEC), 2013 20th Asia-Pacific*, volume 2, pages 105–110. IEEE, 2013.

[80] M. Pezzé and M. Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2008.

[81] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In *Proceedings of SIGOPS/EuroSys*, volume 40. ACM, 2006.

[82] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Proceedings of ICPC*, pages 37–48. IEEE, 2007.

[83] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of MICRO*, pages 135–148. IEEE, 2006.

[84] M. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of International Conference on Software Engineering*, pages 406–416. IEEE, 2002.

[85] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *Transactions on Software Engineering and Methodology (TOSEM)*, 16(1):3, 2007.

[86] K. Schwaber and M. Beedle. *Agile software development with Scrum*, volume 1. Prentice Hall Upper Saddle River, 2002.

[87] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proc. of Security and privacy (SP)*. IEEE, 2010.

[88] J. Schtte, D. Titze, and J. M. D. Fuentes. Appcaulk: Data leak prevention by injecting targeted taint tracking into android apps. In *Proc. IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 370–379, Sept 2014.

[89] G. Shu, B. Sun, T. A. D. Henderson, and A. Podgurski. Javapdg: A new platform for program dependence analysis. In *Proceedings of ICST*, pages 408–415. IEEE, 2013.

[90] D. Silva, R. Terra, and M. T. Valente. Recommending automated extract method refactorings. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 146–156. ACM, 2014.

[91] F. Simon, F. Steinbruckner, and C. Lewerentz. Metrics based refactoring. In *Software Maintenance and Reengineering, 2001. Fifth European Conference on*, pages 30–38. IEEE, 2001.

[92] I. Sommerville. *Software Engineering*. Addison Wesley, 2004.

[93] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *Proceedings of ACM Symposium on Applied Computing (SAC)*, pages 1808–1815, 2013.

[94] M.-A. Storey, L.-T. Cheng, I. Bull, and P. Rigby. Shared waypoints and social tagging to support collaboration in software development. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 195–198. ACM, 2006.

[95] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Sigplan Notices*, volume 39, pages 85–96. ACM, 2004.

[96] F. A. Talib, D. Giannacopoulos, and A. Abran. Designing a measurement method for the portability non-functional requirement. In *Proc. of IWSM-MENSURA*. IEEE, 2013.

[97] E. Tramontana. Automatically characterising components with concerns and reducing tangling. In *Computer Software and Applications Conference Workshops (COMPSACW), 2013 IEEE 37th Annual*, pages 499–504. IEEE, 2013.

[98] M. Trifu. Using dataflow information for concern identification in object-oriented software systems. In *Proceedings of CSMR*, pages 193–202. IEEE, 2008.

[99] O. Tripp et al. Andromeda: Accurate and scalable security analysis of web applications. In *Proceedings of FASE*. Springer, 2013.

[100] N. Tsantalis and A. Chatzigeorgiou. Identification of extract method refactoring opportunities. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 119–128. IEEE, 2009.

[101] N. Tsantalis and A. Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782, 2011.

[102] F. Tusa, M. Paone, M. Villari, and A. Puliafito. Clever: A cloud-enabled virtual environment. In *Proceedings of ISCC*, pages 477–482. IEEE, 2010.

[103] R.-G. Urma and A. Mycroft. Source-code queries with graph databaseswith application to programming language usage and evolution. *Science of Computer Programming*, 97:127–134, 2015.

[104] N. Vachharajani et al. Rifle: An architectural framework for user-centric information-flow security. In *Proceedings of MICRO*. IEEE, 2004.

[105] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[106] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. 1998.

[107] A. Van Deursen, M. Marin, and L. Moonen. Aspect mining and refactoring. In *Proceedings of the 1st International Workshop on Refactoring: Achievements, Challenges, Effects (REFACE), with WCRE*, pages 11–21, 2003.

[108] G. Venkataramani, I. Doudalis, D. Solihin, and M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Proc. of HPCA*. IEEE, 2008.

[109] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proc. of NDSS*, 2007.

[110] H. Washizaki, H. Yamamoto, and Y. Fukazawa. A metrics suite for measuring reusability of software components. In *Proc. of Software Metrics Symposium*. IEEE, 2003.

[111] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Profiledroid: Multi-layer profiling of android applications. In *Proceedings of ACM International Conference on Mobile Computing and Networking (Mobicom)*, pages 137–148, 2012.

[112] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis: Android Malware Under the Magnifying Glass. Technical Report TRISECLAB-0414-001, Vienna Univ. of Techn., 2014.

[113] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[114] I. S. Welch and R. J. Stroud. Re-engineering security as a crosscutting concern. *The Computer Journal*, 46(5):578–589, 2003.

[115] R. Xu, H. Saïdi, and R. Anderson. Aurasium: practical policy enforcement for android applications. In *Proceedings of USENIX Security symposium*, 2012.

[116] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proc. of USENIX Security symposium*, 2006.

[117] L. K. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proc. of USENIX Security Symposium*, 2012.

[118] L. Yang, H. Liu, and Z. Niu. Identifying fragments to be extracted from long methods. In *Software Engineering Conference, 2009. APSEC'09. Asia-Pacific*, pages 43–49. IEEE, 2009.

[119] M. Young and M. Pezze. *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, 2005.

[120] R. Zhang, S. Huang, Z. Qi, and H. Guan. Static program analysis assisted dynamic taint tracking for software vulnerability discovery. *Computers & Mathematics with Applications*, 63(2):469–480, 2012.

[121] J. Zhao. Dependence analysis of java bytecode. In *Computer Software and Applications Conference, 2000. COMPSAC 2000. The 24th Annual International*, pages 486–491. IEEE, 2000.

[122] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *Proceedings of TRUST*. Springer, 2011.

[123] D. Y. Zhu et al. Tainteraser: Protecting sensitive data leaks using application-level taint tracking. *SIGOPS Operating Systems Review*, 45(1):142–154, 2011.