Interprocedural Graph-based Object Usage Model Generation for Detecting Anomalous Usage of Cryptographic APIs

Generierung interprozeduraler, graph-basierter Objektverwendungsmodelle zur Detektierung fehlerhafter Verwendungen kryptographischer APIs

Manuel Fabian Benz (Master of Science Informatik, Master of Science IT-Sicherheit)

Technische Universität Darmstadt
Department of Computer Science
Software Technology Group

Examiner: Prof. Dr.-Ing. Mira Mezini
Supervisor: Sven Amann, M.Sc. and Dr. Karim Ali, Ph.D.

Submission Date: 20.10.2016

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Masterarbeit selbstständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel verwendet. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Darmstadt, den 20.10.2016

Manuel Fabian Benz

Abstract

English

Security of modern applications is oftentimes flawed due to incorrect usage of cryptographic APIs. Researchers have shown that such incorrect usages can automatically be identified using graph-based approaches to detect API usage anomalies. However, these approaches suffer from large amounts of false positives. We have conducted experiments that aim at detecting such API usage anomalies in Android applications utilizing the Java Cryptography Extension (JCE). After manual investigation, we were able to identify 70% of the detected anomalies as false positives caused by the intraprocedural nature of the graph model. This thesis proposes an approach for generating interprocedural graph models of library usage by inlining method calls on the graph level. For this purpose, an augmentation of the previous model that carries necessary information for the inlining process is presented. Furthermore, several heuristics which allow for fine-grained selection of methods that should be inlined are introduced and evaluated. Our experiments on 50 Android applications utilizing the JCE show that the interprocedural model yields a reduction of those false positives by up to 42.86% with an overall reduction of detected anomalies by 30.37%.

Deutsch

Eine häufige Ursache für mangelnde Sicherheit in modernen Anwendungen ist der Missbrauch von kryptographischen APIs. In der Vergangenheit konnte gezeigt werden, dass Graph-basierte Verfahren gut geeignet sind, um die Verwendung solcher APIs zu modellieren und dadurch Fehlverwendungen automatisch zu erkennen. Diese Verfahren weisen jedoch häufig eine große Anzahl an False Positives auf. Um deren Ursache zu identifizieren, führten wir eine Reihe von Experimenten mit Android-Anwendungen, die die Java Cryptography Extension (JCE) verwenden, durch. Dabei wurde deutlich, dass mit ca. 70% der größte Teil dieser Missklassifizierungen durch die ausschließlich intraprozedurale Modellierung der Graphen entsteht. In dieser Arbeit wird ein Ansatz präsentiert, der es ermöglicht, stattdessen eine interprozedurale Repräsentationen dieser Graphen durch das Inlinen von Methoden auf der Graphen-Ebene zu realisieren. Dafür wird eine Erweiterung des vorherigen Modells eingeführt, die die Graphen um die für das Inlinen nötigen Informationen ergänzt. Des Weiteren werden verschiedene Heuristiken vorgestellt und evaluiert, die es ermöglichen, nach spezifischen Kriterien zu entscheiden, ob eine Methode geinlinet werden soll. In unseren Experimenten mit 50 Android-Anwendungen, die die Java Cryptgraphy Extension benutzen, konnten wir zeigen, dass es mit Hilfe des interprozeduralen Modells möglich ist, die Anzahl solcher False Positives, die tatsächlich auf die intraprozedurale Modellierung zurückzuführen sind, um 42.86% zu verringern und dabei auch die Gesamtmenge der gemeldeten Fehlverwendungen um 30.37% zu reduzieren.

Table of Contents

1. Introduction 1
1.1. Research Questions . . . 2
1.2. Thesis Structure . . . 3

2. Related Work 4
2.1. API Misuse . . . 4
2.2. Pattern-based Anomaly Detection . . . 4

3. Overview 5
3.1. GROUMs . . . 5
3.2. Patterns . . . 7
3.3. Anomaly Detection . . . 9
3.3.1. Intra- vs Interprocedural GROUMs . . . 9

4. Interprocedural GROUMs 11
4.1. Inling GROUMs vs Methods . . . 11
4.2. Extended GROUMs . . . 12
4.2.1. Advanced-Edges . . . 12
4.3. Inliner . . . 15
4.3.1. Inline Heuristics . . . 18
4.3.2. Inline Tree . . . 20
4.4. Virtual Callsites . . . 20

5. Implementation 23
5.1. Extended GROUMs . . . 23
5.1.1. Dependency Manager . . . 23
5.1.2. GROUM Input/Output . . . 24
5.1.3. GROUM Optimizations . . . 25
5.2. Inlining Infrastructure . . . 25
5.2.1. InlineBuilder . . . 25
5.2.2. GroumInliner . . . 27
5.2.3. LookupProvider . . . 28
5.2.4. InlineHeuristic . . . 30
5.3. Call-Graphs . . . 30
5.4. Test Cases . . . 30
5.4.1. Interprocedural Pattern Usage Filter . . . 31

6. Evaluation 32
6.1. Setup . . . 32
6.2. Constructed Tests . . . 32

6.3. Real World Tests 36

7. Conclusion **42**

List of Figures **45**

References **46**

A. Appendix **49**

1 Introduction

Previous research shows that misuse of cryptographic APIs is a common problem and oftentimes the cause for security vulnerabilities in software products [EBFK13, ANA+15, GIJ+12, LCWZ14, ANN+16]. Correct usage of such APIs generally includes the proper invocation of API methods, i.e., passing valid and consistent arguments [EBFK13, ANA+15], as well as proper ordering of API method invocations [NKMB16]. When using the Java Cryptography Extension (JCE) API, for example, it is essential to correctly initialize the *Cipher* object to use a (properly configured) secure cryptographic algorithm. Listing 1.1 shows such a correct initialization for an AES algorithm in encryption mode. Note that a lot of JCE providers will default to the Electronic Code Book Mode (ECB), which is considered insecure, when invoking *Cipher.getInstance(...)* without the specification of any block cipher mode of operation [EBFK13, ANA+15].

Nguyen et al. [NNP+09] presented Graph-based Object Usage Models (GROUMs) as a means to model invocations on Java objects, including their order, control and data dependencies. Figure 1.1 shows the GROUM as generated by their approach for the example in Listing 1.1. They employ frequent subgraph mining [EKKB10] to learn usage patterns from example code and use these to detect anomalous usage of Java APIs by matching them against their GROUMs. We adapted their approach and implemented our own toolchain (the *Cryptominer*) for GROUM generation, pattern mining and anomaly detection, specialized (but not exclusively) on cryptographic APIs.

```
public byte[] enryptAES(SecretKey secretKey){
  // Passing just "AES" here would default to ECB mode
  Cipher cipher = Cipher.getInstance("AES/CBC");
  cipher.init(Cipher.ENCRYPT_MODE, secretKey);
  byte[] ctBytes = "ClearText".getBytes();
  return cipher.doFinal(ctBytes);
}
```

Listing 1.1: AES encryption in CBC mode.

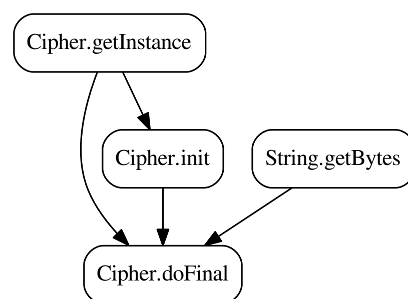

Figure 1.1.: GROUM for AES encryption.

In our experiments with the *Cryptominer* on 50 Android applications using the JCE, we found that a large fraction of detected anomalies are false positives, i.e., a GROUM only matches parts of a pattern indicating a violation while the target GROUM actually models correct usage. Others experienced similar results [NNP+09, EH07, Han05]. After manual inspection of 65 of the 208 detected anomalies, we were able to identify that 70% of those are caused by the separation of object usages over multiple GROUMs, i.e., parts of a pattern are present in one GROUM while the remaining parts are split over others. This happens when a programmer outsources parts of API usage in helper methods (subroutines) to encapsulate code for use in multiple locations or to improve readability. This has been observed by other before and is a general shortcoming of intraprocedural mining methods [LZ05, Lin07]. Nguyen et al. [NNN16] also showed that while there is a lot of semantic repetitiveness in code over a multitude of

projects, it is oftentimes split differently over several subroutines and the subroutines are used repeatedly in different locations of the code. When thinking in terms of object usages, this means that there are a lot of semantically equivalent usages split differently among several methods. This greatly supports our findings and directly motivates the need for an interprocedural approach for GROUM generation to lower the high false positive rate during anomaly detection. In this thesis, we present such an approach that is able to systematically build GROUMs of methods, including the called subroutines and still keep the semantics of the original code by inlining methods on the graph level. For this purpose, we introduce an extension of the GROUM model that has a more sophisticated way of modeling data dependencies inside and between methods and, thus, not only allows for a graph-based inlining, but also renders the model more precise in its representation of the original code. Furthermore, we present several inline heuristics that allow for a use-case-based selection of callees for the inlining. For example, the inlining process can be restricted from inlining API internals by explicitly avoiding such with a white-/blacklist approach. As another example, the inlining might be restricted to just a few levels of the call-hierarchy to restrict the size of the interprocedural GROUMs when using those for the pattern mining.

1.1 Research Questions

In the following the main research questions of the thesis are presented. Theses will be discussed throughout the thesis and answered in Chapter 7.

- **Is interprocedural GROUM generation able to reduce false positives caused by the separation of object usages over multiple methods?**
  Anomaly detection using the intraprocedural model suffers from a large amount of false positives caused by the separation of object usages among several methods. Proper techniques for interprocedural GROUM generation have to be designed, implemented and evaluated for their potential of reducing such false positives.

- **How does interprocedural GROUM generation impact the detection of true positives and other false positives?**
  A method might contain true positives or false positives that are not caused by the separation of object usages. When used in multiple other methods, building interprocedural GROUMs for those might lead to redundant detection of anomalies. It has to be evaluated how the reduction of false positives and the increase (through redundancy) trade off.

- **What are possible strategies for interprocedural GROUM generation? How can the generation of such interprocedural GROUMs avoid including API internals?**
  The original GROUM model captures library usages, such that the anomaly detector can learn usage patterns. If we would transitively analyze every callee, all method invocations would disappear from the model and the detector would not learn from any library-usage examples. Furthermore, additional false positives might appear in the anomaly detection step since GROUMs would no longer model library usages, like the patterns, but rather their internals. Therefore, an approach that captures usages for the interesting APIs, but transitively analyzes all others is needed. The tradeoff between expanding and reducing the scope of the interprocedural analysis has to be evaluated.

1.2 Thesis Structure

The remainder of this document is organized as follows. Chapter 2 gives an overview of related work. Chapter 3 presents fundamental background knowledge and basic terminology as a basis for the thesis. Chapter 4 provides detailed information about our approach of generating interprocedural GROUMs on a conceptual level. Chapter 5 explains the implementation of this approach. Chapter 6 shows and discusses the results of the conducted experiments. Finally, in Chapter 7 we summarize the main contribution of this thesis and discuss its limitations, as well as possible future work.

2 Related Work

In this section a brief overview of related work to the thesis' topic is given.

2.1 API Misuse

The general need for analyzing Android applications for the misuse of cryptographic APIs originates from several studies made by others. Egele et al. [EBFK13] propose an automated approach that employs static program slicing to identify flows between cryptographic data, such as keys and initialization vectors, and the cryptographic operations to identify flaws in the usage of cryptographic APIs. They propose six rules that, when violated, indicate flaws in data flow between those elements. In their experiments on 11.748 Android applications, they found that 10.327 (88%) of those applications have at least one flaw in their usage of cryptographic APIs. Based on this and other publications, Nadi et al. [NKMB16] empirically investigated the reasons why developers struggle with the usage of those APIs. Results from their survey of 48 developers show that while developers are generally confident in selecting the correct cryptographic means for their tasks, such as encryption, signing, they often struggle with applying the concrete cryptographic algorithms correctly. Furthermore, they identified that developers oftentimes find the APIs to be too low-level and modular, i.e., allow for too much configuration freedom and, thus, for introducing misconfigurations. For those reasons we want to support developers in the usage of the APIs and present *Cryptominer* as a means to automatically detect and report misuses of such.

2.2 Pattern-based Anomaly Detection

Wasylkowski et al. [WZL07] present *JADET*, an approach for mining object usage models in Java methods as sequences of method calls. The approach is constraint to modeling object usage examples of a single object, i.e., each type of object needs its own representation. They use means of frequent itemset mining [Han05] to extract patterns of frequent object usages from their model and use those patterns to detect anomalous API usages in code. In their case-study on the *ASPECTJ*[1] library, 790 violations were detected from which they classified 96 (12.2%) true positives and the remaining 694 (87.8%) as false positives. Building on the insights of Wasylkowsky et al., Nguyen et al. [NNP+09] presented so-called Graph-based Object Usage Models (GROUMs) that allow for modeling multiple object usages in Java methods as directed acyclic graphs. They employ frequent subgraph mining methods to extract patterns of frequent object usages from GROUM sets of various applications. With the help of those patterns, they are able to find anomalies in the usage of certain APIs. In their experiments on the *ASPECTJ* library, they detected 244 anomalies from which they manually checked 15 which led to 3 true positives (20%) and 12 false positives (80%). The minimal observed false positive rate in their experiments is 62.5% for the *Fluid project*. With *Cryptominer*, we build upon their findings and adapt their GROUM model to detect cryptographic API misuses in Java code to eventually guide the developer through the usage of such APIs. In our experiments on Android applications using the JCE, we observed that 70% of anomalies are caused by the intraprocedural nature of the model, which motivated the need for an interprocedural approach.

[1] `http://www.eclipse.org/aspectj/`

3 Overview

In this chapter, the structure of GROUMs is explained more thoroughly. Furthermore, basic terminology is provided, as well as other fundamentals, which are important when further processing through the thesis.

3.1 GROUMs

To understand the problems and procedures of generating interprocedural GROUMs, it is necessary to first build up a general understanding of their structure. We present our own implementation of the GROUM model based on the original by Nguyen et al [NNP+09]. Contrary to their implementation which uses abstract syntax trees (ASTs), we employ the *Soot*[1] Java optimization framework [VRCG+99] for GROUM generation. This enables the generator to leverage code semantics in addition to the syntax provided by ASTs. Furthermore, we extended the GROUMs to also capture constants that are used in method invocations to be able to distinguish between correct and anomalous usages of the JCE, as it makes heavy usage of constants for the configuration of API objects (e.g., Listing 1.1).

A GROUM is a directed acyclic graph (DAG) with a set of nodes and a set of edges as defined in graph theory. It models control and data flow between method invocations on specific objects inside a given Java method. 'Directed' specifies that an edge from a node A to a node B is unidirectional, i.e., B can be reached by A but not the other way around. Furthermore, the acyclic property states that if there is a path from A to B, i.e., a chain of directed edges from A to B containing an arbitrary number of nodes, there must not be an edge back from B to A. DAGs provide an intuitive and sound way to model a method's control and data flow, i.e., both, with the exception of loops, propagate downwards through the inspected method and thus fulfill the directed and acyclic properties of the graph. Loops constitute a special case since they allow data and control to flow backwards at some point. Since a lot of frequent subgraph mining algorithms cannot handle those backward edges, they are omitted in the GROUMs [NNP+09, EBFK13]. This does not hurt soundness with regards to pattern mining or the anomaly detection since it is enough to model scope and existence of a loop in the GROUMs. For example, in Figure 3.1 there is an edge from the *LOOP* node to the *RETURN* node which models the control flow when the loop is not taken, furthermore, there is a path between those nodes over *Example.bar* which models the flow through the loop, thus, existence (*LOOP* node) and scope (every node between *LOOP* and the merge point of both outgoing paths) are implicitly represented by the GROUM.

The node and edge types of our model are a slight variation to those defined by Nguyen et al. [NNP+09] and tailored towards the analysis of cryptographic libraries. The following types are part of the model:

Action nodes

Model the invocation of a constructor or method of a specific object. The label of an action node is $C.m$, where C is the class name of the target object and m the method name.

Control nodes

Represent branching points of control structures like if-else blocks or while/for loops.

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

```java
public byte[] encrypt(SecretKey secretKey) throws
    Exception {
  try {
    Cipher cipher = getAESCipher();
    cipher.init(Cipher.ENCRYPT_MODE, secretKey);
    byte[] ctBytes = "ClearText".getBytes();
    return cipher.doFinal(ctBytes);
  } catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
  } catch (NoSuchPaddingException e) {
    e.printStackTrace();
    throw e;
  }
  return null;
}
```

Listing 3.1: AES encryption with exception handling.

```java
public Cipher getAESCipher() {
  String aes = "AES/CBC";
  return Cipher.getInstance(aes);
}
```

Listing 3.2: AES Cipher factory method.

```java
public void control() {
  String a = getString();
  if (a.equals("foo"))
    foo();
  while (a.equals("bar"))
    bar(a);
}
```

Listing 3.3: Control structures.

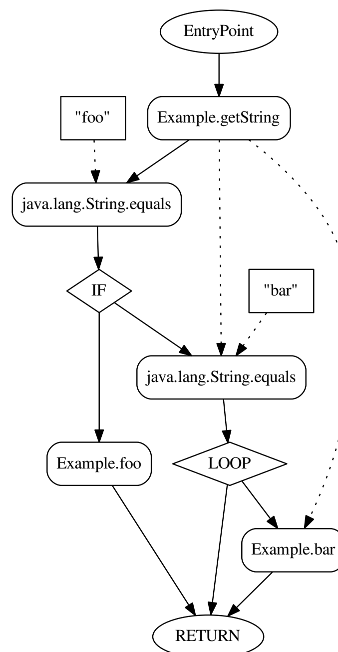

Figure 3.1.: GROUM for Listing 3.3.

Constant nodes
: Model literal constants (such as string or integer constants) and are especially important in the context of cryptographic APIs.

Entry nodes
: Represent the entry point of a method.

Exit nodes
: Model the exits of a method and can be either a *RETURN* (regular exit) or *THROW* (exceptional exit) node.

Edges

Model either control flow or a data dependency between two nodes. Contrary to the implementation of Nguyen et al. [NNP+09], a data dependency between two nodes A and B is only represented by an edge if the statement modeled by A defines a variable (or constant) that is then later used as calling target or parameter by the statement represented by B.

Figure 3.2 shows the GROUM for the AES encryption example in Listing 3.1 as generated by our toolchain before the augmentations (see Section 4.2) we had to make for the process of building inter-procedural GROUMs. Action nodes are shown as rectangular nodes with rounded edges, constants are shown as rectangular nodes, and exit as well as entry points are visualized by elliptical nodes. Dependency edges are visualized as dotted edges to have a clearer optical distinction, while there is actually no semantic distinction between those and control flow edges in the model. Note that there is always just one edge between two nodes and, thus, some edges model control and data flow, e.g., the edge between *getAESCipher* and *javax.crypto.Cipher.init*.

It has to be also noted that there is an edge modeling control flow for every statement inside - and right before - the *try*-block to the first statement of every *catch*-block. These edges are derived from *Soot's ExceptionalUnitGraph* which models the regular and exceptional control flow between Jimple units (statements). It might seem a bit counterintuitive to also have an edge from the statement right before the *try*-block to its handling *catch*-block. This is due to the way *Soot* models exceptional flow: Every statement preceding a statement that might throw an exception has an edge to the exception-handler. If the throwing statement can have side effects, there is also an edge to that handler. This is the case since the statement right before the throwing statement may actually be the last one executed before the exception-handler, or, if the throwing statement has side effects, it might have been executed partially. Since *Soot's* throw analysis, which generates the *ExceptionalUnitGraph*, is not precise in deciding which exact statement inside the *try*-block actually throws an exception, and every statement is a method invocation – hence has side effects – the preceding statement and every statement inside that block is assumed as a potential last statement before execution of the exception handler. We suggest having a look into Árni Einarssons and Janus Dam Nielsens Survivor's Guide to *Soot*[2] for further information on Jimple and *Soot's* control flow graphs.

3.2 Patterns

We employ the *ParSeMiS*[3] frequent subgraph mining framework to mine patterns of frequent object usage in a set of GROUMs. A pattern is a GROUM or sub-GROUM that occurs frequently, i.e., with a frequency higher than a given threshold, in a set of GROUMs and is likely to model the correct usage of some library (or, more generally: correct interaction of Java objects) due to its frequent usage [NNP+09, Wer07]. Figure 3.4 shows a pattern which models encryption with the help of a *Cipher* objects, as mined from a set of GROUMs using the JCE. Again, there is no distinction between data and control flow edges, neither during nor after the mining. Nevertheless, the miner does consider edge labels when matching edges albeit the GROUM model does not use any. Furthermore, a node's type is not considered during mining, only its label.

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

[3] https://www2.informatik.uni-erlangen.de/EN/research/zold/ParSeMiS/index.html

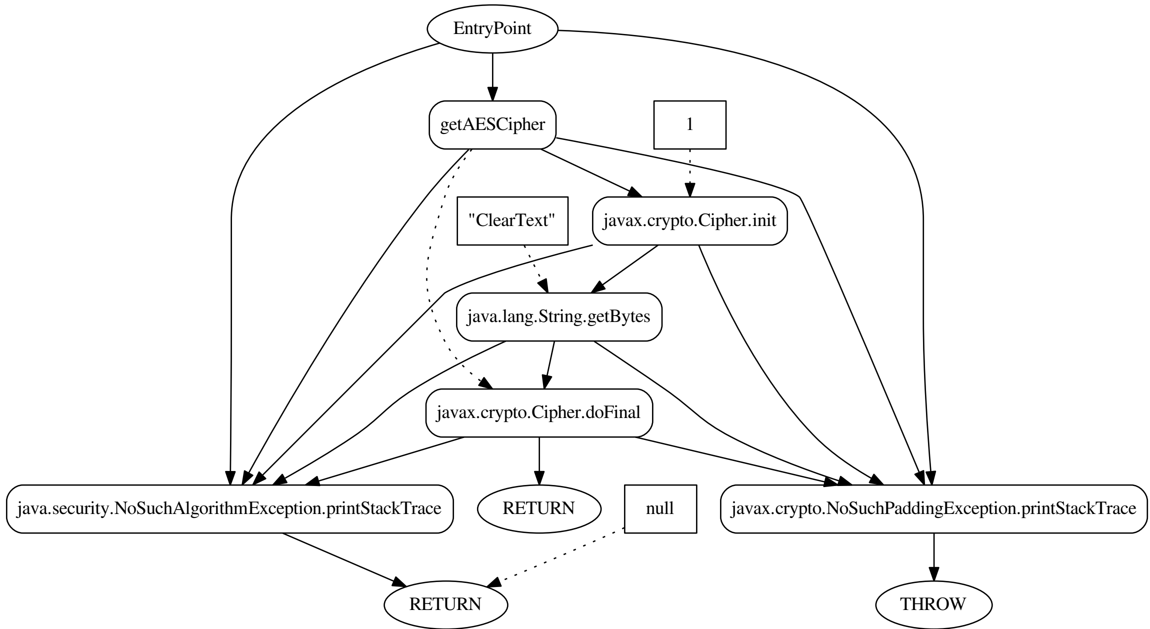

Figure 3.2.: GROUM for Listing 3.1.

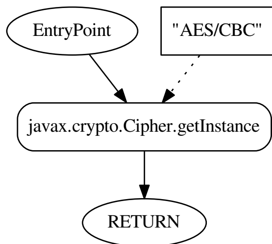

Figure 3.3.: GROUM for Listing 3.2.

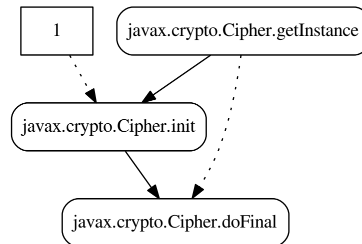

Figure 3.4.: Pattern for javax.crypto.Cipher usage.

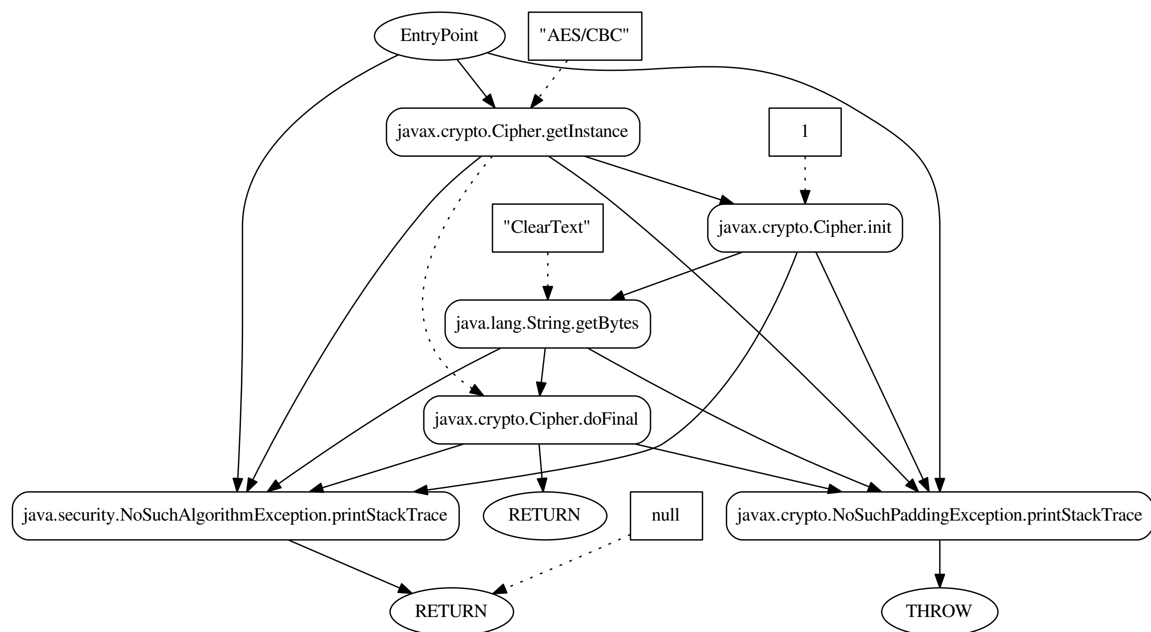

Figure 3.5.: Interprocedural GROUM for Listings 3.1 and 3.2.

3.3 Anomaly Detection

After the mining of a pattern, the pattern can be used to detect anomalies in a set of GROUMs. The GROUM which is used to match the pattern against, is usually called the **target**. If the number of matching nodes between target and pattern exceeds a certain threshold, the match is called an **occurrence**. For example, if the pattern contains four nodes and the threshold is 50%, the pattern and target have to have at least two nodes in common so that the overlap is considered an occurrence. This threshold is also called **minimal overlap**. If an occurrence was found and it does not match the whole pattern, i.e., some node or edge of the pattern is missing in the target, it is considered a **violation** of the pattern and thus an anomaly in usage of the object relation modeled by the pattern. For instance, if a target matches every node of the pattern in Figure 3.4 except for one node, an occurrence of the pattern is found, furthermore, it is violated because of the missing node and thus an anomalous use of JCE-*Cipher* has been detected.

Note that setting the threshold too low leads to large numbers of violations (usually false positives). These are caused by minimal overlaps between pattern and target. Commonly used objects or methods, like *NullPointerException* or *System.out.println(..)*, will oftentimes lead to such false positives. It is crucial to adjust the threshold so that the amount of false positives is low, while still detecting small, i.e., consisting of just a few nodes, flawed object usages.

3.3.1 Intra- vs Interprocedural GROUMs

Consider the GROUMs in Figures 3.2 and 3.3 (further on, referred to as *G1* and *G2*), where the latter models the *getAESCipher* factory method (Listing 3.2) used in the former. When using both as targets for the anomaly detection with the pattern in Figure 3.4 to match against, the detector will report one

violation of the pattern for each of the GROUMs. Firstly, the *javax.crypt.Cipher.getInstance* node is missing in *G1* and, furthermore, *G2* is missing all three other nodes to conform to the pattern. Since the object relation modeled by the pattern actually exists in the code – albeit distributed among two methods – it is obvious that both of the reported anomalies are false positives which are caused by matching targets and patterns on an intraprocedural level. To mitigate false positives caused by this, means to build GROUMs on an interprocedural level are needed. While building those interprocedural GROUMs, it is important to keep the semantics of the original code to not introduce additional anomalies, or, even worse, hide existing ones. To eliminate the two false positives for the previously given example while still keeping semantic soundness, an interprocedural GROUM for Listings 3.1 and 3.2 would look like the one in Figure 3.5. We propose an approach that is capable of this in the following section.

4 Interprocedural GROUMs

In this chapter, the approach we developed to build interprocedural GROUMs is explained on a conceptional level. Furthermore, necessary augmentations to the existing GROUM model are provided.

4.1 Inling GROUMs vs Methods

When constructing interprocedural GROUMs, there are basically two major, but inherently different, approaches: Working on the source code level or working on the GROUM level. When doing the former, one could, for example, use an already existing Java bytecode inliner to inline the methods at the source code level before using the *Cryptominer* to extract the GROUMs. Or, furthermore, one could design an interprocedural code analysis with the *Soot* framework that uses the already existing intraprocedural analysis – used for GROUM generation – and build the interprocedural GROUMs on-the-fly. The latter approach involves replacing nodes by the corresponding GROUMs on the level of the GROUM model, which will further on be referred to as *GROUM inlining*.

All approaches have their benefits and shortcomings. Inlining on a Java bytecode basis has the big advantage that there already exist proper tools for the purpose, and even *Soot* does provide some inlining facilities out of the box [AB05, VRCG+99, BCF+99, SHR+00]. Furthermore, bytecode inlining seems to be the most generic approach and most robust against changes of the GROUM model. The big drawback here is that there are strict rulings that might interfere with what is possible in the context of GROUMs and GROUM-based pattern matching. For instance, most of the tools just inline monomorphic calls, i.e., calls that can have only one possible target object at runtime [BS96]. This is because monomorphic calls can be resolved unambiguously, in contrast to polymorphic calls which can have multiple possible receiver objects of a method call. Furthermore, these tools mostly consider performance, code size or call frequency when deciding if a method should be inlined [CL06, SCWK13, AFSS00], which makes sense in the context of code optimization since inlining too much could bloat the code and greatly increase the program's size with a minimal performance benefit. Lastly, those tools do not provide sufficient potential for extension which, foremost when deciding when to inline, is of high relevance for the purpose of interprocedural GROUMs. Since the aim when building interprocedural GROUMs is the elimination of false positives during the anomaly detection step, usual code inlining heuristics seldom provide the necessary metrics. For example, the decision if a method should be inlined should rather depend on it using the API of interest, or, more generally, if the inlining is able to reduce false positives during anomaly detection, than on its size or number of calls during the program's execution. Using an interprocedural code analysis in *Soot* to augment the already existing intraprocedural builder would be a more flexible approach. One of the biggest benefits here is the ability to use the *Soot* framework and all its capabilities when doing the inlining, e.g., mapping of parameters from caller to callee is basically there out-of-the-box, or, the call-graph is available and do not has to be serialized for later processing. Beyond this, the approach yields similar benefits as inlining GROUMs on the graph level. While inlining directly on the GROUM model does not provide the stated benefits of the other approaches, it does provide the most flexibility since an inliner can be tailored to the needs of GROUM inlining with respect to pattern mining and anomaly detection, e.g., inlining a GROUM only if the result would lead to a decrease of anomalies detected in it. Furthermore, inlining on the GROUM level ensures that only units of interest are handled.

In contrast, inlining on the bytecode level would process everything that is not captured by the GROUM model and thus would introduce unnecessary overhead. Last but not least, such an inliner has the benefit that existing GROUM databases can be used as target for inlining and thus the source – and multiple comparably expensive GROUM generation runs – are not needed. This gives great flexibility in handling GROUMs e.g., keep, compare or mix intra- and interprocedural GROUMs of existing GROUM databases. Because of the stated advantages and the lesser important disadvantages of GROUM inlining, focus of the thesis is given to this apporach.

For the sake of readability, the GROUM that has nodes which are inlined will be referred to as **master** and the "to be inlined" GROUMs as **inlinees**.

4.2 Extended GROUMs

To be able to inline GROUMs, it is necessary to augment the model to carry the necessary semantics for such a task. Firstly, GROUMs and their nodes have to be extended to carry the full signature of the modeled methods. While the pattern matching works on method names only, such as *javax.crypto.Cipher.doFinal(...)*, for the inlining it is important to be able to distinguish methods unambiguously to inline the exact method when there are multiple overloaded versions. For the same purpose, a GROUM itself needs to carry the signature of its modeled method. Besides supplementing action nodes with the signature of the modeled methods, no other changes to the nodes are needed to enable the inlining process. Another very important augmentation is the modeling of method parameters, i.e., which parameters a method has and how these are used. This is necessary to correctly map incoming data flow edges from the master to an inlinee with method invocations inside the inlinee. For example, consider Listings 4.1 and 4.2 where multiple parameters are passed from the master to the inlinee and further processed inside it. When inlining, it is necessary to know from which nodes the parameters are coming from and which nodes are using them to correctly model the data flow. When looking at the GROUMs for the stated listing (Figures 4.1 and 4.2), it is obvious that there are no means to reason which nodes should be connected in which manner besides the usual control flow. One does not know if the *getString* node in the master does have the data dependency to the *inlinee* node because of passing its returned value as the first or second parameter or if it is even used as receiver object of the method call. It's also unclear if the returned value of the inlinee is further used. When looking at the inlinee, there is no information about the passed parameters at all. If the value returned by the *doSomething(...)* method would be also the returned value of the inlinee, there would be at least a data dependency edge from it to the *RETURN* node, but since the control flows the exact same way as the data, data dependency edges are hidden by the control flow edges (the same is observed in the master).

In the following sections, the necessary extensions to the GROUM model are explained in detail. Note that, while the augmentations are crucial for the effectiveness of the inliner, they also leave the model more complete and less ambiguous compared to the previous, more simplistic one.

4.2.1 Advanced-Edges

The *advanced-edges* model introduces several new edge types that carry the semantics needed for GROUM inlining. The two major edge types, namely control and data flow edges, are further subdivided as follows:

```java
public void master(){
  String param = getString();
  String param2 = getOtherString();
  String result = inlinee(param, param2);
  print(result);
}
```

Listing 4.1: Master propagating values to/from inlinee.

```java
public String inlinee(String param, String
    param2) {
  doSomething(param);
  return doSomethingElse(param2);
}
```

Listing 4.2: Inlinee that does something with parameters.

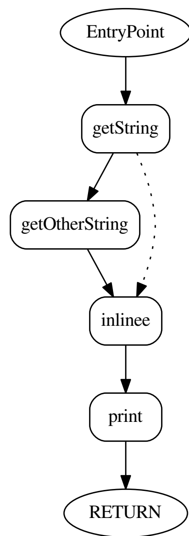

Figure 4.1.: GROUM for Listing 4.1.

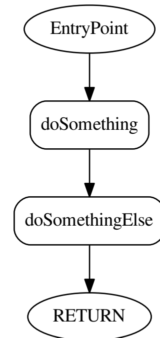

Figure 4.2.: GROUM for Listing 4.2.

```
class Example {
  public void exceptions() {
    try {
      foo();
    } catch (NullPointerException e) {
      printSomething();
    } catch (Exception e) {
      printSomething();
      throw e;
    }
  }
}
```

Listing 4.3: Exception handling.

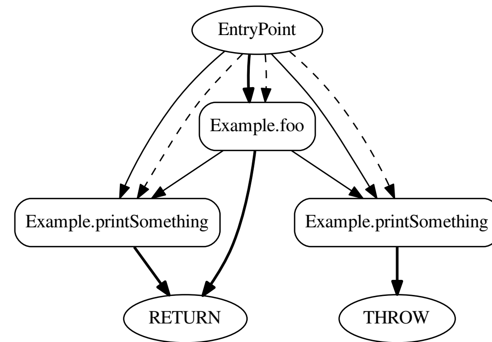

Figure 4.3.: GROUM for Listing 4.3.

Control Flow Edges

For the explanation of the new control flow edges model, consider Listing 4.3 and fig. 4.3. Note that in our GROUM visualizations, control flow edges are always solid and data flow edges are dotted or dashed.

Usage Order Edges model only the non-exceptional control flow and thus the intended usage order of objects. They may branch on conditionals and merge again at some point (also including merge points after exception handling like on the *RETURN* node). But besides that, there is always just up to one incoming and/or outgoing usage order edge for any node. Usage order edges are drawn as bold, solid lines in the visualizations. In Figure 4.3 non-exceptional control flows from the *EntryPoint* node over the *Example.foo* node to the *RETURN* node. Also, if in one of the exception handlers (which are only reached through exceptional flow), control flows normally again, e.g., from the *Exmple.printSomething* node to the *THROW* node.

Exceptional Flow Edges model the exceptional flow in a method, i.e., if some statement is inside a *try*-block, an exceptional edge is inserted between that node and the first node of the exception handler. Exceptional flow edges are shown as solid, non bold edges in the visualization. Considering the example shown in Figure 4.3, there are four of these edges: one for every statement inside and before the *try*-block to each exception handler.

Data Flow Edges

For the explanation of the new data flow edges model, consider Listing 4.4 and fig. 4.4.

Parameter Edges model that either the method modeled by the GROUM has a parameter which is then further used, or that a node gets some variable or constant passed as argument. Note that the term variable is not precise in this context since the argument does not necessarily need to be held in an intermediate variable, but can also be passed in the same statement, e.g., writing `process(s1.concat(param), "Constant")` instead, would produce the exact same GROUM.

Parameter edges hold the indices of the used parameters. These are visualized as edge labels, formatted as *x:y*, where *x* is the index of the formal parameter of the modeled GROUM (in-index)

and y is the index of the parameter which is used when an argument is passed to a method modeled by an action node (out-index). Both are zero-based, i.e., "0:1" models that the first parameter of the GROUM is passed as second argument of a method call. The indices also allow for some special cases: Is $x = -1$ and the edge's source is an entry node; the modeled incoming "parameter" is the *this*-reference that is implicitly passed to the method. Is $x = -1 \neq y$ and the edge's source is not the entry node; the edge models the passing of a value that is returned by the edge's source node to the target node as $y'th$ argument (e.g., the edge between *java.lang.String.concat* and *java.lang.String.compareTo* in Figure 4.4). Another special case is $y = -1$ and $y \neq x$, this models the case where a parameter of the GROUM is used as receiver of a method call (e.g., the edge between *EntryPoint* and *java.lang.String.compareTo*).

Generally speaking, $x = -1$ models that the edge's source returns a dependency, while $y = -1$ models that the edge's target uses the dependency as receiver of a method invocation. Hence, the *this*-reference is modeled as return value of an entry node, likewise, a constant is handled as if it were the return value of a constant node. Furthermore, the exit nodes build another special case. Having a parameter edge with $y = -1$ to a *RETURN* node means that the node returns the dependency. Similarly, having an edge with $y = -1$ to a *THROW* node models that the dependency, i.e., an exception object, is thrown. One could also interpret this as if the exit nodes where method calls with the dependency as receiver object.

Dependency Edges model the case where the edge's source is the return value of a node and its target is the receiver of a method invocation. This edge implicitly models a parameter edge with $x = y = -1$ as indices. The label is omitted here to improve performance during the pattern mining and anomaly detection steps (no unnecessary string comparisons for the label). Examples for this edge type are: The edge from *EntryPoint* to *Example.getString*, modeling the use of the *this*-reference as receiver of the call to *Example.getString*. The edge from *Example.getString* to *java.lang.String.concat*, where the string returned by the former node is then used as receiver of the *concat* method call. And lastly, the edge from *java.lang.String.comareTo* to *RETURN*, which models that the value returned by the former node is returned by the method.

4.3 Inliner

In this section, the process of inlining is described on a conceptional level. Details on the inliner's implementation can be found in Section 5.2.

First, we introduce the term *inline heuristic*. An inline heuristic is a means to decide if a given GROUM should be inlined. This is important because inlining every possible action node in a GROUM might not always be desired. For example, if one wants to analyze object usages of a specific API, it might be beneficial to just inline methods that actually use the API. See Section 4.3.1 for a more in depth explanation of inline heuristics.

To build an interprocedural representation of a given GROUM – the master –, the inliner traverses every node of it, determines if a GROUM is available for the method represented by the node, and decides, based on the specified inline heuristics, if the node should be inlined. If so, the original node and its incoming as well as outgoing edges are removed from the master. Further on, nodes of the inlinee are inserted into the master, with the exception of entry and exit nodes since these model the interface to the master and are only used to map edges of the master to the inlinee. Also, all edges between the nodes of the inlinee are also added to the master. The master now contains an unconnected sub-graph (or island) which essentially is the inlinee without its interfacing nodes.

```java
class Example {
  public String dataFlow(String param) {
    String s1 = getString();
    String s2 = s1.concat("Constant");
    return param.compareTo(s2);
  }
}
```

Listing 4.4: Data flow through method.

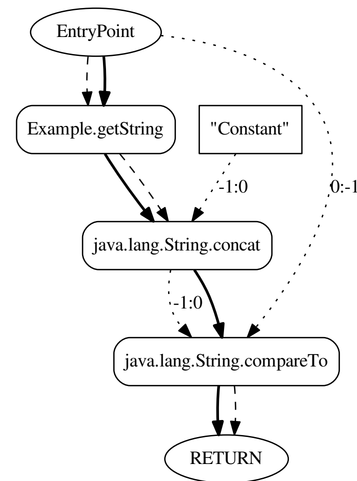

Figure 4.4.: GROUM for Listing 4.4.

After the preliminary inlining step, the inlinee has to be connected to all nodes that were connected to the original node. This is done edge by edge for every incoming or outgoing edge to or from the inlined node. There are basically three situations that can occur: The edge of interest connects a node with an inlinee, it connects an inlinee with a node, or the edge connects two inlinees (happens if two connected nodes are inlined). Based on the case and edge type, different actions have to be taken to correctly insert an edge to, from or between two inlinees:

Inserting edge from node to inlinee

Usage Order or Exceptional Flow Edge: The source node of the edge has to be connected to the first actual node of the inlinee than can be reached through non-exceptional control flow, with a usage order edge. This case is a rather trivial one because there is always only one non-exceptional control flow edge starting from the entry node. Furthermore, for each exceptional successor of the inlinee's entry node, an exceptional flow edge has to be added from the original source to the successor. This is the case because the original source node now is the last statement that might have been executed before exception handling of the inlinee, just like the entry node was before (see Section 3.1 for a recap on exception handling in GROUMs).

Parameter Edge: For each parameter edge originating from the inlinee's entry node whose in-index (x_i) equals the out-index (y_m) of the original edge, a new edge has to be inserted from the original source to the target of the inlinee's parameter edge. Depending on the indices, a different type of edge has to be inserted. If $x_m = y_i = -1$, a dependency edge is inserted, otherwise, a parameter edge with the specific in- and out-indices (x_m and y_i) is inserted.

Dependency Edge: Since a dependency edge from one node to another models that the return value of the source node is the receiver object of the call modeled by the target, the to-be-connected nodes of the inlinee can only be calls with the *this*-reference as receiving target. Since the *this*-reference of the inlinee is the same object as the one returned by the source node of the original edge, a new

dependency edge has to be created from the original source node to every node that is connected to the inlinee's entry node with a dependency edge.

Inserting edge from inlinee to node

Usage Order Edge: For each of the control flow predecessor of every non-exceptional exit node of the inlinee (each node that is connected with a *RETURN* node by a usage order edge), there has to be a usage order edge from this node to the target node of the original edge. This is because non-exceptional control flow can leave the inlinee through every *RETURN* node.

Exceptional Flow Edge: Through the incompleteness of the GROUM model, it cannot be decided which of the inlinee's nodes can produce a specific exception or which type of exception is thrown by a given *THROW* node. Hence, the inliner does insert an exceptional edge from each action node of the inlinee to the exception handler of the master (target of the original edge). This is rather an approximation than a precise step of inlining. For a more precise solution, the model has to be extended so that *THROW* nodes, as well as exceptional flow edges, carry the actual type of exception that is handled. Furthermore, it is necessary that the GROUMs also model implicit exceptional flows, i.e., exceptions that are not explicitly thrown inside the method – through a throw-statement – itself but are further passed upwards in the call-hierarchy. Nevertheless, there will still be situations where only the stated approximation leads to sound results, e.g., catching of runtime exceptions. Since a more elaborated approach to exception handling in GROUMs is out of scope of this thesis and the importance of exception handling in the context of anomaly detection is unclear, we consider this to be addressed in future work.

Parameter or Dependency Edge: Every returned dependency (action or constant node) of the inlinee can be the source of the edge. Thus, every node that is connected to a *RETURN* node with a dependency edge has to be connected to the target of the original edge with an edge of the same type as the original edge. Note that there can also be the case that a parameter of the inlinee is returned by it, which will further be referred to as **loop-through**. Detailed insights on loop-throughs (and similar special cases) will be given in the following section.

Inserting edge from inlinee to inlinee

Usage Order Edge: For each control flow predecessor of every non-exceptional exit node of the source inlinee, there has to be a usage order edge from it to the control flow successor of the target inlinee. Furthermore, for each of the predecessors, there has to be an exceptional flow edge between them and every node of the target inlinee that is connected through an exceptional flow edge to the target's entry node. This models the case where the target's entry node might be the last executed statement before the reach of an exception handler.

Exceptional Flow Edge: For each action node in the source inlinee, an exceptional flow edge is inserted to the first non-exceptional control flow node in the target inlinee. This models the case where the target inlinee is the first node in the exception handler of the master GROUM. The approach is similar to the already stated approximation for exceptional flow edges.

Parameter Edge: This models the case where the source inlinee returns an object that used as argument to the target inlinee. Hence, every returned dependency (action or constant node) of the source

inlinee has to be connected to every node that uses the corresponding parameter in the target inlinee. Since by definition all returned dependencies have an in-index of $x = -1$, the type of the edge depends solely on the out-index of the corresponding parameter edge of the target inlinee. For example, the target inlinee has an action node that is connected by a parameter edge with indices $0 : -1$ (uses the first parameter as receiver object of method call) and the original edge has indices $-1 : 0$, thus, every returned dependency of the source inlinee has to be connected to the stated action node of the target inlinee by a dependency edge. Again, dependencies might get looped-through, in fact, this might happen for both inlinees.

Dependency Edge: This case is also similar to the one for "node to inlinee". The only possible edge targets in the target inlinee can be *this*-reference invocations and thus all returned dependencies of the source inlinee have to be connected with a dependency edge to every action node of the target inlinee that uses the *this*-reference (is connected with a dependency edge to its GROUMs entry node).

Empty Methods and Loop-Throughs

There are also some special cases where the described methods do not work.

First, the inlinee might contain no action nodes, i.e., no method calls are conducted. This happens if a method is empty (e.g., stub) or just returns some constant (see Listing 4.5). In such a case, the entry node is directly connected to an exit node with an usage-order edge as shown in Figure 4.5. For this example, after the preliminary inlining step, only the constant node will be left in the master. Connecting it to the master can be handled like described before. Connecting an incoming or outgoing usage-order edge to or from the inlinee, in contrast, need to be handled differently. Remember, there can only be one outgoing usage-order edge for every action node but multiple incoming ones (since it might be the merge-point of a preceding control structure). Hence, every non-exceptional control flow predecessor of the inlinee has to be connected to the non-exceptional control flow successor of it, to keep control flow sound for the interprocedural GROUM. If multiple connected inlinees do not contain action nodes, the procedure has to be conducted successively until the inserted edge has an entry/exit node of the master or any action node as source and target.

The second special case, a loop-through, occurs if a method returns one of its parameters on some path, also including the *this*-reference (e.g., when implementing fluent interfaces in Java [Bod14, Fow05]). Listing 4.6 and fig. 4.6 shows such a case. Again, the previously described methods for edge insertion do not work here. Generally speaking, a parameter or dependency edge has to be inserted as if there were no inlinee between two nodes. So there has to be a new edge inserted depending on the in-index (x_{m1}) of the incoming edge to the inlinee and the out-index (y_{m2}) of the outgoing edge of the inlinee. This has to be done for each incoming and outgoing parameter or dependency edge of the inlinee for which the parameter or *this*-reference is looped-through. Furthermore, if multiple connected inlinees loop-through variables, the procedure has to be conducted successively until the new edge has a non-loop-through inlinee or normal node as source and target (at the extreme, this leads to connecting an entry and exit node of the master).

4.3.1 Inline Heuristics

When conducting the inlining process, means to decide if a node should be inlined or not are needed. If an action node can be inlined, i.e., a GROUM whose signature matches the one of the action node,

```
class Example {
  public String emptyMethod() {
    return "Constant";
  }
}
```

Listing 4.5: Method returning a constant.

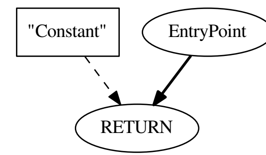

Figure 4.5.: GROUM for Listing 4.5.

```
class Example {
  public String loopThrough(String param) {
    foo();
    return param;
  }
}
```

Listing 4.6: Method looping-through a parameter.

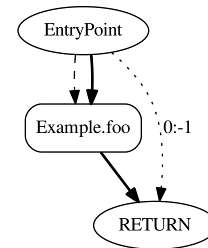

Figure 4.6.: GROUM for Listing 4.6.

a chain of inline heuristics is evaluated to make that decision. This is useful, for example, if one wants to analyze the use of a specific API and thus is only interested in inlining methods that actually use the API. Furthermore, since the whole approach aims at detecting anomalies in object usage of APIs, API methods themselves should not be inlined since this would rather model their internals than their usage. Another big problem when doing the inlining in the context of anomaly detection is that, if a method that contains anomalies (regardless whether they are true or false positives) that would not disappear after the inlining, the number of detected anomalies would increase for every such method that is inlined. For example, if an anomaly for a method *A* is detected, the same anomaly for every method that calls *A* would be detected after the inlining. Thus, the number of detected redundant anomalies might heavily increase after the inlining, dependent on the number of calls to the anomalous method. To overcome this problem, a reasonable selection of inline heuristics for the specific use-case has to be made.

We have implemented several inline heuristics that can be freely combined as proposed in the following:

Always Inline Most basic heuristic which allows every action node for which a GROUM is available to be inlined.

Level Based Inline based on the depth of method calls, i.e., only inline as long as the number of steps in the call-hierarchy is lower than some threshold. For example, from a given master, if inlining two levels, do not inline nodes of a GROUM that is the inlinee of an inlinee of the master.

No Recursion Prohibits the inlining of recursive method calls. Also prevents inlining of recursive calls if they do not occur in the same method but somewhere downwards the call chain.

Same Class Inline if inlinee is defined in the same class as the master.

Same Package Similar to the *same class* heuristic but on the scope of packages.

Returns Or Gets Class Inline if the inlinee uses a specific class in its signature, i.e., returns a value or gets a parameter with the type of a specific class.

Returns Or Gets Package Similar to the *returns or gets class* heuristic but uses packages for matching.

Note that these are heuristics that we considered as reasonable for the purpose of detecting anomalies of JCE usage. The inliner allows for arbitrary heuristics one could come up with. Depending on the use-case, other heuristics might make sense, e.g., inlining based on the inlinee size or frequency when doing the inlining pre-pattern-mining. More on the design and implementation of inline heuristics can be found in Section 5.2.4, furthermore, evaluation of the different heuristics can be found in Section 6.3.

4.3.2 Inline Tree

In some cases it might be necessary to investigate which GROUMs have been inlined into a given master, e.g., if one wants to determine if a detected anomaly has been propagated from an inlinee to the interprocedural master. We propose *inline trees* for that purpose. They model the hierarchy of inlined GROUMs for a given master and can also be seen as a means of observing inlining history, i.e., which GROUMs have been inlined in which order into a master.

Listing 4.7 shows an example of such a tree as produced and printed for a master of our test set. When closely examining the tree, one can observe that the exact order, depth and number of inlinees can easily be derived from its textual representation: There are three methods inlined on the first level, three on the second and two on the third level.

```
<three_layers.Master: void <init>()>
|
|- <three_layers.HelperClass1: void <init>()>
|
|- <three_layers.HelperClass1: javax.crypto.Cipher genCipher()>
|    |
|    |- <three_layers.HelperClass2: void <init>()>
|    |
|    |- <three_layers.HelperClass2: java.security.spec.PKCS8EncodedKeySpec getKeySpec()>
|    |    |
|    |    |- <three_layers.StaticHelper: byte[] getKey()>
|    |
|    |- <three_layers.HelperClass2: java.security.KeyFactory getKeyFactory()>
|         |
|         |- <three_layers.StaticHelper: java.lang.String getKeyAlgorithm()>
|
|- <three_layers.HelperClass1: byte[] encrypt(javax.crypto.Cipher,java.lang.String)>
```

Listing 4.7: Inline tree for a three-layer inlining test with three helper classes.

4.4 Virtual Callsites

Until now it has been assumed that method calls can be unambiguously resolved to their target object. This might not always be the case since Java allows the invocation of virtual methods, i.e., methods whose receiver object is resolved during runtime and might not be known in a static context. For example, consider Listing 4.9 where the method *callFoo* gets an object of type *AbstractClass* and calls the

```
abstract class AbstractClass{
  abstract void foo();
}
```

Listing 4.8: Some abstract class.

```
void callFoo(AbstractClass receiver) {
  receiver.foo();
}
```

Listing 4.9: Method using AbstractClass.

```
class SubClass1 extends AbstractClass{
  @Override
  void foo() {
    doSomething();
  }
}
```

Listing 4.10: Some subclass of AbstractClass.

```
class SubClass1 extends AbstractClass{
  @Override
  void foo() {
    doSomethingElse();
  }
}
```

Listing 4.11: Another subclass of AbstractClass.

virtual method *foo* of that class. At this point it is unclear which of the sub-classes of *AbstractClass* will be passed to the method and, hence, which actual implementation of *foo* needs to be inlined here.

Since the inliner is unable to resolve such calls by the matching of callsite and GROUM signatures alone, the actual call-graph for the analyzed code has to be provided. From it, it is possible to determine which actual receiver object – and thus, which method implementation – can reach the callsite of interest. The precision of such a call-graph can vary tremendously, e.g., some may provide every possible class as type of the receiver object (class hierarchy analysis [DGC95]) where others might only provide classes that are actually used in the analyzed code (rapid type analysis [BS96]). Generally speaking, the choice of call-graph algorithm has to be made based on the specific use-case, e.g., analyzing an application might allow for a more precise call-graph than analyzing a library.

The inliner allows for customizable lookup of inlinees for a given node. To test if resolving virtual method calls is able to reduce false positives during anomaly detection, an implementation of a naive approach, where the inliner resolves these calls to just one of the possible targets as stated by a (class hierarchy analysis) call-graph, is given.

It has to be mentioned that, while using call-graphs to resolve virtual method calls might yield better detection results at some points, i.e., less false positives, the opposite might be true in other cases. For example, consider a master GROUM which has two possible inlinees that again have two possible inlinees: If every of the inlinees on the second level carry an anomaly that is not eliminated by an intraprocedural representation, i.e., is not caused due to the separation of object usages, there are four detected anomalies before and after the inlining. Generally speaking, without resolving virtual callsites, the number of detected anomalies – not caused by the separation of object usages – stays the same, when comparing the sum of anomalies found in the master and all its inlinees with all anomalies found in the master's intraprocedural representation. Let's assume now that one of the inlinees on the second level can be resolved to two different GROUMs for which, again, both produce an anomaly during detection. This leads to two different interprocedural representations of the master, since on the second level there are two different inlinees possible. Comparing the number of detected anomalies for both cases leads to very different results now: Without inlining, there are five anomalies found (one for each possible inlinee), with inlining, however, there are four anomalies detected for each of the interprocedural masters since both contain all three anomalies of the non-virtual method calls and, furthermore, one of the anomalies of the corresponding virtual callsite target. This gets even worse if there are multiple combinations of virtual callsite targets that can be inlined, i.e, for each possible combination of inlinees an interprocedural GROUM has to be generated, again carrying all anomalies of shared inlinees. If the inlinee hierarchy gets deep, the number of possible combinations can easily explode, based on the number of virtual callsites

and possible receiver objects of those calls. The situation might not be that bad in cases where one object is used as target for multiple callsites in one method. For example, if a method has two virtual calls that are both targeted on the same object and the object can have two different actual receiver objects at runtime, the combinations shrink from four to two possible interprocedural representations. Nevertheless, the call-graph needs to carry such information to do this optimization.

It is unclear how big the benefits or drawbacks of resolving virtual callsites are in practice. Techniques have to be found to overcome the stated problems, or at least minimize their downsides. Furthermore, optimal call-graph representations have to be determined and evaluated for their trade-offs. Since a comprehensive handling of virtual callsites is out of scope for this thesis, we consider this as future work. Nevertheless, an implementation of a naive approach to handle virtual callsites is given in Section 5.3. Furthermore, an outlook on the potential of their resolution can be found in Chapter 6.

5 Implementation

This chapter contains detailed information on the implementation of the inliner, its components and most of the previously explained concepts.

5.1 Extended GROUMs

For the implementation of the *extended GROUMs* model, we have applied minor changes to the *Groum* class found in *grouminer.groum.sit.Groum*. The extension of the model mostly consists of augmentations and subclassings of the previously existing *Node* and *Edge* classes, to conform to the specification stated in Section 4.2.

5.1.1 Dependency Manager

For generating the *extended-GROUMs* model, a lot of changes had to be made to the dependency management during GROUM generation. Thus, we decided to make the GROUM generator more robust to future changes of the GROUM model. The *IDependyManager* interface abstracts from the process of dependency handling, i.e., generation of data dependency edges, and encapsulates the needed procedures. If one wants to implement a different handling of dependencies during GROUM generation, it is sufficient to pass a corresponding implementation of the interface to the GROUM generator. The interface allows for handling parameters, constants and basically every other possible dependency. If one does not need a specific kind of dependency, the implementation of the corresponding method can just be omitted. Figure 5.1 shows the public interface of the *IDependencyManager*.

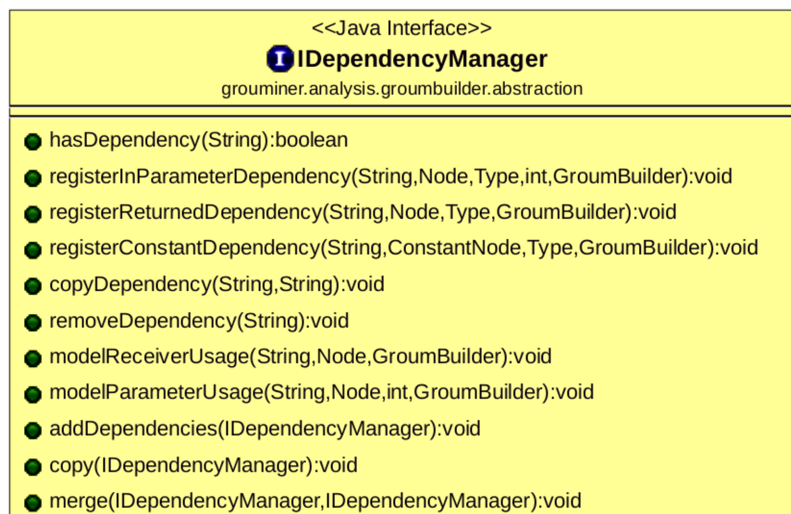

Figure 5.1.: *IDependencyManager* class diagram.

In the context of the interface, a *dependency* is a string that is used to identify an actual dependable object in the code, e.g., a variable, constant or parameter. To register such a dependency, the corresponding

method is called when the dependency is defined in code, e.g., a dependency for a parameter is registered when the GROUM generator starts analyzing the target method. The *Type* parameter of the corresponding registration call denotes the type of dependency, e.g., *java.lang.String*. The *Node* parameter of the registration method denotes the GROUM node that defines the dependency, e.g., the *EntryPoint* node for a parameter of the target method. If, for example, some method is called that uses a dependency, one of the *modelXXXUsage* methods is called, depending on if it is used as receiver object or as argument of that call. The corresponding *modelXXXUsage* method is responsible for inserting the actual representation of the dependency for the specific GROUM model. For example, if the *modelReveiverUsage* method is called for the *advanced-edges* implementation, and the dependency links to a parameter of the target method, a parameter-edge from the registered node – the *EntryPoint* node – to the given target node is inserted into the GROUM. For a comprehensive understanding of the interface, we suggest looking into one of its implementations.

5.1.2 GROUM Input/Output

The previous GROUM model is serialized as a general DAG in the DOT-graph[1] format without any distinction between node and edge types and, thus, without any information other than the general graph structure. Therefore, a new parser and serializer which keep all meta information of the GROUMs for the inlining process are needed. This is necessary since the GROUM generation and the inlining steps are separated, allowing for more flexibility. The new serializer and parser enable the toolchain to save and load existing GROUMs at any time, and thus also allow for further treatment of the GROUMs after the generation process, e.g., GROUM optimizations as in Section 5.1.3. Note that for pattern mining or anomaly detection, the general graph structure, i.e., labeled nodes that are connected by labeled edges, is sufficient. However, the new GROUM format only augments the old representation and is fully compatible with the format expected by the other parts of the toolchain.

For serialization, a GROUM object is converted to its DOT-graph representation, containing meta data of its nodes and edges, e.g., signature of the method call modeled by an action-node, as node-/edge-attributes as proposed by the DOT-graph language. For example, Listing 5.1 shows the DOT-graph representation of an action node that models the call to a method *Example.foo*. While only the *label* attribute is of interest for the processing steps working on general graphs, the format also contains additional meta information such as the *shape* and *style* properties, used for visualization, or the actual method signature of the called method, which is important for the inliner. Both, the parser and the serializer allow for arbitrary extensions of the model. The *Node* and *LabelledEdge* classes, which are superclasses of each node- or edge-type used in the GROUM model, declare methods that are used by the parser and serializer for parsing/serializing the specific node or edge type. When implementing new node/edge types, its sufficient to implement the proposed methods to handle the attributes of interest, for the serializer and parser to work.

```
[label="Example.foo" shape=box style=rounded nodeType="grouminer.groum.sit.nodes.
    ActionNode" signature="<Example: void foo()>" callee="foo" clazz="Example"];
```

Listing 5.1: Serialized action node.

The GROUM parser employs the DOT-graph parser provided by the *ParSeMiS* framework, which already supports arbitrary node- and edge-attributes. However, the support for graph-attributes, as proposed by the DOT-graph language, is not implemented in *ParSeMiS*' DOT-graph parser. Since GROUMs need to carry the signature of the method it models, the parser had to be extended for this feature.

[1] http://www.graphviz.org/Home.php

5.1.3 GROUM Optimizations

Since the GROUMs are saved as general graphs before the implementation of a custom serializer/parser, it was not possible to recover their internal structure besides the general graph structure, i.e., no meta data like node types. Thus, it was not possible to make modifications on the GROUMs that consider model-related data after the generation step (or, more precisely, after the serialization at the end of the generation step). Nevertheless, in some cases one might want to modify or optimize GROUM databases after the generation step to, for example, obtain other results during mining or anomaly detection, or to compare slightly different versions of a GROUM. For instance, the pattern mining requires edge labels but not the edge types, thus all edges are equivalent besides the difference in label. One might want to eliminate redundant edges introduced by the *advanced-edges* model, i.e., two edges from one node to another, or maybe even the labels of some edges to get a more abstract view of the GROUM. We introduce *GROUM optimizations* as means of implementing such modifications to existing GROUMs.

The toolchain was extended to contain an additional unit, which can be invoked to run an optimization on a set of GROUMs. An optimization can be provided by implementing the *IGroumOptimizer* interface (*grouminer.optimization.IGroumOptimizer*). The interface allows for arbitrary modification a developer might aim for. An implementation for collapsing redundant edges between two nodes and the removal of entry/exit nodes are provided, which both might be useful before running the pattern miner on a set of GROUMs. Furthermore, we also provide an implementation that allows for arbitrary chaining of optimizations, for the case that one wants to conduct multiple modifications successively.

5.2 Inlining Infrastructure

This section contains detailed information about the implementation of the inliner and its components.

Figure 5.2 shows an overview of the most important components of the inlining infrastructure and their coupling. The core of the whole inlining infrastructure is the *GroumInliner*. It is responsible for converting a given intraprocedural GROUM – the master – into its interprocedural representation. By using a subclass of the *AbstractInlineBuilder* class, it abstracts from the actions that have to be conducted when working with a specific GROUM model (Section 4.3 gives a comprehensive summary of such actions for the *advanced-edges* model). This is achieved by modeling the general traversal of a given master, deciding when to inline, and invoking the specific *InlineBuilder* implementation to do the actual inlining for a given node or edge. For the decision if an action node should be inlined, the *GroumInliner* utilizes implementations of the *ILookupProvider* and *IInlineHeuristic* interfaces. The former is used to look up a GROUM that is a valid representation of the given node, e.g., a GROUM with the exact same signature as used in the call. If so, the latter is used to evaluate if the looked-up GROUM should be inlined, based on various contextual information. After the inlining process, the *GroumInliner* provides an instance of the *InlinedGroum* class, which is a subclass of the original *Groum* class and is the corresponding interprocedural representation of the given master. During the whole inlining process, the *InlineBuilder* keeps track of the inlined GROUMs by building an inline tree, which can be accessed through the *InlinedGroum* class for further processing.

5.2.1 InlineBuilder

The *InlineBuilder* is responsible for building GROUMs, i.e., keeping track of nodes and edges as well as building the final *InlinedGroum*, and executing the actual inlining process for a specific inlinee and

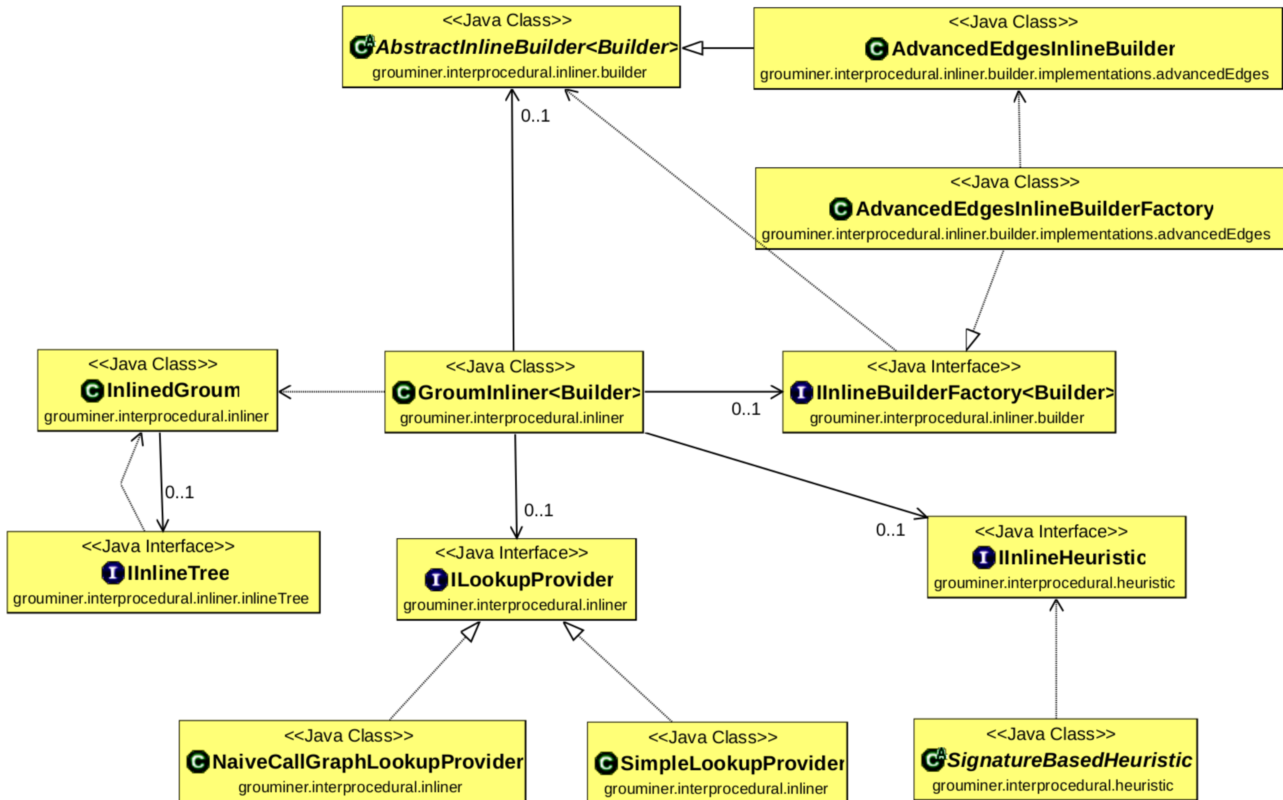

Figure 5.2.: Inlining infrastructure overview.

GROUM model, i.e., connecting nodes of the master and its inlinees in the corresponding manner. Figure 5.3 shows the public interface of the *AbstractInlineBuilder* as used by the *GroumInliner*. The builder is designed to model either a single GROUM-node, or a complete GROUM. This allows the *GroumInliner* to handle inlinees, i.e., GROUMs that should replace nodes of the master, and non-inlinees in the same way. The builder provides some helper methods that can be used to determine if it models an inlinee (*isInlinee()*) or a method without control flow other than from an entry- to an exit-node (*hasNoActionNodes()*), which is useful for the implementation of its *inlineEdge(...)* and *inlineNode(...)* methods. The *GroumInliner* keeps an instance of the *GroumBuilder*, the master-builder, used to build the interprocedural representation of the original master. This is done by constructing a builder for each node of the original master, either representing a whole GROUM – the inlinee – or just a single node and calling the *inlineNode(...)* method on the master-builder for each of those child-builders. If a child-builder contains a single node, it is just inserted into the master-builder. If it contains an inlinee, the implementation depends on the model and should be implemented by a corresponding subclass, e.g., for the *advanced-edges* model, everything except entry- and exit-nodes is added to the builder. For each edge of the master, the *GroumInliner* further on calls *inlineEdge(...)* on the master-builder, passing the child builders for the corresponding source and target nodes of the original edge, and the edge itself. The specific *AbstractInlineBuilder* implementation then has to handle the correct insertion of that edge, which depends on the used GROUM model. Generally speaking, different procedures have to be conducted when inserting an edge between any combination of inlinee and non-inlinee, i.e., two non-inlinees will, most of the time, be connected with the exact same edge as in the original GROUM, while inserting a parameter edge from a non-inlinee to an inlinee will possibly lead to multiple new edges. Note that the *InlineBuilder* is also responsible for handling empty methods and loop-throughs, as described in Section 4.3.

We provide an implementation for the *advanced-edges* model as described in Section 4.3.

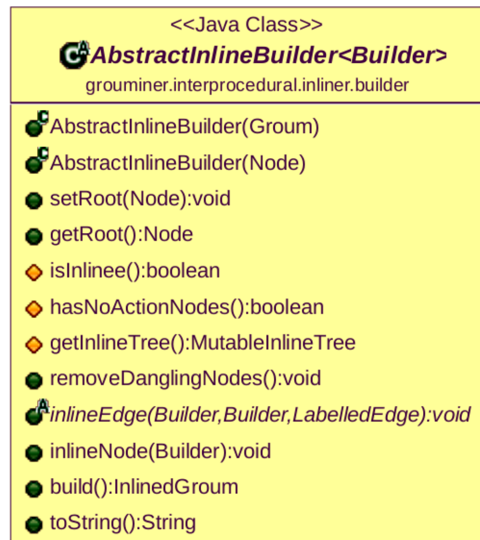

Figure 5.3.: Class diagram of the *AbstractInlineBuilder*.

5.2.2 GroumInliner

As already mentioned, the *GroumInliner* class abstracts from the specific implementation for inlining nodes and edges of a given model. Furthermore, the decision, if a node should be expanded, and which GROUM should be used as replacement, is provided by an implementation of the *IInlineHeuristic* and *ILookupProvider* interfaces. The *GroumInliner* is responsible for coordinating the whole inlining process, i.e., traversing through the master and recursively through every inlinee, going down the inline hierarchy. It employs depth-first traversal for both proceeding through a given GROUM and proceeding through the inline hierarchy. Figure 5.4 shows the public interface of the *GroumInliner* class. We suggest using the provided *GroumInlinerFactory*, which offers several configurations and helper methods for building an instance of the inliner. The inlining process can be initiated by calling the *inline(...)* method, providing the master that should be translated to its interprocedural representation according to the inliner's configuration.

Algorithm 5.1 summarizes the general inlining procedure as pseudo-code. The algorithm consists of three major procedures that traverse through the given master:

INLINE is analogous to the public *inline(...)* method as shown in Figure 5.4. The method does nothing but invoking the *PRIVATEINLINE*-procedure with the given master (M) and, afterwards, builds the *InlinedGroum* with help of the returned *InlineBuilder* (B_M).

PRIVATEINLINE is responsible for starting the inlining process on a given GROUM and is used to traverse through the inline hierarchy (line 18). It generates a new instance of the *InlineBuilder* (B_M) for the given GROUM. Note that no nodes or edges are added in this step, the master-builder inherits certain meta data of the master, i.e., name and signature. Furthermore, it starts the inlining process for each of the masters root-nodes, e.g., entry- and constant-node for the *advanced-edges model*, by invoking the *INLINENODE*-procedure. After finishing the inline process, dangling nodes, i.e., nodes which are not connected to the rest of the graph, are removed from B_M. Dangling nodes are

introduced during the inlining if a master does not use dependencies returned by an inlinee, e.g., callees may return constants that are not used by the caller.

INLINENODE constructs an *InlineBuilder* for a given node n and implements the depth-first traversal of the master. If n was already inlined, the corresponding builder is returned (line 14). If not, the *LookupProvider* is asked to provide an inlinee for the given node. If there is a suitable inlinee for n, and the given inline heuristic allows to go deeper in the hierarchy, *PRIVATEINLINE* is called recursively with the provided inlinee, rendering it the new master on that level. If n does not allow for an inlining, a new *InlineBuilder*, containing only the node, is constructed. In either of the cases, the newly generated builder for n, B_n, is passed to the *inlineNode*method of B_M, which behaves as described in the previous section (line 22). After adding B_n to B_M, the process has to be repeated for each successor of n until n has no more successors, i.e., n is an exit node. From there on, the method invokes B_M's *inlineEdge(...)* method for each edge between the current node and its successors, passing the corresponding builders and the edge itself. The inlining stops if every node and edge of the master has been traversed.

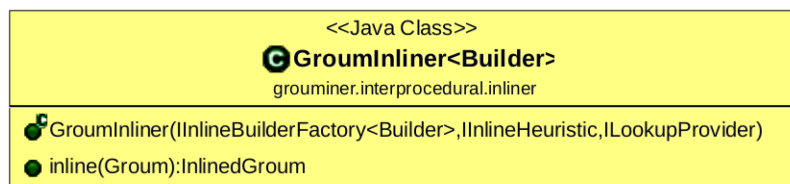

Figure 5.4.: Public interface of the *GroumInliner*.

5.2.3 LookupProvider

The *ILookupProvider* interface is a simple means for providing inlinees for a given GROUM-node. For this purpose, it contains only one method that has to resolve an inlinee based on the current master and the node of interest (see Figure 5.5). The *LookupProver* is invoked in line 17 of the inlining algorithm (Algorithm 5.1). Two implementations of the interface are provided:

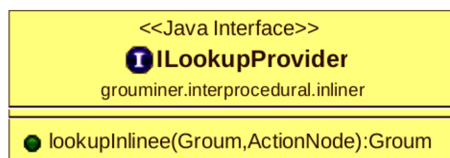

Figure 5.5.: *ILookupProvider* interface.

SimpleLookupProvider is given a set of GROUMs as viable inlinees and compares the exact signature of the call, as modeled by the given action-node, with the signatures of the provided GROUMs. The *SimpleLookupProvider* returns the matching GROUM, if available. The provider does not resolve calls to virtual methods since it has no means to determine the exact receiver object of such a call.

NaiveCallGraphLookupProvider has to be provided with a set of GROUMs and a call-graph. The provider uses the call-graph to resolve calls to one random possible receiver and uses its signature for the lookup. Thus, the provider is an extension of the *SimpleLookupProvider*, but additionally resolves virtual callsites to exactly one of the possible target inlinees.

Algorithm 5.1 Pseudo-code of the inlining procedure.

```
Input:   Master GROUM as M
Output:  Intraprocedural version M_I of M
 1: function INLINE(M)
 2:     return BUILD(PRIVATEINLINE(M))
 3: end function

Input:   Master GROUM M
Output:  InlineBuilder B_M for M
 4: function PRIVATEINLINE(M)
 5:     B_M ← NEWBUILDER(M)
 6:     for all r ∈ ROOTSOF(M) do
 7:         INLINENODE(B_M, r)
 8:     end for
 9:     REMOVEDANGLINGNODES(B_M)
10:     return B_M
11: end function

Input:   GROUM-node n, master-builder B_M
Output:  InlineBuilder B_n for n
12: function INLINENODE(B_M, n)
13:     if B_n exists then
14:         return B_n
15:     else
16:         i_n ←LOOKUPINLINEE(n)
17:         if  i_n ≠ null AND SHOULDBEINLINED(i_n) then
18:             B_n ← PRIVATEINLINE(i_n)
19:         else
20:             B_n ← NEWBUILDER(n)
21:         end if
22:         BUILDER_INLINENODE(B_M, B_n)
23:         for all suc ∈ SUCCESSORSOF(n) do
24:             B_suc ← INLINENODE(B_M, suc)
25:             for all  e ∈ EDGESBETWEEN(n, suc) do
26:                 BUILDER_INLINEEDGE(B_M, B_n, B_suc, e)
27:             end for
28:         end for
29:     end if
30:     return B_n
31: end function
```

5.2.4 InlineHeuristic

The implementation of an inline heuristic has to conform to the *IInlineHeuristic* interface (Figure 5.6). The inliner is provided with an implementation of the interface and calls the *reset()* method of the given heuristic before every new inlining process, i.e., when the public *inline(...)* method is called on the corresponding inliner. Furthermore, the *newLevel(...)* method is called every time the inliner reaches a new level of the inlining hierarchy. It is also called in the very first level, i.e., before traversal of the master. The method has to provide a new instance of the hierarchy implementation for the given level. This enables the implementation to alter its state based on the current inline level, while not affecting higher levels (or inlinees on other inline-paths). The main task of the heuristic – deciding if a GROUM should be inlined –, however, is conducted in the *evaluate(...)* method. The method has to evaluate if a GROUM, as provided by the *LookupProvider*, should be inlined. Thus, an inline heuristic is able to decide if a GROUM should be inlined based on its predecessing inlinees/masters, the target, and the information that is present during instantiation of that heuristic.

Implementations for the heuristics stated in Section 4.3.1 are in the *grouminer.interprocedural.heuristic* package. For chaining multiple heuristics, the *HeuristicChain* class can be used. The class provides functionality for connecting multiple inline heuristics in either conjunctional or disjunctional units. The class itself is an implementation of the interface and, therefore, allows connecting several inline heuristics to arbitrary logical expressions.

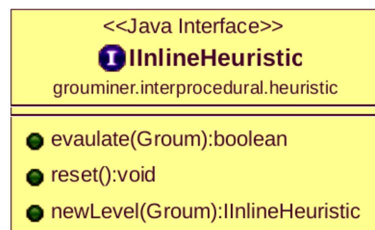

Figure 5.6.: *IInlineHeuristic* interface.

5.3 Call-Graphs

Since the GROUM generation process is decoupled from the inliner, means to de-/serialize the call-graphs are needed. We employ the *ProBe*[2] framework for this purpose. *CryptoMiner* did not handle call-graphs before, thus, its interface was extended to allow for passing-through arguments to the GROUMbuilding *Soot* analysis. This allows for using the call-graph algorithm and configuration which is most useful for the specific use-case. Since the call-graphs can grow very large and the GROUM generator already supports filtering for methods of interest, e.g., only methods that use the JCE, the possibility to filter the call-graphs for those methods was also added. Helper methods for de-/serialization can be found in *grouminer.util.CallGraphUtils*.

5.4 Test Cases

The inliner comes with a comprehensive test-suite, containing several test cases for the general functionality of the inliner, heuristics and the inline tree. Testing of the inliner is done by providing Java source

[2] `https://github.com/karimhamdanali/probe`

code for a target intraprocedural master, its inlinees, and an interprocedural version of that master. The inlined version of the master will then be compared with the GROUM as generated for the provided interprocedural version. Furthermore, test cases are provided that focus on the potential of the inliner in regards to the reduction of false positives during the anomaly detection process. These rather test the actual usefulness of the inliner than its correctness. Results for the conducted tests can be found in Section 6.2.

5.4.1 Interprocedural Pattern Usage Filter

To test the potential of the inliner, synthetic test cases were undesired. Therefore, we provide another tool that uses the *Soot* framework, to filter for methods which potentially contain interprocedural usage of patterns. If provided with a set of applications and the class or package of interest, the tool delivers a list of masters and inlinees that are likely to contain interprocedural object usage of the specified types.

To obtain those lists, the tool exploits the following facts: If an object-usage is distributed among several methods – an interprocedural object usage –, the objects have to be propagated to or from those methods, i.e, if method *A* calls method *B* to do something on an object, the object has to be passed to *B* in some way. Since *Cryptominer* does not handle class fields by now, the only way to exchange objects beyond the scope of a method, is using the method's return value or parameters. By filtering for methods whose signatures (only considering return and parameter types) contain certain classes or packages of interest, i.e., classes/packages which are modeled by the patterns, it is likely to find methods that would profit from an inlining. With the help of the call-graph, it is easy to find all callers of those callees. These are also likely to be the root (or an intermediate) method of an interprocedural pattern-usage, i.e, they separate object usages over several callers, and thus valuable masters.

6 Evaluation

The evaluation of the interprocedural approach is conducted in two steps: Firstly, the general viability of the inlining is tested by applying it on a test set that only contains methods which are likely to profit from it, i.e., methods that contain parts of patterns and calls to methods that also contain parts of those patterns. Thereby, it is possible to prove the general ability of the approach for reducing false positives in the anomaly detection step. Furthermore, we have conducted experiments on a much larger test-set to show the applicability of the approach on more general data set.

6.1 Setup

We investigated 102 Android applications obtained from the *AppCrawl*[1] project and filtered for JCE usage, leaving 50 relevant application. Out of those, again only methods using the JCE are extracted and converted to the GROUM model, resulting in a test set of 1012 intraprocedural GROUMs. From this set, seven patterns of JCE usage are mined with a minimum frequency of 11, i.e., an object usage has to occur at least 11 times in the test set to be considered a pattern. Invoking the anomaly detector with a minimal overlap of 30% between pattern and target results in an overall sum of 208 violations. A manual investigation of 64 of those violations yielded that 70% of them are caused by the intraprocedural nature of the GROUMs. We use the same test-set to evaluate how much of those violations can be eliminated by the presented interprocedural approach.

6.2 Constructed Tests

To test the potential of the interprocedural model, the test set is filtered for GROUMs that are likely to profit from it. Therefore, the *Interprocedural Pattern Usage Filter* (Section 5.4.1) is used to obtain only GROUMs from the test set which contain the *javax.crypto.Cipher* class in their signature and their callers. The filtering results in 26 strict-callers, i.e., not called by the others, 12 intermediate-callers and 170 strict-callees. Looking at the numbers, it is striking that the amount of strict-/intermediate-callers is very low compared to the amount of strict-callees. An investigation of the stated methods shows that 157 of the callees are never called in the test set, leaving us with 25 $(170 + 12 - 157)$ possible inlinees. It seems that a lot of these unused methods are helpers, i.e., provide pre-configured instances of the *javax.crypto.Cipher* class, from the *Bouncy/Spongy Castle* cryptography libraries, which are common service providers for the JCE. While most of the applications ship a version of one of these providers, the largest fraction of their API is left unused. Since all of those unused methods contain parts of *javax.crypto.Cipher* object usage, we expect those to produce high numbers of false positives in the experiments. A possible solution to mitigate those false positives is the exclusion of such libraries from the test set. However, an investigation of the callees that are actually used in the test set yields that most of those are helper methods provided by the libraries as well. They are invoked in application code to deliver pre-configured *javax.crypto.Cipher* objects; thus, an inlining of those might lead to a reduction of false positives found in application code.

[1] `http://appcrawlr.com/`

Table 6.1.: Results for the constructed test set

				#violations			
min-overlap	LookupProvider	#inlinees	#inlinings	intra	inter	diff	reduction
30%	Simple	22	63	48	47	1	2.08%
	NaiveCallGraph	23	80	48	35	13	27.08%
	Provider-diff	1	17	-	-	12	25%
20%|25%	Simple	22	63	141	91	50	35.46%
	NaiveCallGraph	23	80	141	45	96	68.09%
	Provider-diff	1	17	-	-	45	32.63%
15%	Simple	22	63	149	105	44	29.53%
	NaiveCallGraph	23	80	149	59	90	60.40%
	Provider-diff	1	17	-	-	46	30.87%
0%	Simple	22	63	184	141	43	23.37%
	NaiveCallGraph	23	80	184	85	99	53.80%
	Provider-diff	1	17	-	-	56	30.34%

For the experiment, two versions of the inliner are configured, one using the *SimpleLookupProvider* and the other using the *NaiveCallGraphLookupProvider*. Both are configured to use the *AlwaysInlineHeuristic*, since the set of inlinees is designed to be profitable when inlined and thus other heuristics would just restrict the inlining capabilities for the experiment. The anomaly detector is provided with the same seven patterns as in our previous experiments, having only the intra-/interprocedural strict-callers as targets. Table 6.1 shows the results of the experiment. The *min-overlap* column specifies the minimal overlap between pattern and target as chosen for the individual test run. For each run, the results for both of the lookup providers are visualized in one row, followed by the difference of both.

SimpleLookupProvider

Considering the results of a minimal overlap of 30%, the number of eliminated violations (reduction rate of 2.08%) is unexpectedly low for the *SimpleLookupProvider*, although 22 of the 25 possible inlinees are actually inlined. When comparing those numbers with the results for lower minimal overlaps, it can be noticed that the number of inlinees and inlinings does not change, while the actual violation reduction rate goes up to 35.46% with an absolute reduction of 43-50 violations. So why do the results for a minimal overlap of 30% vary so much from the others? Considering the fact that the number of inlinees/inlinings is consistent across all experiments and the absolute violation reduction does not vary that much for a minimal overlap of 0-25%, we assume that the absolute number of eliminated false positives stays more or less the same across all experiments. Based on that assumption it gets obvious that the interprocedural model introduces violations that were not detected in the intraprocedural model (if using a minimal overlap of 30%). In the intraprocedural model, object usages which are separated among several methods oftentimes do not exceed the minimal overlap threshold. Thus, these go unnoticed by the detector. Since in the interprocedural model those usages are joined together, the number of overall occurrences grows. For instance, if each of two targets and a pattern have one node in common and the pattern has four nodes, a threshold of 30% will yield no occurrences in the targets (since one of four nodes is an overlap of 25%). When building an interprocedural model of those two targets, however, the object usages in these are combined and thus exceed the threshold. Hence, an occurrence is found after the inlining which was ignored before. Considering this, the interprocedural approach seems to

eliminate about the same number of violations as it introduces when using a threshold of 30%. Nevertheless, this can be seen as an improvement against the threshold approach since the threshold just hides violations that are not very expressive in the intraprocedural context, while the interprocedural model actually resolves those violations. In other words, using the threshold helps hide violations due to interprocedural usages when working with the intraprocedural model, while using the interprocedural model can resolve those violations (instead of hiding them) and, thus, yield actual true positives that go unnoticed with the intraprocedural approach.

Since the test set was designed to contain methods that are likely to profit from the inlining, the reduction of violations - and thus false positives - was expected to be close to 100%. However, using the *SimpleLookupProvider* results in a maximum of 35.46% eliminated violation. A manual investigation of the violations yielded four main causes for this:

Incompleteness of patterns: While the test set contains only methods that have usage of the *javax.crypto.Cipher* class, the used pattern models just some of the use cases of the class. For instance, consider the pattern in Figure 3.4: While a lot of targets contain the pattern after inlining, there are still several other instances of similar patterns (like using a *CipherOutputStream* instead of invoking the *doFinal(...)* method manually). So while occurrences of the pattern are found and marked as violations, those are false positives which are caused by the incompleteness of the patterns. The inliner is not able to eliminate such false positives since they are not caused by the separation of patterns but rather due to the incompleteness of those. It has to be stated that providing an implementation of the *CipherOutputStream* might actually eliminate the false positive for this concrete example, but the general problem resides.

Field usage: Since *Cryptominer* does not allow for handling class fields, these are omitted in the model. However, sometimes patterns are distributed between several methods of a class where the object of usage is shared by a class field. Consider the case where some method m instantiates such a class C and afterwards calls a method *encrypt()* on the instance. C itself builds a *Cipher* object in its constructor and assigns it to one of its fields. The field is then accessed in the *encrypt()* method to call *doFinal(...)* on the *Cipher* object. If every call in m is inlined, an interprocedural version of m (m_I) which contains both, the internals of the constructor and the *encrypt()* method of C, is generated. However, since neither the constructor returns the *Cipher* object, nor the *encrypt()* method takes it as a parameter, there is no means to determine if those are related in the current GROUM model. Thus, no edge between the *Cipher* object's usage in the constructor and in the *encrypt()* method can be inserted into m_I. Hence, if the pattern in Figure 3.4 is used for detection, the detector will detect multiple violations (depending on the minimum overlap) of that pattern in m_I.

Virtual callsites: The simple lookup approach does not allow the resolving of callsites for which the exact type of the receiver is statically unknown. We observed several of such callsites in the test set. All of those seem to invoke some helper of the JCE library to construct a *Cipher* object that is then further on used by the application. Multiple implementations of the helper exist.

Incomplete usage: We observed that even after the inlining some object usages remain incomplete, e.g., for the pattern in Figure 3.4, the *doFinal(...)* method is still missing in some of the interprocedural GROUMs. As already mentioned, a lot of the methods in the test set are helpers provided by libraries like *BouncyCastle*, this also holds for some of the masters. Even after the inlining, the interprocedural GROUM is missing some object usages that are expected do be found in application

code using the modeled helpers. Since the helpers are never used in the test set, the inliner stands no chance to eliminate those incomplete usages.

NaiveCallGraphLookupProvider

While the other cases can not be handled by the inliner, the resolution of virtual callsites can be. To test the potential of resolving those, we came up with the previously explained naive approach. Again, consider Table 6.1: The *Provider-diff* column shows the difference between using the *SimpleLookupProvider* and the *NaiveCallGraphLookupProvider* for the given experimental setup. It is striking that there is only one additional GROUM inlined compared to the simple lookup approach, yielding a violation reduction of an additional 25-32.63%. Considering the average amount of inlinings by inlinee of $2.52 = 63/25$ for the simple approach, it is remarkable that this single GROUM has so many inlinings (17). Furthermore, the *NaiveCallGraphLookupProvider* yields an additional violation reduction of 45-56 (30.34-32.64%) violations for a minimal overlap of 0-25%, which is about twice as much compared to the *SimpleLookupProvider*. For a minimal overlap of 30%, the difference in violation reduction is even 13-times as much. This shows that at least 12 of the exposed violations (caused by exceeding the minimum overlap threshold after inlining) can be eliminated by the interprocedural model with a resolution of callsites, rendering the approach even more superior in contrast to simply ignoring of such cases.

Resolving virtual callsites to one single possible target yields a reduction of false positives of up to 68.09%. This can be seen as an upper bound for the experiment. A more sophisticated approach for handling virtual callsites would need to inline each of the possible targets and thus potentially introduce additional violations. This might not necessarily be a bad thing, e.g., if one version of such an inlinee builds a true positive with the master which another does not. Nevertheless, since the number of true positives is comparably low, it is much more likely that such an inlinee introduces one of the other two stated causes (incompleteness of patterns, class fields) for false positives in the test set, which are much more frequent. While this alone might lead to a much higher number of – redundant – violations, due to the high amount of inlinings, the problem gets even worse: One version of every master has to be created for all potential callees of any combination of virtual callsites (for example, four masters if there is one virtual callsite with four possible targets, 16 if there are two virtual callsites with four possible targets each). This would lead to an even bigger amount of redundant violations (false positives as well as true positives). However, the problem of introduced redundant violations is also observable for the simple lookup approach. Means to identify and eliminate such redundancies have to be found. Nevertheless, even with the introduction of such redundancies, the *NaiveCallGraphLookupProvider* yields a very promising reduction of false positives by up to 69.09%.

The absolute violation reduction for the *SimpleLookupProvider* decreases when the minimal overlap goes below 20%. While having a too high threshold reveals – ignored – occurrences, a too low threshold introduces more redundant violations; either of the two compensate the number of reduced violations in some way. In other words, if the threshold is set low, even overlaps of just one node are considered as occurrence and thus also reported as violations (there are no one-node patterns); if these are inlined at several callsites, the detected violations will be detected at all of these callsites if they were not eliminated by the inliner. It seems that a minimal overlap of 20-25% yields the best results in the tradeoff between hidden and redundant violations. However, inspecting the values for the *NaiveCallGraphLookupProvider* shows that the absolute number of violation reduction even increases towards a minimal overlap of 0%. This is the case because some of the redundancies are violations that can be resolved by the handling virtual callsites.

Table 6.2.: Results for the whole test set with a minimal overlap of 30% using the *SimpleLookupProvider*

			#violations			
Heuristic	#inlinees	#inlinings	intra	inter	diff	reduction
AlwaysInline	232	568	208	223	-15	-7.21%
ReturnsOrGetsPackage(javax.crypto)	170	429	208	223	-13	-6.25%
ReturnsOrGetsClass(javax.crypto.Cipher)	56	140	208	215	-7	-3.37%
IsInSameClass	74	172	208	207	1	0.48%
IsInSameOrSubPackage	214	488	208	225	-17	-8.17%
LevelBased(1)	232	512	208	233	-25	-12.02%
LevelBased(2)	232	574	208	227	-19	-9.13%
LevelBased(3)	232	575	208	225	-17	-8.17%
LevelBased(4)	232	568	208	223	-15	-7.21%
LevelBased(5)	232	568	208	223	-15	-7.21%

6.3 Real World Tests

We also conducted tests on the whole test set as described in Section 6.1. Therefore, we used several combinations of the inline heuristics stated in Section 4.3.1 for the tests. The *NoRecursionHeuristic* is used in addition to every heuristic combination, to not run the inliner in an infinite loop. While we conducted experiments for 53 different combinations of heuristics, we only show, for each test run, the ones that yield the most interesting insights. A comprehensive listing of all test results can be found in Appendix A. In contrast to the constructed tests, where only a few dedicated masters were chosen, the inliner is invoked for every GROUM in the test set. To avoid introducing additional redundant violations during the anomaly detection, we remove inlined GROUMs from the set before conducting the anomaly detection.

Table 6.2 shows the results of the anomaly detection for a minimal overlap threshold of 30% using the *SimpleLookupProvider*.

Considering the results of this test run, several things are noticeable: Firstly, it is conspicuous that the number of violations rises compared to the intraprocedural model. As we have already observed during the constructed tests, this is caused by the introduction of violations which were hidden by the high minimal overlap threshold. Furthermore, the heuristics that restrict inlining (*IsInSameClass* and *ReturnsOrGets(javax.crypto.Cipher)*) have the lowest introduction of new violations. The *IsInSameClass* heuristic does even provide an improvement, albeit a very small one. The reason for this is the comparably low amount of newly introduced (redundant) violations due to a lower number of inlinings. Moreover, comparing the results for the *AlwaysInline* and the *IsInSameOrSubPackage* heuristics, it is observable that 92% of inlinees seem to be in the same or a sub-package as the master and, similarly, 86% of inlinings seem to happen between classes in the same or a subpackage. Additionally, it stands out that using the level-based heuristics leads to the same amount of inlinees as always inlining while the number of inlinings stagnates when reaching the fourth level. This indicates a maximum inline hierarchy of four in the test set. Since this holds for every test run, the data for level four and five will be omitted in the following visualizations. Furthermore, this shows that inlining only one level of the hierarchy already leads to the usage of all possible inlinees with 90% of the maximum number of inlinings. However, proceeding deeper in the inline hierarchy seems to yield an improvement of violation reduction by up to 40% with only 10% more inlinings compared to the first level. One might notice that the amount of inlinings is higher for inlining two or three levels compared to always inlining. This seems a bit coun-

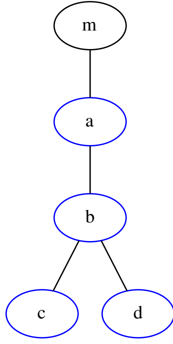

Figure 6.1.: Three levels

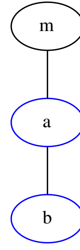

Figure 6.2.: Two levels 1

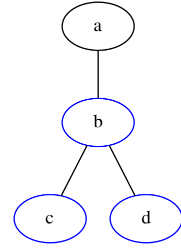

Figure 6.3.: Two levels 2

Table 6.3.: Results for the whole test set with a minimal overlap of 30% and the *NaiveCallGraphProvider*

			#violations			
Heuristic	#inlinees	#inlinings	intra	inter	diff	reduction
AlwaysInline	273	947	208	254	-46	-22.12%
ReturnsOrGetsPackage(javax.crypto)	198	697	208	242	-34	-16.35%
ReturnsOrGetsClass(javax.crypto.Cipher)	63	227	208	232	-24	-11.54%
IsInSameClass	77	180	208	207	1	0.48%
IsInSameOrSubPackage	229	553	208	225	-44	-12.02%
LevelBased(1)	273	735	208	252	-44	-21.15%
LevelBased(2)	273	936	208	265	-57	-27.40%
LevelBased(3)	273	954	208	250	-42	-20.19%

terintuitive at first. Consider the inline trees in Figures 6.1 to 6.3. Blue nodes are inlinees and black nodes are master GROUMs. When inlining three levels, the master (m) has four possible inlinees: a, b, c and d. When inlining only two levels, there are two possible masters considering the inline hierarchy in Figure 6.1: m, which has a and b as inlinee, and a which has b, c and d as inlinees. Thus, we get four possible inlinings for three levels of inlining and and five for two levels of inlining. This behavior appears if the inline depth is constraint to a value that is smaller than the maximum inline depth (with the exception for inlining only one level). Since the maximum inline depth is four in our test set, we can observe higher amounts of inlinings when only inlining two or three levels.

Table 6.3 shows the results of the anomaly detection with a minimal overlap threshold of 30% and the *NaiveCallGraphProvider*. Most of the observations for the previous run still hold here. However, there are a few differences: A much higher introduction of new violations compared to the *SimpleLookupProvider* occur. This is not unexpected since even more hidden violations are revealed when processing more inlinees. Comparing the always inline cases, there are 15% more inlinees and 60% more inlinings with the *NaiveCallGraphLookupProvider*. The high amount of newly introduced inlinings compared to the low amount of newly introduced inlinees supports the observations made for the constructed test cases. Furthermore, contrary to the results for the *SimpleLookupProvider* run, the amount of introduced violations fluctuates dependent on the maximum level of inlining. Having a look at the number of inlinings for those cases suggests that when inlining just one level, a significantly lower amount of virtual callsites is processed compared to inlining two or more levels. Also, the resolution of those callsites seem to be most beneficial when reaching the third level for the test set.

To mitigate newly introduced violations that were hidden by the high minimal overlap threshold, we employ a post-filtering step which filters out interprocedural GROUMs that produce higher (or equal)

Table 6.4.: Results for a minimal overlap of 30%, the *SimpleLookupProvider* and post filtering

			#violations			
Heuristic	#inlinees	#inlinings	intra	inter	diff	reduction
AlwaysInline	28	53	208	191	17	8.17%
ReturnsOrGetsPackage(javax.crypto)	17	31	208	192	16	7.69%
ReturnsOrGetsClass(javax.crypto.Cipher)	16	30	208	192	16	7.69%
IsInSameClass	6	8	208	192	16	7.69%
IsInSameOrSubPackage	12	23	208	195	14	6.25%
LevelBased(1)	5	10	208	194	14	6.73%
LevelBased(2)	23	44	208	192	16	7.69%
LevelBased(3)	26	51	208	191	17	8.17%

Table 6.5.: Results for a minimal overlap of 30%, the *NaiveCallGraphLookupProvider* and post filtering

			#violations			
Heuristic	#inlinees	#inlinings	intra	inter	diff	reduction
AlwaysInline	67	175	208	151	57	27.40%
ReturnsOrGetsPackage(javax.crypto)	46	137	208	151	57	27.40%
ReturnsOrGetsClass(javax.crypto.Cipher)	36	109	208	151	57	27.40%
IsInSameClass	6	8	208	192	16	7.69%
IsInSameOrSubPackage	15	29	208	196	1512	5.77%
LevelBased(1)	11	21	208	190	18	8.65%
LevelBased(2)	43	118	208	162	46	22.12%
LevelBased(3)	65	173	208	151	57	27.40%

amounts of violations than their intraprocedural version and all inlinees together. After the filtering, the intraprocedural versions of the filtered GROUMs are re-added to the test set, to not sophisticate the results.

Again, the minimal overlap is set to 30%. Table 6.4 shows the filtered results for the *SimpleLookup-Provider* and Table 6.5 for the *NaiveCallGraphLookupProvider*, respectively.

Looking at the filtered results for the *SimpleLookupProvider*, a violation reduction of 6.25-8.17% is observable. Compared to the results for the same configuration without the post-filtering, the improvement is quite noticeable. However, the number of inlinees and inlinings is very low in comparison. This shows that just a fraction of possible inlinings are able to compensate the newly introduced violations which occur due to exceeding the minimal overlap threshold. Again, we expect a high number of redundant violations to hamper better reduction rates. It is striking that constraining the inlining with special heuristics yields lower reduction rates than inlining when ever possible. We expect the filtering to mitigate most of the improvements due to constraining the inlining since only interprocedural GROUMs are kept that yield less violations than their intraprocedural analogs anyway.

For the *NaiveCallGraphLookupProvider* with filtering, much better reduction rates of up to 27.40% are observed. It seems that resolving virtual callsites is very profitable with regards to reducing violations for the test set. This supports the observations made during the constructed tests and also hold for the following test runs. Again the best reduction seems to happen when not constraining the inlining. Furthermore, while we previously observed that the largest fraction of inlinings occur in the same or some sub-package of the corresponding master, we can now see that those seem to be quite unprofitable since most of those are filtered out. The stated observations for this test run also hold for the following exper-

Table 6.6.: Results for a minimal overlap of 20% and the *SimpleLookupProvider*

			#violations			
Heuristic	#inlinees	#inlinings	intra	inter	diff	reduction
AlwaysInline	232	568	613	555	58	9.46%
ReturnsOrGetsPackage(javax.crypto)	170	429	613	566	47	7.67%
ReturnsOrGetsClass(javax.crypto.Cipher)	56	140	613	566	47	7.67%

Table 6.7.: Results for a minimal overlap of 20%, the *SimpleLookupProvider* and post filtering

			#violations			
Heuristic	#inlinees	#inlinings	intra	inter	diff	reduction
AlwaysInline	59	125	613	543	70	11.42%
ReturnsOrGetsPackage(javax.crypto)	40	91	613	558	55	8.97%
ReturnsOrGetsClass(javax.crypto.Cipher)	20	50	613	562	51	8.32%

iments, thus we will now focus on the most promising heuristics only: *AlwaysInline*, *ReturnsOrGetsClass* and *ReturnsOrGetsPackage*.

Tables 6.6 and 6.7 show the results for a minimal overlap of 20% and the *SimpleLookupProvider* with and without post-filtering. Regarding the results without filtering, it is remarkable that the amount of inlinees and inlinings is exactly the same as for the a minimal overlap of 30%. Nevertheless, the absolute and relative reduction rate is positive for the run. This strongly supports our observation of newly introduced violations when setting the the minimal overlap threshold to high. However, the overall reduction of violations is superior to using a minimal overlap of 30%, with or without filtering. An interesting aspect here is also the fact that the filtering still improves the reduction of violations between both runs. Thus, there seem to be a few interprocedural GROUMs that introduce more violations than their intraprocedural counterparts even with a threshold of 20%. We believe that this is because of the introduction of redundant violations which are caused due to the reasons stated in Section 6.2 (i.e., incompleteness of patterns, field usage, virtual callsites, incomplete usage). Furthermore, the high amount of filtered-out interprocedural GROUMs with a minimal improvement of violation reduction suggests that a lot of those produce equal amounts of violations as their intraprocedural counterparts. This might be either because no violations could be resolved by the inlining or because the resolved and newly introduced violations (due to redundancies) compensate each other. While not constraining the inlining with specific heuristics leads to better reduction rates again, this time the benefit of always inlining is even more significant.

Tables 6.8 and 6.9 show the results for a minimal overlap of 20% and the *NaiveCallGraphLookup-Provider* with and without post-filtering. Again, it is observable that while the amount of inlinees and inlinings is the same as for a minimal overlap of 30% without filtering, the relative and absolute violation reduction is much higher for this configuration. We observed the best results for this configuration which is also supported by the results for the constructed tests. Furthermore, the test run finally approves that while using a specific heuristic in some cases yields slight improvements, using no constraints for inlining does not lower the reduction rates much. Most of the time, using the *AlwaysInlineHeuristic* does yield the same or even better results than a specific heuristic. Thus, and since using a specific heuristic oftentimes leads to much lower reduction rates, we suggest not constraining the inlining at all.

Experiments with no minimal overlap threshold were also conducted. Table 6.10 shows a summary of those for the *AlwaysInline* heuristic only. It is not surprising that without filtering the violation reduction is still better than with a minimal overlap of 30%. However, using the filter produces better results for

Table 6.8.: Results for a minimal overlap of 20% and the *NaiveCallGraphLookupProvider*

			#violations			
Heuristic	#inlinees	#inlinings	intra	inter	diff	reduction
AlwaysInline	273	947	613	446	167	27.24%
ReturnsOrGetsPackage(javax.crypto)	198	697	613	444	169	27.57%
ReturnsOrGetsClass(javax.crypto.Cipher)	63	227	613	440	173	28.22%

Table 6.9.: Results for a minimal overlap of 20%, the *NaiveCallGraphLookupProvider* and post filtering

			#violations			
Heuristic	#inlinees	#inlinings	intra	inter	diff	reduction
AlwaysInline	129	409	613	433	180	29.36%
ReturnsOrGetsPackage(javax.crypto)	78	290	613	432	181	29.53%
ReturnsOrGetsClass(javax.crypto.Cipher)	44	192	613	436	177	28.87%

Table 6.10.: Results for a minimal overlap of 0%

				#violations			
Filter	Naive-cg	#inlinees	#inlinings	intra	inter	diff	reduction
false	false	232	568	1105	1069	36	3.26%
true	false	70	144	1105	1042	63	5.70%
false	true	273	947	1105	936	169	15.29%
true	true	138	428	1105	924	181	16.38%

the *SimpleLookupProvider* and the *NaiveCallGraphLookupProvider* with a minimal overlap of 30%. While there are no newly introduced violations due to exceeding the threshold after inlining for this test run, having no threshold leads to more redundancies of small – non-expressive – occurrences, as already mentioned in Section 6.2.

For none of our cases the inlining took longer than 0.28 seconds, processing all 1012 GROUMs. The maximum observed average increase of nodes for a given master is 50% with an average increase of 65% for the edges. The highest reduction of violations for the whole test set is 30.67% by using the *NaiveCallGraphLookupProvider*, filtering, a minimal overlap of 20% and a disjunctive combination of the *ReturnsOrGetsClass(javax.crypto.Cipher)* and *IsInSameOrSubPackage* heuristics.

Since we do not change the semantics of the object usages but just combine the ones that are separated over several methods, we believe that no true positives are eliminate by the approach. Considering that about 70% of all violations are false positives caused by the intraprocedural nature of the old model, the reached violation reduction of 30.67% can also be seen as a reduction of those false positives by 42.86%.

While using the ”wrong” heuristic can yield much lower reduction rates than inlining every possible call, using the ones tailored towards expected usages (*ReturnsOrGetsPackage*, *ReturnsOrGetsClass*) slightly improves the results in some cases. Furthermore, while using the filtering improves the results significantly when using high minimal overlap thresholds, it is less beneficial for lower thresholds. Therefore, we consider the choice of heuristic and the filtering less important than choosing a reasonable threshold and having a sophisticated method to resolve virtual callsites. Furthermore, we believe that tackling the stated reasons for false positives that are not caused due to the separation of patterns (Section 6.2), might greatly improve the inliner's capabilities and, thus, the overall reduction of false positives. While handling class fields by extending the GROUM generation is an applicable improvement, the elimination of violations due to the incompleteness of patterns and incomplete usages are a

general problem that can not easily be resolved. Means to detect redundant violations caused by those, and due to the resolution of multiple virtual callsite targets, have to be found. This might greatly improve the reduction of false positives when using the interprocedural model, and might eventually lead to reduction rates near to the assumed 70%.

7 Conclusion

The work presented in this thesis aims at an automatic approach for generating interprocedural Graph-based Object Usage Models (GROUMs) to reduce the high amount of false positives introduced due to the intraprocedural nature of the previous model. For this purpose, we introduce an extension of the graph model that captures additional semantics of relations between nodes of the graphs and the graphs themselves. This not only enables the application of graph-based inlining techniques on those GROUMs, but also renders the model more complete and precise in regards to its representation of the original object relationships in code. We present rules which, when applied on the extended graph model, allow for correct insertion of callee-GROUMs into caller-GROUMs to generate an interprocedural representation of such caller-GROUMs. Furthermore, we introduce heuristics that can be used and freely combined to decide which nodes should be extended for their callee-GROUM and, thus, prevent the undesired inlining of methods, such as API calls of interest. We have also presented a naive approach which resolves virtual calls that can have multiple calling targets to exactly one of the possible target is presented. To evaluate the inlining, a test set of 1012 GROUMs using the Java Cryptographic Extensions is extracted from 50 Android applications. For the same test set a false positive ratio of 70% due to interprocedural object usages was preliminary assessed. In our experiments on a selected portion of those GROUMs which are likely to profit from the inlining, *CryptoMiner* achieves violation reduction rates by up to 35.46% without handling virtual callsites and 68.09% with handling those. We assume all of those eliminated violations to be false positives. Furthermore, we conducted experiments on the whole test set which yielded violation reduction rates by up to 11.42% without handling callsites with multiple possible targets, and up to 30.67% with the resolution of those, respectively. Again, we assume all of those eliminated violations to be false positives. Based on the preliminary assessed 70% of false positives that are caused by the intraprocedural nature of the old model, a very promising reduction rate of those false positives by up to 42.86% is achieved.

In our experiment, we observed increasing numbers of false positives when conducting the anomaly detection with high minimal-overlap thresholds between patterns and targets. We identify the reason for this to be the revelation of small, incomplete object usages that are hidden by the threshold in the intraprocedural model. Inlining those leads to more complete usages that exceed the threshold and, thus, to more detected violations. While using a high threshold might be a good idea in the intraprocedural model to hide small, inexpressive object usages that are separated over several methods, the interprocedural model allows those to be combined and tested for anomalous usage, and therefore, also eliminates false positives that where hidden by the threshold but likewise reveals true positives that were hidden by it.

Limitations of the Approach

We identified four main causes that hamper the inlining of reaching the expected violation reduction rate of 70%. Firstly, the *incompleteness of patterns*, that is, patterns of correct object usage are missing in the pattern set. A correct object usage might be detected as violation because the used pattern set is missing the pattern for that usage but does contain a very similar pattern, e.g., with a one-node-difference. It is generally a non-trivial task to attack this problem since the possible combinations of object usages are enormous and, thus, having a complete pattern set might not be achievable. The

approach is furthermore limited by *incomplete usages*, where the test set contains methods that contain parts of correct object usage, but are never called. These methods are usually library methods that have to be called by application code to complete the object usage. Violations are detected in those because they miss some parts of the patterns, but since application code is responsible for calling those and, thus, completing the object usage, it is undecidable if those are true or false positives. Another reason for false positives that cannot be eliminated by the approach is the lack of the current model to capture *field usages*. Hence, it is not possible to detect relations between receiver objects of several method calls where the object is stored inside a field instead of a variable. Since those object usages cannot be connected in the model, violations are detected. Last but not least, there is the general problem of introducing *redundant violations* when inlining one GROUM into several others. Each of the GROUMs that have the same method inlined, contains all of the violations of that method after inlining. True positives and false positives are duplicated alike. While the *incompleteness of patterns* and *incomplete usages* can generally not be attacked by the inliner, the other two can. We believe that, especially, a sophisticated approach for the elimination of duplicate violations is a very promising augmentation to the inlining and may yield significantly better results.

In the following the research questions stated in Section 1.1 are answered and a proposal of future work is given:

Research Questions

- **Is interprocedural GROUM generation able to reduce false positives caused by the separation of object usages over multiple methods?**
  In our experiments we were able to show that using an interprocedural GROUM model is able to significantly reduce false positives that are caused by the separation of object usages among several methods. We reached reduction rates of detected violations by up to 30.67% and, thus, by up to 42.86% for those false positives. However, we believe that tackling the stated limitations of the approach can yield even better results.

- **How does interprocedural GROUM generation impact the detection of true positives and other false positives?**
  Large amounts of duplicated violations – true and false positives alike – were observed during the experiments. Nevertheless, significant reduction of violations could be reached despite the introduced redundancies. Thus, the eliminated amount of false positives seems to be superior to the introduced duplicates.

- **What are possible strategies for interprocedural GROUM generation? How can the generation of such interprocedural GROUMs avoid including API internals?**
  While we provide several heuristics to constrain the inlining of proceeding to deep in the inline hierarchy, i.e., inline internals of API calls so that those are no longer modeled by the GROUMs, we did not observe any of those cases. We assume that our test set does not contain implementations of APIs of interest. Furthermore, we observed that while constraining the inlining can yield slight improvement in some cases; in most cases it does rather hurt the potential of the inliner. Thus, we suggest using the *AlwaysInline* heuristic as long as there are no inlinings of API internals observable. Still, if conducting experiments on specific APIs, it is easy to provide a heuristic that explicitly prevents calls to that API to get inlined.

Future Work

- **A more sophisticated handling of virtual callsites**
  In this thesis, a naive approach for handling virtual calls with multiple target objects is presented. However, the approach does resolve such calls to just one of the possible targets. Inlining all possible receivers of such a call leads to several versions of a caller-GROUM and, thus, to large amounts of duplicated violations. A more sophisticated approach is needed, which allows for inlining every possible method of such a call, while not leading to duplicated versions of the caller-GROUM, or at least version without duplicate violations.

- **Extending the GROUM model to capture field usages**
  The current model does not capture class *field usages*. Extending the model for such would allow resolving relations between calls on such class fields and, thus, the elimination of false positives that occur due to interprocedural object usages where the object is shared by a class field.

- **Eliminating duplicate violations introduced by inlining**
  We consider the most promising extension to be an approach for detecting and eliminating duplicated violations. These occur if violations that are not caused due to interprocedural object usage (alone) are introduced in several caller-GROUMs. Means have to be found that allow the identification of introduced duplicates and a collection of those, so that these violations are reported as a composite while still keeping track of all possible occurrences.

- **Improved modeling of exceptional flow**
  The current model has an over-approximative way of handling exceptional flow. Since there is no way to identify the type of a thrown exception, exceptional control flow cannot be precisely handled. Currently, when inlining, every throw statement is connected to every exception handler of the outer GROUM. Means to correctly map paths where an exception is thrown with the corresponding exception handler have to be developed.

List of Figures

1.1. Original GROUM for AES encryption example 1

3.1. Original GROUM for control structure example 6
3.2. Original GROUM for AES encryption example with exception handling 8
3.3. Original GROUM for the AES Cipher factory method 8
3.4. Pattern for javax.crypto.Cipher usage mined from GROUMs using the JCE 8
3.5. Interprocedural GROUM for AES encryption example with exception handling 9

4.1. Master for parameter passing example in old model 13
4.2. Inlinee for parameter passing example in old model 13
4.3. GROUM for the exception handling example with advanced edges 14
4.4. GROUM for the data flow example with advanced edges 16
4.5. GROUM for a method with no control flow 19
4.6. GROUM for a method that loops-through a parameter 19

5.1. Class diagram for the *IDependencyManager* interface 23
5.2. Overwiev of the inlining infrastructure 26
5.3. Class diagram of the *AbstractInlineBuilder* 27
5.4. Class diagram of the *GroumInliner* class 28
5.5. Class diagram of the *ILookupProvider* interface 28
5.6. Class diagram of the *IInlineHeuristic* interface 30

6.1. Example inline tree for three levels of inlining 37
6.2. First example inline tree for two levels of inlining 37
6.3. Second example inline tree for two levels of inlining 37

References

[AB05] Cyrille Artho and Armin Biere. Subroutine inlining and bytecode abstraction to simplify static and dynamic analysis. *Electronic Notes in Theoretical Computer Science*, 141(1):109 – 128, 2005. ISSN 1571-0661. URL http://www.sciencedirect.com/science/article/pii/S1571066105051467.

[AFSS00] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. A comparative study of static and profile-based heuristics for inlining. *SIGPLAN Not.*, 35(7):52–64, January 2000. ISSN 0362-1340. URL http://doi.acm.org/10.1145/351403.351416.

[ANA+15] Steven Arzt, Sarah Nadi, Karim Ali, Eric Bodden, Sebastian Erdweg, and Mira Mezini. Towards secure integration of cryptographic software. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 1–13. ACM, New York, NY, USA, 2015. ISBN 978-1-4503-3688-8. URL http://doi.acm.org/10.1145/2814228.2814229.

[ANN+16] Sven Amani, Sarah Nadi, Hoan A. Nguyen, Tien N. Nguyen, and Mira Mezini. Mubench: A benchmark for api-misuse detectors. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 464–467. ACM, New York, NY, USA, 2016. ISBN 978-1-4503-4186-8. URL http://doi.acm.org/10.1145/2901739.2903506.

[BCF+99] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The jalapeÑo dynamic optimizing compiler for java. In *Proceedings of the ACM 1999 Conference on Java Grande*, JAVA '99, pages 129–141. ACM, New York, NY, USA, 1999. ISBN 1-58113-161-5. URL http://doi.acm.org/10.1145/304065.304113.

[Bod14] Eric Bodden. Ts4j: A fluent interface for defining and computing typestate analyses. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14, pages 1–6. ACM, New York, NY, USA, 2014. ISBN 978-1-4503-2919-4. URL http://doi.acm.org/10.1145/2614628.2614629.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. *SIGPLAN Not.*, 31(10):324–341, October 1996. ISSN 0362-1340. URL http://doi.acm.org/10.1145/236338.236371.

[CL06] Dhruva R. Chakrabarti and Shin-Ming Liu. Inline analysis: Beyond selection heuristics. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 221–232. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2499-0. URL http://dx.doi.org/10.1109/CGO.2006.17.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101. Springer-Verlag, London, UK, UK, 1995. ISBN 3-540-60160-0. URL http://dl.acm.org/citation.cfm?id=646153.679523.

[EBFK13] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 73–84. ACM,

New York, NY, USA, 2013. ISBN 978-1-4503-2477-9. URL http://doi.acm.org/10.1145/2508859.2516693.

[EH07] William Eberle and Lawrence Holder. Discovering structural anomalies in graph-based data. *2007 7th IEEE International Conference on Data Mining Workshops*, pages 393–398, 2007.

[EKKB10] Frank Eichinger, Klaus Krogmann, Roland Klug, and Klemens Böhm. *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2010, Barcelona, Spain, September 20-24, 2010, Proceedings, Part I*, chapter Software-Defect Localisation by Mining Dataflow-Enabled Call Graphs, pages 425–441. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-15880-3. URL http://dx.doi.org/10.1007/978-3-642-15880-3_33.

[Fow05] Martin Fowler. Fluentinterfaces, December 2005. URL http://martinfowler.com/bliki/FluentInterface.html.

[GIJ+12] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: Validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 38–49. ACM, New York, NY, USA, 2012. ISBN 978-1-4503-1651-4. URL http://doi.acm.org/10.1145/2382196.2382204.

[Han05] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. ISBN 1558609016.

[LCWZ14] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail?: A case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys '14, pages 7:1–7:7. ACM, New York, NY, USA, 2014. ISBN 978-1-4503-3024-4. URL http://doi.acm.org/10.1145/2637166.2637237.

[Lin07] Christian Lindig. Mining patterns and violations using concept analysis. Technical report, Universität des Saarlandes, Saarbrücken, Germany, June 2007.

[LZ05] Zhenmin Li and Yuanyuan Zhou. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. *SIGSOFT Softw. Eng. Notes*, 30(5):306–315, September 2005. ISSN 0163-5948. URL http://doi.acm.org/10.1145/1095430.1081755.

[NKMB16] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: Why do java developers struggle with cryptography apis? In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 935–946. ACM, New York, NY, USA, 2016. ISBN 978-1-4503-3900-1. URL http://doi.acm.org/10.1145/2884781.2884790.

[NNN16] Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. A large-scale study on repetitiveness, containment, and composability of routines in open-source projects. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 362–373. ACM, New York, NY, USA, 2016. ISBN 978-1-4503-4186-8. URL http://doi.acm.org/10.1145/2901739.2901759.

[NNP+09] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-001-2. URL http://doi.acm.org/10.1145/1595696.1595767.

[SCWK13] Douglas Simon, John Cavazos, Christian Wimmer, and Sameer Kulkarni. Automatic construction of inlining heuristics using machine learning. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–12. IEEE Computer Society, Washington, DC, USA, 2013. ISBN 978-1-4673-5524-7. URL http://dx.doi.org/10.1109/CGO.2013.6495004.

[SHR+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. *SIGPLAN Not.*, 35(10):264–280, October 2000. ISSN 0362-1340. URL http://doi.acm.org/10.1145/354222.353189.

[VRCG+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999. URL http://dl.acm.org/citation.cfm?id=781995.782008.

[Wer07] Tobias Werth. Design and implementation of a dag-miner (entwurf und implementation eines dag-miners). Master's thesis, Friedrich-Alexander-Universität Erlangen–Nürnberg, January 2007. URL https://www2.informatik.uni-erlangen.de/EN/teaching/thesis/download/i2D00355.pdf.

[WZL07] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 35–44. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-811-4. URL http://doi.acm.org/10.1145/1287624.1287632.

A Appendix

The following listings contain the data obtained in our experiments. Every table corresponds to a specific minimal overlap threshold (i.e., 30%, 20% and 0%). The number of violations for the intraprocedural model can be found in the corresponding tables heading. General information about the experiment setup can be found in section 6.1. The average number of nodes for the intraprocedural model is 18, while the average number of edges is 31. Abbreviations used in the listings can be found in table A.1. Since we filter out inlinees to not sophisticate the test results, the number of targets in every test run is the number of GROUMs (1012) minus the number of inlinees. An exception for this is when inlining is constraint to a certain inline level that is smaller than the maximum possible level of inlining. In such a case some GROUMs can be masters, thus targets for the anomaly detection, and inlinees at the same time (see figs. 6.1 to 6.3 for an example). A more detailed explanation for this phenomenon can be found in section 6.3.

Table A.1.: Abbreviations

Abbreviation	Description
filter	**true**: interprocedural GROUMs that introduce additional violations are filtered out **false**: no filtering
cg	**true**: the *NaiveCallGraphLookupProvider* is used **false**: the *SimpleLookupProvider* is used
t	Number of targets for the anomaly detection
ees	Number of inlinees
ings	Number of inlinings
v	Violations with the interprocedural model
diff	Violation difference between the inter- and intraprocedural model
impr	Violation reduction by the interprocedural model
time	Time to convert all intraprocedural GROUMs to their interprocedural analog
n	Average number of nodes in the interprocedural model
e	Average number of edges in the interprocedural model
RoGPkg	*ReturnsOrGetsPackageHeuristic*
RoGCls	*ReturnsOrGetsClassHeuristic*
IISOrSubPkg	*IsInSameOrSubPackageHeuristic*
IISCls	*IsInSameClassHeuristic*
Level(x)	*LevelBasedHeuristic* with level x
(AND)	Conjunctional *HeuristicChain*
(OR)	Disjunctional *HeuristicChain*

Table A.2.: Results for a minimal overlap threshold of 30% (violations intra: 208)

filter	cg	heuristic	t	ees	ings	v	diff	impr	time	n	e
false	false	AlwaysInline	780	232	568	223	-15	-7.21%	0.240 s	25	46
false	false	RoGPkg(javax.crypto)	842	170	429	221	-13	-6.25%	0.149 s	23	41
false	false	RoGCls(javax.crypto.Cipher)	956	56	140	215	-7	-3.37%	0.082 s	19	34
false	false	IISOrSubPkg	798	214	488	225	-17	-8.17%	0.079 s	24	45
false	false	IISCls	938	74	172	207	1	0.48%	0.054 s	20	35
false	false	LevelHeuristic(1)	826	232	512	233	-25	-12.02%	0.079 s	24	44

filter	cg	heuristic	t	ees	ings	v	diff	impr	time	n	e
false	false	Level(2)	786	232	574	227	-19	-9.13%	0.098 s	25	46
false	false	Level(3)	782	232	575	225	-17	-8.17%	0.071 s	25	46
false	false	Level(4)	780	232	568	223	-15	-7.21%	0.067 s	25	46
false	false	Level(5)	780	232	568	223	-15	-7.21%	0.066 s	25	46
false	false	(AND): [Level(1), RoGPkg(javax.crypto)]	876	170	387	219	-11	-5.29%	0.052 s	22	39
false	false	(AND): [Level(2), RoGPkg(javax.crypto)]	847	170	433	225	-17	-8.17%	0.057 s	22	41
false	false	(AND): [Level(3), RoGPkg(javax.crypto)]	843	170	431	223	-15	-7.21%	0.070 s	22	41
false	false	(AND): [Level(4), RoGPkg(javax.crypto)]	842	170	429	221	-13	-6.25%	0.056 s	23	41
false	false	(AND): [Level(5), RoGPkg(javax.crypto)]	842	170	429	221	-13	-6.25%	0.056 s	23	41
false	false	(AND): [Level(1), RoGCls(javax.crypto.Cipher)]	969	56	125	216	-8	-3.85%	0.043 s	19	34
false	false	(AND): [Level(2), RoGCls(javax.crypto.Cipher)]	957	56	140	219	-11	-5.29%	0.045 s	19	34
false	false	(AND): [Level(3), RoGCls(javax.crypto.Cipher)]	956	56	140	215	-7	-3.37%	0.045 s	19	34
false	false	(AND): [Level(4), RoGCls(javax.crypto.Cipher)]	956	56	140	215	-7	-3.37%	0.045 s	19	34
false	false	(AND): [Level(5), RoGCls(javax.crypto.Cipher)]	956	56	140	215	-7	-3.37%	0.045 s	19	34
false	false	(AND): [Level(1), IISCls]	947	74	169	213	-5	-2.40%	0.046 s	19	34
false	false	(AND): [Level(2), IISCls]	938	74	172	207	1	0.48%	0.047 s	20	35
false	false	(AND): [Level(3), IISCls]	938	74	172	207	1	0.48%	0.056 s	20	35
false	false	(AND): [Level(4), IISCls]	938	74	172	207	1	0.48%	0.059 s	20	35
false	false	(AND): [Level(5), IISCls]	938	74	172	207	1	0.48%	0.059 s	20	35
false	false	(AND): [Level(1), IISOrSubPkg]	832	214	451	227	-19	-9.13%	0.058 s	23	43
false	false	(AND): [Level(2), IISOrSubPkg]	799	214	489	225	-17	-8.17%	0.062 s	24	44
false	false	(AND): [Level(3), IISOrSubPkg]	798	214	488	225	-17	-8.17%	0.062 s	24	45
false	false	(AND): [Level(4), IISOrSubPkg]	798	214	488	225	-17	-8.17%	0.063 s	24	45
false	false	(AND): [Level(5), IISOrSubPkg]	798	214	488	225	-17	-8.17%	0.062 s	24	45
false	false	(OR): [Level(1), RoGPkg(javax.crypto)]	796	232	563	236	-28	-13.46%	0.064 s	25	46
false	false	(OR): [Level(2), RoGPkg(javax.crypto)]	782	232	574	223	-15	-7.21%	0.070 s	25	47
false	false	(OR): [Level(3), RoGPkg(javax.crypto)]	781	232	573	223	-15	-7.21%	0.066 s	25	46
false	false	(OR): [Level(4), RoGPkg(javax.crypto)]	780	232	568	223	-15	-7.21%	0.066 s	25	46
false	false	(OR): [Level(5), RoGPkg(javax.crypto)]	780	232	568	223	-15	-7.21%	0.066 s	25	46
false	false	(OR): [Level(1), RoGCls(javax.crypto.Cipher)]	810	232	532	236	-28	-13.46%	0.087 s	24	45
false	false	(OR): [Level(2), RoGCls(javax.crypto.Cipher)]	784	232	573	223	-15	-7.21%	0.064 s	25	46
false	false	(OR): [Level(3), RoGCls(javax.crypto.Cipher)]	781	232	573	223	-15	-7.21%	0.065 s	25	46
false	false	(OR): [Level(4), RoGCls(javax.crypto.Cipher)]	780	232	568	223	-15	-7.21%	0.065 s	25	46
false	false	(OR): [Level(5), RoGCls(javax.crypto.Cipher)]	780	232	568	223	-15	-7.21%	0.065 s	25	46
false	false	(OR): [Level(1), IISCls]	800	232	547	230	-22	-10.58%	0.063 s	25	46
false	false	(OR): [Level(2), IISCls]	782	232	570	223	-15	-7.21%	0.065 s	25	46
false	false	(OR): [Level(3), IISCls]	780	232	568	223	-15	-7.21%	0.065 s	25	46
false	false	(OR): [Level(4), IISCls]	780	232	568	223	-15	-7.21%	0.066 s	25	46
false	false	(OR): [Level(5), IISCls]	780	232	568	223	-15	-7.21%	0.065 s	25	46
false	false	(OR): [Level(1), IISOrSubPkg]	789	232	562	231	-23	-11.06%	0.065 s	25	46
false	false	(OR): [Level(2), IISOrSubPkg]	781	232	569	223	-15	-7.21%	0.065 s	25	46
false	false	(OR): [Level(3), IISOrSubPkg]	780	232	568	223	-15	-7.21%	0.066 s	25	46
false	false	(OR): [Level(4), IISOrSubPkg]	780	232	568	223	-15	-7.21%	0.066 s	25	46
false	false	(OR): [Level(5), IISOrSubPkg]	780	232	568	223	-15	-7.21%	0.065 s	25	46
false	false	(AND): [RoGPkg(javax.crypto), IISOrSubPkg]	850	162	384	219	-11	-5.29%	0.055 s	22	40
false	false	(AND): [RoGPkg(javax.crypto), IISCls]	951	61	152	201	7	3.37%	0.045 s	19	34
false	false	(OR): [RoGCls(javax.crypto.Cipher), IISOrSubPkg]	795	217	505	224	-16	-7.69%	0.064 s	24	45
false	false	(OR): [RoGCls(javax.crypto.Cipher), IISCls]	903	109	282	222	-14	-6.73%	0.052 s	21	38
true	false	AlwaysInline	984	28	53	191	17	8.17%	0.065 s	18	32
true	false	RoGPkg(javax.crypto)	995	17	31	192	16	7.69%	0.058 s	18	32
true	false	RoGCls(javax.crypto.Cipher)	996	16	30	192	16	7.69%	0.047 s	18	32
true	false	IISOrSubPkg	1000	12	23	195	13	6.25%	0.065 s	18	31
true	false	IISCls	1006	6	8	192	16	7.69%	0.051 s	18	31
true	false	Level(1)	1007	5	10	194	14	6.73%	0.061 s	18	31
true	false	Level(2)	990	23	44	192	16	7.69%	0.068 s	18	32
true	false	Level(3)	986	26	51	191	17	8.17%	0.067 s	18	32
true	false	Level(4)	984	28	53	191	17	8.17%	0.068 s	18	32
true	false	Level(5)	984	28	53	191	17	8.17%	0.068 s	18	32
true	false	(AND): [Level(1), RoGPkg(javax.crypto)]	1007	5	10	194	14	6.73%	0.054 s	18	31
true	false	(AND): [Level(2), RoGPkg(javax.crypto)]	998	14	27	193	15	7.21%	0.059 s	18	32
true	false	(AND): [Level(3), RoGPkg(javax.crypto)]	995	17	31	192	16	7.69%	0.057 s	18	32
true	false	(AND): [Level(4), RoGPkg(javax.crypto)]	995	17	31	192	16	7.69%	0.058 s	18	32
true	false	(AND): [Level(5), RoGPkg(javax.crypto)]	995	17	31	192	16	7.69%	0.059 s	18	32
true	false	(AND): [Level(1), RoGCls(javax.crypto.Cipher)]	1008	4	9	194	14	6.73%	0.045 s	18	31
true	false	(AND): [Level(2), RoGCls(javax.crypto.Cipher)]	999	13	26	193	15	7.21%	0.046 s	18	32
true	false	(AND): [Level(3), RoGCls(javax.crypto.Cipher)]	996	16	30	192	16	7.69%	0.047 s	18	32
true	false	(AND): [Level(4), RoGCls(javax.crypto.Cipher)]	996	16	30	192	16	7.69%	0.046 s	18	32

filter	cg	heuristic	t	ees	ings	v	diff	impr	time	n	e
true	false	(AND): [Level(5), RoGCls(javax.crypto.Cipher)]	996	16	30	192	16	7.69%	0.048 s	18	32
true	false	(AND): [Level(1), IISCls]	1008	4	6	194	14	6.73%	0.056 s	18	31
true	false	(AND): [Level(2), IISCls]	1006	6	8	192	16	7.69%	0.049 s	18	31
true	false	(AND): [Level(3), IISCls]	1006	6	8	192	16	7.69%	0.057 s	18	31
true	false	(AND): [Level(4), IISCls]	1006	6	8	192	16	7.69%	0.049 s	18	31
true	false	(AND): [Level(5), IISCls]	1006	6	8	192	16	7.69%	0.048 s	18	31
true	false	(AND): [Level(1), IISOrSubPkg]	1007	5	10	194	14	6.73%	0.060 s	18	31
true	false	(AND): [Level(2), IISOrSubPkg]	1000	12	23	195	13	6.25%	0.066 s	18	31
true	false	(AND): [Level(3), IISOrSubPkg]	1000	12	23	195	13	6.25%	0.065 s	18	31
true	false	(AND): [Level(4), IISOrSubPkg]	1000	12	23	195	13	6.25%	0.065 s	18	31
true	false	(AND): [Level(5), IISOrSubPkg]	1000	12	23	195	13	6.25%	0.066 s	18	31
true	false	(OR): [Level(1), RoGPkg(javax.crypto)]	995	17	31	192	16	7.69%	0.068 s	18	32
true	false	(OR): [Level(2), RoGPkg(javax.crypto)]	987	26	48	191	17	8.17%	0.069 s	18	32
true	false	(OR): [Level(3), RoGPkg(javax.crypto)]	986	26	51	191	17	8.17%	0.070 s	18	32
true	false	(OR): [Level(4), RoGPkg(javax.crypto)]	984	28	53	191	17	8.17%	0.069 s	18	32
true	false	(OR): [Level(5), RoGPkg(javax.crypto)]	984	28	53	191	17	8.17%	0.069 s	18	32
true	false	(OR): [Level(1), RoGCls(javax.crypto.Cipher)]	995	17	31	192	16	7.69%	0.065 s	18	32
true	false	(OR): [Level(2), RoGCls(javax.crypto.Cipher)]	987	26	48	191	17	8.17%	0.069 s	18	32
true	false	(OR): [Level(3), RoGCls(javax.crypto.Cipher)]	986	26	51	191	17	8.17%	0.069 s	18	32
true	false	(OR): [Level(4), RoGCls(javax.crypto.Cipher)]	984	28	53	191	17	8.17%	0.069 s	18	32
true	false	(OR): [Level(5), RoGCls(javax.crypto.Cipher)]	984	28	53	191	17	8.17%	0.069 s	18	32
true	false	(OR): [Level(1), IISCls]	991	21	37	190	18	8.65%	0.067 s	18	32
true	false	(OR): [Level(2), IISCls]	986	26	51	191	17	8.17%	0.069 s	18	32
true	false	(OR): [Level(3), IISCls]	984	28	53	191	17	8.17%	0.068 s	18	32
true	false	(OR): [Level(4), IISCls]	984	28	53	191	17	8.17%	0.067 s	18	32
true	false	(OR): [Level(5), IISCls]	984	28	53	191	17	8.17%	0.068 s	18	32
true	false	(OR): [Level(1), IISOrSubPkg]	991	21	37	190	18	8.65%	0.074 s	18	32
true	false	(OR): [Level(2), IISOrSubPkg]	986	26	51	191	17	8.17%	0.068 s	18	32
true	false	(OR): [Level(3), IISOrSubPkg]	984	28	53	191	17	8.17%	0.074 s	18	32
true	false	(OR): [Level(4), IISOrSubPkg]	984	28	53	191	17	8.17%	0.068 s	18	32
true	false	(OR): [Level(5), IISOrSubPkg]	984	28	53	191	17	8.17%	0.068 s	18	32
true	false	(AND): [RoGPkg(javax.crypto), IISOrSubPkg]	1000	12	23	195	13	6.25%	0.057 s	18	31
true	false	(AND): [RoGPkg(javax.crypto), IISCls]	1006	6	8	192	16	7.69%	0.047 s	18	31
true	false	(OR): [RoGCls(javax.crypto.Cipher), IISOrSubPkg]	995	17	31	192	16	7.69%	0.066 s	18	32
true	false	(OR): [RoGCls(javax.crypto.Cipher), IISCls]	995	17	31	192	16	7.69%	0.055 s	18	32
false	true	AlwaysInline	739	273	947	254	-46	-22.12%	0.264 s	27	51
false	true	RoGPkg(javax.crypto)	814	198	697	242	-34	-16.35%	0.184 s	23	42
false	true	RoGCls(javax.crypto.Cipher)	949	63	227	232	-24	-11.54%	0.155 s	19	35
false	true	IISOrSubPkg	783	229	553	233	-25	-12.02%	0.196 s	25	46
false	true	IISCls	935	77	180	207	1	0.48%	0.158 s	20	35
false	true	Level(1)	827	273	735	252	-44	-21.15%	0.196 s	25	46
false	true	Level(2)	759	273	936	265	-57	-27.40%	0.220 s	27	51
false	true	Level(3)	743	273	954	250	-42	-20.19%	0.222 s	27	51
false	true	Level(4)	739	273	947	254	-46	-22.12%	0.222 s	27	51
false	true	Level(5)	739	273	947	254	-46	-22.12%	0.221 s	27	51
false	true	(AND): [Level(1), RoGPkg(javax.crypto)]	877	198	576	241	-33	-15.87%	0.171 s	22	40
false	true	(AND): [Level(2), RoGPkg(javax.crypto)]	827	198	697	260	-52	-25.00%	0.187 s	23	42
false	true	(AND): [Level(3), RoGPkg(javax.crypto)]	815	198	699	244	-36	-17.31%	0.183 s	23	42
false	true	(AND): [Level(4), RoGPkg(javax.crypto)]	814	198	697	242	-34	-16.35%	0.183 s	23	42
false	true	(AND): [Level(5), RoGPkg(javax.crypto)]	814	198	697	242	-34	-16.35%	0.184 s	23	42
false	true	(AND): [Level(1), RoGCls(javax.crypto.Cipher)]	972	63	158	232	-24	-11.54%	0.150 s	19	34
false	true	(AND): [Level(2), RoGCls(javax.crypto.Cipher)]	954	63	227	250	-42	-20.19%	0.159 s	19	35
false	true	(AND): [Level(3), RoGCls(javax.crypto.Cipher)]	949	63	227	232	-24	-11.54%	0.154 s	19	35
false	true	(AND): [Level(4), RoGCls(javax.crypto.Cipher)]	949	63	227	232	-24	-11.54%	0.154 s	19	35
false	true	(AND): [Level(5), RoGCls(javax.crypto.Cipher)]	949	63	227	232	-24	-11.54%	0.155 s	19	35
false	true	(AND): [Level(1), IISCls]	944	77	177	213	-5	-2.40%	0.150 s	19	34
false	true	(AND): [Level(2), IISCls]	935	77	180	207	1	0.48%	0.148 s	20	35
false	true	(AND): [Level(3), IISCls]	935	77	180	207	1	0.48%	0.161 s	20	35
false	true	(AND): [Level(4), IISCls]	935	77	180	207	1	0.48%	0.169 s	20	35
false	true	(AND): [Level(5), IISCls]	935	77	180	207	1	0.48%	0.164 s	20	35
false	true	(AND): [Level(1), IISOrSubPkg]	824	229	501	236	-28	-13.46%	0.193 s	24	44
false	true	(AND): [Level(2), IISOrSubPkg]	786	229	553	233	-25	-12.02%	0.194 s	25	46
false	true	(AND): [Level(3), IISOrSubPkg]	783	229	553	233	-25	-12.02%	0.210 s	25	46
false	true	(AND): [Level(4), IISOrSubPkg]	783	229	553	233	-25	-12.02%	0.220 s	25	46
false	true	(AND): [Level(5), IISOrSubPkg]	783	229	553	233	-25	-12.02%	0.219 s	25	46
false	true	(OR): [Level(1), RoGPkg(javax.crypto)]	759	273	937	264	-56	-26.92%	0.242 s	27	50
false	true	(OR): [Level(2), RoGPkg(javax.crypto)]	742	273	955	254	-46	-22.12%	0.221 s	27	51

filter	cg	heuristic	t	ees	ings	v	diff	impr	time	n	e
false	true	(OR): [Level(3), RoGPkg(javax.crypto)]	740	273	952	254	-46	-22.12%	0.219 s	27	51
false	true	(OR): [Level(4), RoGPkg(javax.crypto)]	739	273	947	254	-46	-22.12%	0.226 s	27	51
false	true	(OR): [Level(5), RoGPkg(javax.crypto)]	739	273	947	254	-46	-22.12%	0.247 s	27	51
false	true	(OR): [Level(1), RoGCls(javax.crypto.Cipher)]	800	273	814	262	-54	-25.96%	0.224 s	26	47
false	true	(OR): [Level(2), RoGCls(javax.crypto.Cipher)]	749	273	952	260	-52	-25.00%	0.223 s	27	51
false	true	(OR): [Level(3), RoGCls(javax.crypto.Cipher)]	742	273	964	260	-52	-25.00%	0.227 s	27	51
false	true	(OR): [Level(4), RoGCls(javax.crypto.Cipher)]	739	273	947	254	-46	-22.12%	0.227 s	27	51
false	true	(OR): [Level(5), RoGCls(javax.crypto.Cipher)]	739	273	947	254	-46	-22.12%	0.243 s	27	51
false	true	(OR): [Level(1), IISCls]	797	273	778	249	-41	-19.71%	0.221 s	26	49
false	true	(OR): [Level(2), IISCls]	755	273	932	261	-53	-25.48%	0.237 s	27	51
false	true	(OR): [Level(3), IISCls]	741	273	947	248	-40	-19.23%	0.263 s	27	51
false	true	(OR): [Level(4), IISCls]	739	273	947	254	-46	-22.12%	0.220 s	27	51
false	true	(OR): [Level(5), IISCls]	739	273	947	254	-46	-22.12%	0.233 s	27	51
false	true	(OR): [Level(1), IISOrSubPkg]	782	273	818	251	-43	-20.67%	0.233 s	27	50
false	true	(OR): [Level(2), IISOrSubPkg]	752	273	934	261	-53	-25.48%	0.237 s	27	51
false	true	(OR): [Level(3), IISOrSubPkg]	741	273	947	248	-40	-19.23%	0.224 s	27	51
false	true	(OR): [Level(4), IISOrSubPkg]	739	273	947	254	-46	-22.12%	0.221 s	27	51
false	true	(OR): [Level(5), IISOrSubPkg]	739	273	947	254	-46	-22.12%	0.235 s	27	51
false	true	(AND): [RoGPkg(javax.crypto), IISOrSubPkg]	843	169	414	228	-20	-9.62%	0.185 s	22	41
false	true	(AND): [RoGPkg(javax.crypto), IISCls]	949	63	159	201	7	3.37%	0.157 s	19	34
false	true	(OR): [RoGCls(javax.crypto.Cipher), IISOrSubPkg]	773	239	660	246	-38	-18.27%	0.201 s	25	47
false	true	(OR): [RoGCls(javax.crypto.Cipher), IISCls]	893	119	379	239	-31	-14.90%	0.167 s	21	38
true	true	AlwaysInline	945	67	175	151	57	27.40%	0.240 s	19	35
true	true	RoGPkg(javax.crypto)	966	46	137	151	57	27.40%	0.207 s	19	34
true	true	RoGCls(javax.crypto.Cipher)	976	36	109	151	57	27.40%	0.171 s	19	33
true	true	IISOrSubPkg	997	15	29	196	12	5.77%	0.205 s	18	32
true	true	IISCls	1006	6	8	192	16	7.69%	0.149 s	18	31
true	true	Level(1)	1001	11	21	190	18	8.65%	0.195 s	18	31
true	true	Level(2)	970	43	118	162	46	22.12%	0.213 s	19	33
true	true	Level(3)	947	65	173	151	57	27.40%	0.219 s	19	35
true	true	Level(4)	945	67	175	151	57	27.40%	0.222 s	19	35
true	true	Level(5)	945	67	175	151	57	27.40%	0.220 s	19	35
true	true	(AND): [Level(1), RoGPkg(javax.crypto)]	1002	10	20	190	18	8.65%	0.171 s	18	31
true	true	(AND): [Level(2), RoGPkg(javax.crypto)]	982	30	92	162	46	22.12%	0.182 s	18	33
true	true	(AND): [Level(3), RoGPkg(javax.crypto)]	966	46	137	151	57	27.40%	0.184 s	19	34
true	true	(AND): [Level(4), RoGPkg(javax.crypto)]	966	46	137	151	57	27.40%	0.185 s	19	34
true	true	(AND): [Level(5), RoGPkg(javax.crypto)]	966	46	137	151	57	27.40%	0.198 s	19	34
true	true	(AND): [Level(1), RoGCls(javax.crypto.Cipher)]	1006	6	13	190	18	8.65%	0.151 s	18	31
true	true	(AND): [Level(2), RoGCls(javax.crypto.Cipher)]	989	23	71	162	46	22.12%	0.157 s	18	32
true	true	(AND): [Level(3), RoGCls(javax.crypto.Cipher)]	976	36	109	151	57	27.40%	0.156 s	19	33
true	true	(AND): [Level(4), RoGCls(javax.crypto.Cipher)]	976	36	109	151	57	27.40%	0.156 s	19	33
true	true	(AND): [Level(5), RoGCls(javax.crypto.Cipher)]	976	36	109	151	57	27.40%	0.163 s	19	33
true	true	(AND): [Level(1), IISCls]	1008	4	6	194	14	6.73%	0.150 s	18	31
true	true	(AND): [Level(2), IISCls]	1006	6	8	192	16	7.69%	0.149 s	18	31
true	true	(AND): [Level(3), IISCls]	1006	6	8	192	16	7.69%	0.150 s	18	31
true	true	(AND): [Level(4), IISCls]	1006	6	8	192	16	7.69%	0.153 s	18	31
true	true	(AND): [Level(5), IISCls]	1006	6	8	192	16	7.69%	0.154 s	18	31
true	true	(AND): [Level(1), IISOrSubPkg]	1007	5	10	194	14	6.73%	0.185 s	18	31
true	true	(AND): [Level(2), IISOrSubPkg]	997	15	29	196	12	5.77%	0.206 s	18	32
true	true	(AND): [Level(3), IISOrSubPkg]	997	15	29	196	12	5.77%	0.199 s	18	32
true	true	(AND): [Level(4), IISOrSubPkg]	997	15	29	196	12	5.77%	0.199 s	18	32
true	true	(AND): [Level(5), IISOrSubPkg]	997	15	29	196	12	5.77%	0.204 s	18	32
true	true	(OR): [Level(1), RoGPkg(javax.crypto)]	955	57	153	152	56	26.92%	0.218 s	19	34
true	true	(OR): [Level(2), RoGPkg(javax.crypto)]	948	65	170	151	57	27.40%	0.225 s	19	35
true	true	(OR): [Level(3), RoGPkg(javax.crypto)]	947	65	173	151	57	27.40%	0.225 s	19	35
true	true	(OR): [Level(4), RoGPkg(javax.crypto)]	945	67	175	151	57	27.40%	0.221 s	19	35
true	true	(OR): [Level(5), RoGPkg(javax.crypto)]	945	67	175	151	57	27.40%	0.220 s	19	35
true	true	(OR): [Level(1), RoGCls(javax.crypto.Cipher)]	961	51	135	151	57	27.40%	0.200 s	19	34
true	true	(OR): [Level(2), RoGCls(javax.crypto.Cipher)]	948	65	166	151	57	27.40%	0.227 s	19	35
true	true	(OR): [Level(3), RoGCls(javax.crypto.Cipher)]	947	65	173	151	57	27.40%	0.220 s	19	35
true	true	(OR): [Level(4), RoGCls(javax.crypto.Cipher)]	945	67	175	151	57	27.40%	0.220 s	19	35
true	true	(OR): [Level(5), RoGCls(javax.crypto.Cipher)]	945	67	175	151	57	27.40%	0.222 s	19	35
true	true	(OR): [Level(1), IISCls]	985	27	52	186	22	10.58%	0.205 s	18	32
true	true	(OR): [Level(2), IISCls]	966	46	125	161	47	22.60%	0.230 s	19	34
true	true	(OR): [Level(3), IISCls]	945	67	175	151	57	27.40%	0.223 s	19	35
true	true	(OR): [Level(4), IISCls]	945	67	175	151	57	27.40%	0.225 s	19	35
true	true	(OR): [Level(5), IISCls]	945	67	175	151	57	27.40%	0.225 s	19	35

filter	cg	heuristic	t	ees	ings	v	diff	impr	time	n	e
true	true	(OR): [Level(1), IISOrSubPkg]	982	30	58	187	21	10.10%	0.217 s	18	32
true	true	(OR): [Level(2), IISOrSubPkg]	966	46	125	161	47	22.60%	0.234 s	19	34
true	true	(OR): [Level(3), IISOrSubPkg]	945	67	175	151	57	27.40%	0.224 s	19	35
true	true	(OR): [Level(4), IISOrSubPkg]	945	67	175	151	57	27.40%	0.224 s	19	35
true	true	(OR): [Level(5), IISOrSubPkg]	945	67	175	151	57	27.40%	0.224 s	19	35
true	true	(AND): [RoGPkg(javax.crypto), IISOrSubPkg]	1000	12	23	195	13	6.25%	0.187 s	18	31
true	true	(AND): [RoGPkg(javax.crypto), IISCls]	1006	6	8	192	16	7.69%	0.148 s	18	31
true	true	(OR): [RoGCls(javax.crypto.Cipher), IISOrSubPkg]	962	50	141	152	56	26.92%	0.203 s	19	34
true	true	(OR): [RoGCls(javax.crypto.Cipher), IISCls]	970	42	122	151	57	27.40%	0.171 s	19	33

Table A.3.: Results for a minimal overlap threshold of 20% (violations intra: 613)

filter	cg	heuristic	t	ees	ings	v	diff	impr	time	n	e
false	false	AlwaysInline	780	232	568	555	58	9.46%	0.209 s	25	46
false	false	RoGPkg(javax.crypto)	842	170	429	566	47	7.67%	0.119 s	23	41
false	false	RoGCls(javax.crypto.Cipher)	956	56	140	566	47	7.67%	0.082 s	19	34
false	false	IISOrSubPkg	798	214	488	569	44	7.18%	0.101 s	24	45
false	false	IISCls	938	74	172	578	35	5.71%	0.066 s	20	35
false	false	Level(1)	826	232	512	607	6	0.98%	0.086 s	24	44
false	false	Level(2)	786	232	574	563	50	8.16%	0.078 s	25	46
false	false	Level(3)	782	232	575	559	54	8.81%	0.076 s	25	46
false	false	Level(4)	780	232	568	555	58	9.46%	0.068 s	25	46
false	false	Level(5)	780	232	568	555	58	9.46%	0.083 s	25	46
false	false	(AND): [Level(1),RoGPkg(javax.crypto)]	876	170	387	599	14	2.28%	0.053 s	22	39
false	false	(AND): [Level(2),RoGPkg(javax.crypto)]	847	170	433	573	40	6.53%	0.080 s	22	41
false	false	(AND): [Level(3),RoGPkg(javax.crypto)]	843	170	431	569	44	7.18%	0.057 s	22	41
false	false	(AND): [Level(4),RoGPkg(javax.crypto)]	842	170	429	566	47	7.67%	0.058 s	23	41
false	false	(AND): [Level(5),RoGPkg(javax.crypto)]	842	170	429	566	47	7.67%	0.058 s	23	41
false	false	(AND): [Level(1),RoGCls(javax.crypto.Cipher)]	969	56	125	583	30	4.89%	0.044 s	19	34
false	false	(AND): [Level(2),RoGCls(javax.crypto.Cipher)]	957	56	140	572	41	6.69%	0.057 s	19	34
false	false	(AND): [Level(3),RoGCls(javax.crypto.Cipher)]	956	56	140	566	47	7.67%	0.058 s	19	34
false	false	(AND): [Level(4),RoGCls(javax.crypto.Cipher)]	956	56	140	566	47	7.67%	0.048 s	19	34
false	false	(AND): [Level(5),RoGCls(javax.crypto.Cipher)]	956	56	140	566	47	7.67%	0.046 s	19	34
false	false	(AND): [Level(1),IISCls]	947	74	169	591	22	3.59%	0.047 s	19	34
false	false	(AND): [Level(2),IISCls]	938	74	172	578	35	5.71%	0.047 s	20	35
false	false	(AND): [Level(3),IISCls]	938	74	172	578	35	5.71%	0.047 s	20	35
false	false	(AND): [Level(4),IISCls]	938	74	172	578	35	5.71%	0.048 s	20	35
false	false	(AND): [Level(5),IISCls]	938	74	172	578	35	5.71%	0.047 s	20	35
false	false	(AND): [Level(1),IISOrSubPkg]	832	214	451	599	14	2.28%	0.058 s	23	43
false	false	(AND): [Level(2),IISOrSubPkg]	799	214	489	569	44	7.18%	0.063 s	24	44
false	false	(AND): [Level(3),IISOrSubPkg]	798	214	488	569	44	7.18%	0.063 s	24	45
false	false	(AND): [Level(4),IISOrSubPkg]	798	214	488	569	44	7.18%	0.078 s	24	45
false	false	(AND): [Level(5),IISOrSubPkg]	798	214	488	569	44	7.18%	0.063 s	24	45
false	false	(OR): [Level(1),RoGPkg(javax.crypto)]	796	232	563	577	36	5.87%	0.067 s	25	46
false	false	(OR): [Level(2),RoGPkg(javax.crypto)]	782	232	574	556	57	9.30%	0.065 s	25	47
false	false	(OR): [Level(3),RoGPkg(javax.crypto)]	781	232	573	556	57	9.30%	0.065 s	25	46
false	false	(OR): [Level(4),RoGPkg(javax.crypto)]	780	232	568	555	58	9.46%	0.065 s	25	46
false	false	(OR): [Level(5),RoGPkg(javax.crypto)]	780	232	568	555	58	9.46%	0.065 s	25	46
false	false	(OR): [Level(1),RoGCls(javax.crypto.Cipher)]	810	232	532	583	30	4.89%	0.062 s	24	45
false	false	(OR): [Level(2),RoGCls(javax.crypto.Cipher)]	784	232	573	556	57	9.30%	0.068 s	25	46
false	false	(OR): [Level(3),RoGCls(javax.crypto.Cipher)]	781	232	573	556	57	9.30%	0.065 s	25	46
false	false	(OR): [Level(4),RoGCls(javax.crypto.Cipher)]	780	232	568	555	58	9.46%	0.065 s	25	46
false	false	(OR): [Level(5),RoGCls(javax.crypto.Cipher)]	780	232	568	555	58	9.46%	0.065 s	25	46
false	false	(OR): [Level(1),IISCls]	800	232	547	583	30	4.89%	0.068 s	25	46
false	false	(OR): [Level(2),IISCls]	782	232	570	556	57	9.30%	0.065 s	25	46
false	false	(OR): [Level(3),IISCls]	780	232	568	555	58	9.46%	0.066 s	25	46
false	false	(OR): [Level(4),IISCls]	780	232	568	555	58	9.46%	0.065 s	25	46
false	false	(OR): [Level(5),IISCls]	780	232	568	555	58	9.46%	0.065 s	25	46
false	false	(OR): [Level(1),IISOrSubPkg]	789	232	562	565	48	7.83%	0.065 s	25	46
false	false	(OR): [Level(2),IISOrSubPkg]	781	232	569	556	57	9.30%	0.066 s	25	46
false	false	(OR): [Level(3),IISOrSubPkg]	780	232	568	555	58	9.46%	0.066 s	25	46
false	false	(OR): [Level(4),IISOrSubPkg]	780	232	568	555	58	9.46%	0.065 s	25	46
false	false	(OR): [Level(5),IISOrSubPkg]	780	232	568	555	58	9.46%	0.065 s	25	46
false	false	(AND): [RoGPkg(javax.crypto),IISOrSubPkg]	850	162	384	574	39	6.36%	0.054 s	22	40
false	false	(AND): [RoGPkg(javax.crypto),IISCls]	951	61	152	573	40	6.53%	0.048 s	19	34
false	false	(OR): [RoGCls(javax.crypto.Cipher),IISOrSubPkg]	795	217	505	558	55	8.97%	0.063 s	24	45

filter	cg	heuristic	t	ees	ings	v	diff	impr	time	n	e
false	false	(OR): [RoGCls(javax.crypto.Cipher),IISCls]	903	109	282	572	41	6.69%	0.052 s	21	38
true	false	AlwaysInline	953	59	125	543	70	11.42%	0.065 s	19	34
true	false	RoGPkg(javax.crypto)	972	40	91	558	55	8.97%	0.063 s	19	33
true	false	RoGCls(javax.crypto.Cipher)	992	20	50	562	51	8.32%	0.047 s	18	32
true	false	IISOrSubPkg	968	44	84	552	61	9.95%	0.073 s	19	33
true	false	IISCls	995	17	30	564	49	7.99%	0.049 s	18	32
true	false	Level(1)	980	33	54	553	60	9.79%	0.065 s	18	32
true	false	Level(2)	958	56	124	550	63	10.28%	0.067 s	19	34
true	false	Level(3)	955	59	132	547	66	10.77%	0.067 s	19	34
true	false	Level(4)	953	59	125	543	70	11.42%	0.068 s	19	34
true	false	Level(5)	953	59	125	543	70	11.42%	0.068 s	19	34
true	false	(AND): [Level(1),RoGPkg(javax.crypto)]	990	23	42	563	50	8.16%	0.053 s	18	32
true	false	(AND): [Level(2),RoGPkg(javax.crypto)]	974	39	90	564	49	7.99%	0.058 s	19	33
true	false	(AND): [Level(3),RoGPkg(javax.crypto)]	973	40	93	561	52	8.48%	0.063 s	19	33
true	false	(AND): [Level(4),RoGPkg(javax.crypto)]	972	40	91	558	55	8.97%	0.059 s	19	33
true	false	(AND): [Level(5),RoGPkg(javax.crypto)]	972	40	91	558	55	8.97%	0.065 s	19	33
true	false	(AND): [Level(1),RoGCls(javax.crypto.Cipher)]	1001	11	23	567	46	7.50%	0.046 s	18	31
true	false	(AND): [Level(2),RoGCls(javax.crypto.Cipher)]	993	20	50	568	45	7.34%	0.055 s	18	32
true	false	(AND): [Level(3),RoGCls(javax.crypto.Cipher)]	992	20	50	562	51	8.32%	0.048 s	18	32
true	false	(AND): [Level(4),RoGCls(javax.crypto.Cipher)]	992	20	50	562	51	8.32%	0.054 s	18	32
true	false	(AND): [Level(5),RoGCls(javax.crypto.Cipher)]	992	20	50	562	51	8.32%	0.048 s	18	32
true	false	(AND): [Level(1),IISCls]	998	14	23	569	44	7.18%	0.047 s	18	32
true	false	(AND): [Level(2),IISCls]	995	17	30	564	49	7.99%	0.049 s	18	32
true	false	(AND): [Level(3),IISCls]	995	17	30	564	49	7.99%	0.048 s	18	32
true	false	(AND): [Level(4),IISCls]	995	17	30	564	49	7.99%	0.049 s	18	32
true	false	(AND): [Level(5),IISCls]	995	17	30	564	49	7.99%	0.049 s	18	32
true	false	(AND): [Level(1),IISOrSubPkg]	983	30	50	555	58	9.46%	0.068 s	18	32
true	false	(AND): [Level(2),IISOrSubPkg]	968	44	84	552	61	9.95%	0.065 s	19	33
true	false	(AND): [Level(3),IISOrSubPkg]	968	44	84	552	61	9.95%	0.065 s	19	33
true	false	(AND): [Level(4),IISOrSubPkg]	968	44	84	552	61	9.95%	0.065 s	19	33
true	false	(AND): [Level(5),IISOrSubPkg]	968	44	84	552	61	9.95%	0.066 s	19	33
true	false	(OR): [Level(1),RoGPkg(javax.crypto)]	964	50	107	556	57	9.30%	0.066 s	19	34
true	false	(OR): [Level(2),RoGPkg(javax.crypto)]	956	57	125	544	69	11.26%	0.068 s	19	34
true	false	(OR): [Level(3),RoGPkg(javax.crypto)]	954	59	130	544	69	11.26%	0.067 s	19	34
true	false	(OR): [Level(4),RoGPkg(javax.crypto)]	953	59	125	543	70	11.42%	0.068 s	19	34
true	false	(OR): [Level(5),RoGPkg(javax.crypto)]	953	59	125	543	70	11.42%	0.067 s	19	34
true	false	(OR): [Level(1),RoGCls(javax.crypto.Cipher)]	968	46	91	554	59	9.62%	0.064 s	19	33
true	false	(OR): [Level(2),RoGCls(javax.crypto.Cipher)]	956	57	125	544	69	11.26%	0.067 s	19	34
true	false	(OR): [Level(3),RoGCls(javax.crypto.Cipher)]	954	59	130	544	69	11.26%	0.073 s	19	34
true	false	(OR): [Level(4),RoGCls(javax.crypto.Cipher)]	953	59	125	543	70	11.42%	0.095 s	19	34
true	false	(OR): [Level(5),RoGCls(javax.crypto.Cipher)]	953	59	125	543	70	11.42%	0.076 s	19	34
true	false	(OR): [Level(1),IISCls]	969	44	88	546	67	10.93%	0.066 s	19	33
true	false	(OR): [Level(2),IISCls]	955	57	123	544	69	11.26%	0.068 s	19	34
true	false	(OR): [Level(3),IISCls]	953	59	125	543	70	11.42%	0.068 s	19	34
true	false	(OR): [Level(4),IISCls]	953	59	125	543	70	11.42%	0.069 s	19	34
true	false	(OR): [Level(5),IISCls]	953	59	125	543	70	11.42%	0.068 s	19	34
true	false	(OR): [Level(1),IISOrSubPkg]	961	51	107	543	70	11.42%	0.067 s	19	34
true	false	(OR): [Level(2),IISOrSubPkg]	955	57	123	544	69	11.26%	0.068 s	19	34
true	false	(OR): [Level(3),IISOrSubPkg]	953	59	125	543	70	11.42%	0.070 s	19	34
true	false	(OR): [Level(4),IISOrSubPkg]	953	59	125	543	70	11.42%	0.069 s	19	34
true	false	(OR): [Level(5),IISOrSubPkg]	953	59	125	543	70	11.42%	0.069 s	19	34
true	false	(AND): [RoGPkg(javax.crypto),IISOrSubPkg]	976	36	74	563	50	8.16%	0.057 s	19	33
true	false	(AND): [RoGPkg(javax.crypto),IISCls]	996	16	29	565	48	7.83%	0.049 s	18	32
true	false	(OR): [RoGCls(javax.crypto.Cipher),IISOrSubPkg]	964	48	98	544	69	11.26%	0.067 s	19	34
true	false	(OR): [RoGCls(javax.crypto.Cipher),IISCls]	982	30	71	561	52	8.48%	0.056 s	19	33
false	true	AlwaysInline	739	273	947	446	167	27.24%	0.268 s	27	51
false	true	RoGPkg(javax.crypto)	814	198	697	444	169	27.57%	0.195 s	23	42
false	true	RoGCls(javax.crypto.Cipher)	949	63	227	440	173	28.22%	0.163 s	19	35
false	true	IISOrSubPkg	783	229	553	579	34	5.55%	0.204 s	25	46
false	true	IISCls	935	77	180	576	37	6.04%	0.154 s	20	35
false	true	Level(1)	827	273	735	637	-24	-3.92%	0.216 s	25	46
false	true	Level(2)	759	273	936	518	95	15.50%	0.221 s	27	51
false	true	Level(3)	743	273	954	457	156	25.45%	0.227 s	27	51
false	true	Level(4)	739	273	947	446	167	27.24%	0.228 s	27	51
false	true	Level(5)	739	273	947	446	167	27.24%	0.225 s	27	51
false	true	(AND): [Level(1),RoGPkg(javax.crypto)]	877	198	576	613	0	0.00%	0.175 s	22	40
false	true	(AND): [Level(2),RoGPkg(javax.crypto)]	827	198	697	496	117	19.09%	0.186 s	23	42

filter	cg	heuristic	t	ees	ings	v	diff	impr	time	n	e
false	true	(AND): [Level(3),RoGPkg(javax.crypto)]	815	198	699	447	166	27.08%	0.189 s	23	42
false	true	(AND): [Level(4),RoGPkg(javax.crypto)]	814	198	697	444	169	27.57%	0.196 s	23	42
false	true	(AND): [Level(5),RoGPkg(javax.crypto)]	814	198	697	444	169	27.57%	0.189 s	23	42
false	true	(AND): [Level(1),RoGCls(javax.crypto.Cipher)]	972	63	158	590	23	3.75%	0.153 s	19	34
false	true	(AND): [Level(2),RoGCls(javax.crypto.Cipher)]	954	63	227	486	127	20.72%	0.161 s	19	35
false	true	(AND): [Level(3),RoGCls(javax.crypto.Cipher)]	949	63	227	440	173	28.22%	0.158 s	19	35
false	true	(AND): [Level(4),RoGCls(javax.crypto.Cipher)]	949	63	227	440	173	28.22%	0.158 s	19	35
false	true	(AND): [Level(5),RoGCls(javax.crypto.Cipher)]	949	63	227	440	173	28.22%	0.157 s	19	35
false	true	(AND): [Level(1),IISCls]	944	77	177	589	24	3.92%	0.150 s	19	34
false	true	(AND): [Level(2),IISCls]	935	77	180	576	37	6.04%	0.152 s	20	35
false	true	(AND): [Level(3),IISCls]	935	77	180	576	37	6.04%	0.152 s	20	35
false	true	(AND): [Level(4),IISCls]	935	77	180	576	37	6.04%	0.155 s	20	35
false	true	(AND): [Level(5),IISCls]	935	77	180	576	37	6.04%	0.157 s	20	35
false	true	(AND): [Level(1),IISOrSubPkg]	824	229	501	613	0	0.00%	0.188 s	24	44
false	true	(AND): [Level(2),IISOrSubPkg]	786	229	553	583	30	4.89%	0.207 s	25	46
false	true	(AND): [Level(3),IISOrSubPkg]	783	229	553	579	34	5.55%	0.201 s	25	46
false	true	(AND): [Level(4),IISOrSubPkg]	783	229	553	579	34	5.55%	0.202 s	25	46
false	true	(AND): [Level(5),IISOrSubPkg]	783	229	553	579	34	5.55%	0.201 s	25	46
false	true	(OR): [Level(1),RoGPkg(javax.crypto)]	759	273	937	465	148	24.14%	0.231 s	27	50
false	true	(OR): [Level(2),RoGPkg(javax.crypto)]	742	273	955	447	166	27.08%	0.224 s	27	51
false	true	(OR): [Level(3),RoGPkg(javax.crypto)]	740	273	952	447	166	27.08%	0.228 s	27	51
false	true	(OR): [Level(4),RoGPkg(javax.crypto)]	739	273	947	446	167	27.24%	0.227 s	27	51
false	true	(OR): [Level(5),RoGPkg(javax.crypto)]	739	273	947	446	167	27.24%	0.227 s	27	51
false	true	(OR): [Level(1),RoGCls(javax.crypto.Cipher)]	800	273	814	474	139	22.68%	0.211 s	26	47
false	true	(OR): [Level(2),RoGCls(javax.crypto.Cipher)]	749	273	952	456	157	25.61%	0.222 s	27	51
false	true	(OR): [Level(3),RoGCls(javax.crypto.Cipher)]	742	273	964	454	159	25.94%	0.225 s	27	51
false	true	(OR): [Level(4),RoGCls(javax.crypto.Cipher)]	739	273	947	446	167	27.24%	0.230 s	27	51
false	true	(OR): [Level(5),RoGCls(javax.crypto.Cipher)]	739	273	947	446	167	27.24%	0.226 s	27	51
false	true	(OR): [Level(1),IISCls]	797	273	778	613	0	0.00%	0.208 s	26	49
false	true	(OR): [Level(2),IISCls]	755	273	932	511	102	16.64%	0.220 s	27	51
false	true	(OR): [Level(3),IISCls]	741	273	947	453	160	26.10%	0.234 s	27	51
false	true	(OR): [Level(4),IISCls]	739	273	947	446	167	27.24%	0.232 s	27	51
false	true	(OR): [Level(5),IISCls]	739	273	947	446	167	27.24%	0.233 s	27	51
false	true	(OR): [Level(1),IISOrSubPkg]	782	273	818	586	27	4.40%	0.219 s	27	50
false	true	(OR): [Level(2),IISOrSubPkg]	752	273	934	504	109	17.78%	0.224 s	27	51
false	true	(OR): [Level(3),IISOrSubPkg]	741	273	947	453	160	26.10%	0.230 s	27	51
false	true	(OR): [Level(4),IISOrSubPkg]	739	273	947	446	167	27.24%	0.226 s	27	51
false	true	(OR): [Level(5),IISOrSubPkg]	739	273	947	446	167	27.24%	0.237 s	27	51
false	true	(AND): [RoGPkg(javax.crypto),IISOrSubPkg]	843	169	414	580	33	5.38%	0.184 s	22	41
false	true	(AND): [RoGPkg(javax.crypto),IISCls]	949	63	159	573	40	6.53%	0.150 s	19	34
false	true	(OR): [RoGCls(javax.crypto.Cipher),IISOrSubPkg]	773	239	660	436	177	28.87%	0.207 s	25	47
false	true	(OR): [RoGCls(javax.crypto.Cipher),IISCls]	893	119	379	444	169	27.57%	0.171 s	21	38
true	true	AlwaysInline	883	129	409	433	180	29.36%	0.225 s	22	40
true	true	RoGPkg(javax.crypto)	934	78	290	432	181	29.53%	0.195 s	20	36
true	true	RoGCls(javax.crypto.Cipher)	968	44	192	436	177	28.87%	0.158 s	19	34
true	true	IISOrSubPkg	959	53	111	561	52	8.48%	0.203 s	19	34
true	true	IISCls	994	18	31	562	51	8.32%	0.154 s	18	32
true	true	Level(1)	966	48	98	555	58	9.46%	0.200 s	19	33
true	true	Level(2)	911	107	327	474	139	22.68%	0.226 s	21	39
true	true	Level(3)	889	126	400	438	175	28.55%	0.225 s	22	41
true	true	Level(4)	883	129	409	433	180	29.36%	0.226 s	22	40
true	true	Level(5)	883	129	409	433	180	29.36%	0.229 s	22	40
true	true	(AND): [Level(1),RoGPkg(javax.crypto)]	981	33	80	567	46	7.50%	0.177 s	18	32
true	true	(AND): [Level(2),RoGPkg(javax.crypto)]	944	72	252	475	138	22.51%	0.188 s	20	36
true	true	(AND): [Level(3),RoGPkg(javax.crypto)]	935	78	292	435	178	29.04%	0.190 s	20	36
true	true	(AND): [Level(4),RoGPkg(javax.crypto)]	934	78	290	432	181	29.53%	0.189 s	20	36
true	true	(AND): [Level(5),RoGPkg(javax.crypto)]	934	78	290	432	181	29.53%	0.189 s	20	36
true	true	(AND): [Level(1),RoGCls(javax.crypto.Cipher)]	995	18	55	571	42	6.85%	0.153 s	18	32
true	true	(AND): [Level(2),RoGCls(javax.crypto.Cipher)]	975	40	172	476	137	22.35%	0.159 s	19	34
true	true	(AND): [Level(3),RoGCls(javax.crypto.Cipher)]	968	44	192	436	177	28.87%	0.159 s	19	34
true	true	(AND): [Level(4),RoGCls(javax.crypto.Cipher)]	968	44	192	436	177	28.87%	0.159 s	19	34
true	true	(AND): [Level(5),RoGCls(javax.crypto.Cipher)]	968	44	192	436	177	28.87%	0.165 s	19	34
true	true	(AND): [Level(1),IISCls]	997	15	24	567	46	7.50%	0.151 s	18	32
true	true	(AND): [Level(2),IISCls]	994	18	31	562	51	8.32%	0.155 s	18	32
true	true	(AND): [Level(3),IISCls]	994	18	31	562	51	8.32%	0.153 s	18	32
true	true	(AND): [Level(4),IISCls]	994	18	31	562	51	8.32%	0.152 s	18	32
true	true	(AND): [Level(5),IISCls]	994	18	31	562	51	8.32%	0.152 s	18	32

filter	cg	heuristic	t	ees	ings	v	diff	impr	time	n	e
true	true	(AND): [Level(1),IISOrSubPkg]	979	34	59	559	54	8.81%	0.189 s	18	32
true	true	(AND): [Level(2),IISOrSubPkg]	960	53	109	565	48	7.83%	0.202 s	19	34
true	true	(AND): [Level(3),IISOrSubPkg]	959	53	111	561	52	8.48%	0.201 s	19	34
true	true	(AND): [Level(4),IISOrSubPkg]	959	53	111	561	52	8.48%	0.209 s	19	34
true	true	(AND): [Level(5),IISOrSubPkg]	959	53	111	561	52	8.48%	0.205 s	19	34
true	true	(OR): [Level(1),RoGPkg(javax.crypto)]	896	118	377	443	170	27.73%	0.224 s	21	39
true	true	(OR): [Level(2),RoGPkg(javax.crypto)]	886	127	409	434	179	29.20%	0.234 s	22	40
true	true	(OR): [Level(3),RoGPkg(javax.crypto)]	884	129	414	434	179	29.20%	0.227 s	22	40
true	true	(OR): [Level(4),RoGPkg(javax.crypto)]	883	129	409	433	180	29.36%	0.227 s	22	40
true	true	(OR): [Level(5),RoGPkg(javax.crypto)]	883	129	409	433	180	29.36%	0.229 s	22	40
true	true	(OR): [Level(1),RoGCls(javax.crypto.Cipher)]	912	103	294	435	178	29.04%	0.206 s	21	38
true	true	(OR): [Level(2),RoGCls(javax.crypto.Cipher)]	889	126	398	441	172	28.06%	0.227 s	22	41
true	true	(OR): [Level(3),RoGCls(javax.crypto.Cipher)]	886	129	426	441	172	28.06%	0.225 s	22	41
true	true	(OR): [Level(4),RoGCls(javax.crypto.Cipher)]	883	129	409	433	180	29.36%	0.240 s	22	40
true	true	(OR): [Level(5),RoGCls(javax.crypto.Cipher)]	883	129	409	433	180	29.36%	0.228 s	22	40
true	true	(OR): [Level(1),IISCls]	947	67	147	554	59	9.62%	0.209 s	19	34
true	true	(OR): [Level(2),IISCls]	908	108	326	468	145	23.65%	0.224 s	21	39
true	true	(OR): [Level(3),IISCls]	887	126	393	434	179	29.20%	0.227 s	22	41
true	true	(OR): [Level(4),IISCls]	883	129	409	433	180	29.36%	0.227 s	22	40
true	true	(OR): [Level(5),IISCls]	883	129	409	433	180	29.36%	0.233 s	22	40
true	true	(OR): [Level(1),IISOrSubPkg]	940	74	176	556	57	9.30%	0.224 s	20	35
true	true	(OR): [Level(2),IISOrSubPkg]	907	108	328	464	149	24.31%	0.247 s	21	39
true	true	(OR): [Level(3),IISOrSubPkg]	887	126	393	434	179	29.20%	0.237 s	22	41
true	true	(OR): [Level(4),IISOrSubPkg]	883	129	409	433	180	29.36%	0.228 s	22	40
true	true	(OR): [Level(5),IISOrSubPkg]	883	129	409	433	180	29.36%	0.228 s	22	40
true	true	(AND): [RoGPkg(javax.crypto),IISOrSubPkg]	976	36	78	569	44	7.18%	0.181 s	19	33
true	true	(AND): [RoGPkg(javax.crypto),IISCls]	996	16	29	565	48	7.83%	0.152 s	18	32
true	true	(OR): [RoGCls(javax.crypto.Cipher),IISOrSubPkg]	914	98	309	425	188	30.67%	0.210 s	21	39
true	true	(OR): [RoGCls(javax.crypto.Cipher),IISCls]	955	57	218	433	180	29.36%	0.172 s	19	35

Table A.4.: Results for a minimal overlap threshold of 0% (violations intra: 1105)

filter	cg	heuristic	t	ees	ings	v	diff	impr	time	n	e
false	false	AlwaysInline	780	232	568	1069	36	3.26%	0.283 s	25	46
false	false	RoGPkg(javax.crypto)	842	170	429	1076	29	2.62%	0.158 s	23	41
false	false	RoGCls(javax.crypto.Cipher)	956	56	140	1065	40	3.62%	0.094 s	19	34
false	false	IISOrSubPkg	798	214	488	1081	24	2.17%	0.088 s	24	45
false	false	IISCls	938	74	172	1071	34	3.08%	0.063 s	20	35
false	false	Level(1)	826	232	512	1133	-28	-2.53%	0.081 s	24	44
false	false	Level(2)	786	232	574	1082	23	2.08%	0.081 s	25	46
false	false	Level(3)	782	232	575	1076	29	2.62%	0.073 s	25	46
false	false	Level(4)	780	232	568	1069	36	3.26%	0.084 s	25	46
false	false	Level(5)	780	232	568	1069	36	3.26%	0.067 s	25	46
false	false	(AND): [Level(1), RoGPkg(javax.crypto)]	876	170	387	1115	-10	-0.90%	0.055 s	22	39
false	false	(AND): [Level(2), RoGPkg(javax.crypto)]	847	170	433	1088	17	1.54%	0.066 s	22	41
false	false	(AND): [Level(3), RoGPkg(javax.crypto)]	843	170	431	1082	23	2.08%	0.066 s	22	41
false	false	(AND): [Level(4), RoGPkg(javax.crypto)]	842	170	429	1076	29	2.62%	0.061 s	23	41
false	false	(AND): [Level(5), RoGPkg(javax.crypto)]	842	170	429	1076	29	2.62%	0.066 s	23	41
false	false	(AND): [Level(1), RoGCls(javax.crypto.Cipher)]	969	56	125	1082	23	2.08%	0.055 s	19	34
false	false	(AND): [Level(2), RoGCls(javax.crypto.Cipher)]	957	56	140	1076	29	2.62%	0.046 s	19	34
false	false	(AND): [Level(3), RoGCls(javax.crypto.Cipher)]	956	56	140	1065	40	3.62%	0.045 s	19	34
false	false	(AND): [Level(4), RoGCls(javax.crypto.Cipher)]	956	56	140	1065	40	3.62%	0.045 s	19	34
false	false	(AND): [Level(5), RoGCls(javax.crypto.Cipher)]	956	56	140	1065	40	3.62%	0.045 s	19	34
false	false	(AND): [Level(1), IISCls]	947	74	169	1090	15	1.36%	0.046 s	19	34
false	false	(AND): [Level(2), IISCls]	938	74	172	1071	34	3.08%	0.047 s	20	35
false	false	(AND): [Level(3), IISCls]	938	74	172	1071	34	3.08%	0.047 s	20	35
false	false	(AND): [Level(4), IISCls]	938	74	172	1071	34	3.08%	0.047 s	20	35
false	false	(AND): [Level(5), IISCls]	938	74	172	1071	34	3.08%	0.047 s	20	35
false	false	(AND): [Level(1), IISOrSubPkg]	832	214	451	1117	-12	-1.09%	0.072 s	23	43
false	false	(AND): [Level(2), IISOrSubPkg]	799	214	489	1081	24	2.17%	0.062 s	24	44
false	false	(AND): [Level(3), IISOrSubPkg]	798	214	488	1081	24	2.17%	0.062 s	24	45
false	false	(AND): [Level(4), IISOrSubPkg]	798	214	488	1081	24	2.17%	0.062 s	24	45
false	false	(AND): [Level(5), IISOrSubPkg]	798	214	488	1081	24	2.17%	0.062 s	24	45
false	false	(OR): [Level(1), RoGPkg(javax.crypto)]	796	232	563	1097	8	0.72%	0.064 s	25	46
false	false	(OR): [Level(2), RoGPkg(javax.crypto)]	782	232	574	1070	35	3.17%	0.069 s	25	47
false	false	(OR): [Level(3), RoGPkg(javax.crypto)]	781	232	573	1070	35	3.17%	0.067 s	25	46

filter	cg	heuristic	t	ees	ings	v	diff	impr	time	n	e
false	false	(OR): [Level(4), RoGPkg(javax.crypto)]	780	232	568	1069	36	3.26%	0.066 s	25	46
false	false	(OR): [Level(5), RoGPkg(javax.crypto)]	780	232	568	1069	36	3.26%	0.066 s	25	46
false	false	(OR): [Level(1), RoGCls(javax.crypto.Cipher)]	810	232	532	1108	-3	-0.27%	0.061 s	24	45
false	false	(OR): [Level(2), RoGCls(javax.crypto.Cipher)]	784	232	573	1070	35	3.17%	0.065 s	25	46
false	false	(OR): [Level(3), RoGCls(javax.crypto.Cipher)]	781	232	573	1070	35	3.17%	0.065 s	25	46
false	false	(OR): [Level(4), RoGCls(javax.crypto.Cipher)]	780	232	568	1069	36	3.26%	0.065 s	25	46
false	false	(OR): [Level(5), RoGCls(javax.crypto.Cipher)]	780	232	568	1069	36	3.26%	0.064 s	25	46
false	false	(OR): [Level(1), IISCls]	800	232	547	1102	3	0.27%	0.062 s	25	46
false	false	(OR): [Level(2), IISCls]	782	232	570	1070	35	3.17%	0.064 s	25	46
false	false	(OR): [Level(3), IISCls]	780	232	568	1069	36	3.26%	0.065 s	25	46
false	false	(OR): [Level(4), IISCls]	780	232	568	1069	36	3.26%	0.065 s	25	46
false	false	(OR): [Level(5), IISCls]	780	232	568	1069	36	3.26%	0.065 s	25	46
false	false	(OR): [Level(1), IISOrSubPkg]	789	232	562	1083	22	1.99%	0.064 s	25	46
false	false	(OR): [Level(2), IISOrSubPkg]	781	232	569	1070	35	3.17%	0.064 s	25	46
false	false	(OR): [Level(3), IISOrSubPkg]	780	232	568	1069	36	3.26%	0.065 s	25	46
false	false	(OR): [Level(4), IISOrSubPkg]	780	232	568	1069	36	3.26%	0.065 s	25	46
false	false	(OR): [Level(5), IISOrSubPkg]	780	232	568	1069	36	3.26%	0.068 s	25	46
false	false	(AND): [RoGPkg(javax.crypto), IISOrSubPkg]	850	162	384	1085	20	1.81%	0.086 s	22	40
false	false	(AND): [RoGPkg(javax.crypto), IISCls]	951	61	152	1068	37	3.35%	0.046 s	19	34
false	false	(OR): [RoGCls(javax.crypto.Cipher), IISOrSubPkg]	795	217	505	1069	36	3.26%	0.063 s	24	45
false	false	(OR): [RoGCls(javax.crypto.Cipher), IISCls]	903	109	282	1071	34	3.08%	0.057 s	21	38
true	false	AlwaysInline	942	70	144	1042	63	5.70%	0.064 s	19	35
true	false	RoGPkg(javax.crypto)	967	45	98	1058	47	4.25%	0.058 s	19	33
true	false	RoGCls(javax.crypto.Cipher)	992	20	50	1058	47	4.25%	0.047 s	18	32
true	false	IISOrSubPkg	957	55	103	1051	54	4.89%	0.064 s	19	34
true	false	IISCls	992	20	38	1053	52	4.71%	0.048 s	18	32
true	false	Level(1)	970	43	72	1046	59	5.34%	0.061 s	19	33
true	false	Level(2)	947	67	143	1054	51	4.62%	0.066 s	19	34
true	false	Level(3)	944	70	151	1049	56	5.07%	0.067 s	20	35
true	false	Level(4)	942	70	144	1042	63	5.70%	0.067 s	19	35
true	false	Level(5)	942	70	144	1042	63	5.70%	0.068 s	19	35
true	false	(AND): [Level(1), RoGPkg(javax.crypto)]	986	27	48	1058	47	4.25%	0.054 s	18	32
true	false	(AND): [Level(2), RoGPkg(javax.crypto)]	969	44	97	1069	36	3.26%	0.057 s	19	33
true	false	(AND): [Level(3), RoGPkg(javax.crypto)]	968	45	100	1064	41	3.71%	0.058 s	19	33
true	false	(AND): [Level(4), RoGPkg(javax.crypto)]	967	45	98	1058	47	4.25%	0.057 s	19	33
true	false	(AND): [Level(5), RoGPkg(javax.crypto)]	967	45	98	1058	47	4.25%	0.064 s	19	33
true	false	(AND): [Level(1), RoGCls(javax.crypto.Cipher)]	1001	11	23	1058	47	4.25%	0.047 s	18	31
true	false	(AND): [Level(2), RoGCls(javax.crypto.Cipher)]	993	20	50	1069	36	3.26%	0.063 s	18	32
true	false	(AND): [Level(3), RoGCls(javax.crypto.Cipher)]	992	20	50	1058	47	4.25%	0.047 s	18	32
true	false	(AND): [Level(4), RoGCls(javax.crypto.Cipher)]	992	20	50	1058	47	4.25%	0.054 s	18	32
true	false	(AND): [Level(5), RoGCls(javax.crypto.Cipher)]	992	20	50	1058	47	4.25%	0.054 s	18	32
true	false	(AND): [Level(1), IISCls]	995	17	31	1058	47	4.25%	0.049 s	18	32
true	false	(AND): [Level(2), IISCls]	992	20	38	1053	52	4.71%	0.049 s	18	32
true	false	(AND): [Level(3), IISCls]	992	20	38	1053	52	4.71%	0.048 s	18	32
true	false	(AND): [Level(4), IISCls]	992	20	38	1053	52	4.71%	0.049 s	18	32
true	false	(AND): [Level(5), IISCls]	992	20	38	1053	52	4.71%	0.053 s	18	32
true	false	(AND): [Level(1), IISOrSubPkg]	973	40	68	1048	57	5.16%	0.061 s	19	33
true	false	(AND): [Level(2), IISOrSubPkg]	957	55	103	1051	54	4.89%	0.071 s	19	34
true	false	(AND): [Level(3), IISOrSubPkg]	957	55	103	1051	54	4.89%	0.066 s	19	34
true	false	(AND): [Level(4), IISOrSubPkg]	957	55	103	1051	54	4.89%	0.071 s	19	34
true	false	(AND): [Level(5), IISOrSubPkg]	957	55	103	1051	54	4.89%	0.076 s	19	34
true	false	(OR): [Level(1), RoGPkg(javax.crypto)]	953	61	126	1055	50	4.52%	0.066 s	19	34
true	false	(OR): [Level(2), RoGPkg(javax.crypto)]	945	68	144	1043	62	5.61%	0.074 s	20	35
true	false	(OR): [Level(3), RoGPkg(javax.crypto)]	943	70	149	1043	62	5.61%	0.077 s	20	35
true	false	(OR): [Level(4), RoGPkg(javax.crypto)]	942	70	144	1042	63	5.70%	0.072 s	19	35
true	false	(OR): [Level(5), RoGPkg(javax.crypto)]	942	70	144	1042	63	5.70%	0.077 s	19	35
true	false	(OR): [Level(1), RoGCls(javax.crypto.Cipher)]	957	57	110	1052	53	4.80%	0.069 s	19	34
true	false	(OR): [Level(2), RoGCls(javax.crypto.Cipher)]	945	68	144	1043	62	5.61%	0.070 s	20	35
true	false	(OR): [Level(3), RoGCls(javax.crypto.Cipher)]	943	70	149	1043	62	5.61%	0.068 s	20	35
true	false	(OR): [Level(4), RoGCls(javax.crypto.Cipher)]	942	70	144	1042	63	5.70%	0.071 s	19	35
true	false	(OR): [Level(5), RoGCls(javax.crypto.Cipher)]	942	70	144	1042	63	5.70%	0.067 s	19	35
true	false	(OR): [Level(1), IISCls]	958	55	107	1044	61	5.52%	0.065 s	19	34
true	false	(OR): [Level(2), IISCls]	944	68	142	1043	62	5.61%	0.067 s	19	35
true	false	(OR): [Level(3), IISCls]	942	70	144	1042	63	5.70%	0.069 s	19	35
true	false	(OR): [Level(4), IISCls]	942	70	144	1042	63	5.70%	0.077 s	19	35
true	false	(OR): [Level(5), IISCls]	942	70	144	1042	63	5.70%	0.068 s	19	35
true	false	(OR): [Level(1), IISOrSubPkg]	950	62	126	1042	63	5.70%	0.069 s	19	34

filter	cg	heuristic	t	ees	ings	v	diff	impr	time	n	e
true	false	(OR): [Level(2), IISOrSubPkg]	944	68	142	1043	62	5.61%	0.095 s	19	35
true	false	(OR): [Level(3), IISOrSubPkg]	942	70	144	1042	63	5.70%	0.067 s	19	35
true	false	(OR): [Level(4), IISOrSubPkg]	942	70	144	1042	63	5.70%	0.068 s	19	35
true	false	(OR): [Level(5), IISOrSubPkg]	942	70	144	1042	63	5.70%	0.068 s	19	35
true	false	(AND): [RoGPkg(javax.crypto), IISOrSubPkg]	971	41	81	1064	41	3.71%	0.058 s	19	33
true	false	(AND): [RoGPkg(javax.crypto), IISCls]	995	17	30	1056	49	4.43%	0.047 s	18	32
true	false	(OR): [RoGCls(javax.crypto.Cipher), IISOrSubPkg]	953	59	117	1042	63	5.70%	0.065 s	19	34
true	false	(OR): [RoGCls(javax.crypto.Cipher), IISCls]	979	33	79	1056	49	4.43%	0.054 s	19	33
false	true	AlwaysInline	739	273	947	936	169	15.29%	0.276 s	27	51
false	true	RoGPkg(javax.crypto)	814	198	697	928	177	16.02%	0.204 s	23	42
false	true	RoGCls(javax.crypto.Cipher)	949	63	227	919	186	16.83%	0.156 s	19	35
false	true	IISOrSubPkg	783	229	553	1093	12	1.09%	0.200 s	25	46
false	true	IISCls	935	77	180	1067	38	3.44%	0.158 s	20	35
false	true	Level(1)	827	273	735	1167	-62	-5.61%	0.204 s	25	46
false	true	Level(2)	759	273	936	1023	82	7.42%	0.215 s	27	51
false	true	Level(3)	743	273	954	951	154	13.94%	0.233 s	27	51
false	true	Level(4)	739	273	947	936	169	15.29%	0.229 s	27	51
false	true	Level(5)	739	273	947	936	169	15.29%	0.224 s	27	51
false	true	(AND): [Level(1), RoGPkg(javax.crypto)]	877	198	576	1129	-24	-2.17%	0.173 s	22	40
false	true	(AND): [Level(2), RoGPkg(javax.crypto)]	827	198	697	990	115	10.41%	0.189 s	23	42
false	true	(AND): [Level(3), RoGPkg(javax.crypto)]	815	198	699	934	171	15.48%	0.187 s	23	42
false	true	(AND): [Level(4), RoGPkg(javax.crypto)]	814	198	697	928	177	16.02%	0.187 s	23	42
false	true	(AND): [Level(5), RoGPkg(javax.crypto)]	814	198	697	928	177	16.02%	0.184 s	23	42
false	true	(AND): [Level(1), RoGCls(javax.crypto.Cipher)]	972	63	158	1089	16	1.45%	0.151 s	19	34
false	true	(AND): [Level(2), RoGCls(javax.crypto.Cipher)]	954	63	227	974	131	11.86%	0.158 s	19	35
false	true	(AND): [Level(3), RoGCls(javax.crypto.Cipher)]	949	63	227	919	186	16.83%	0.160 s	19	35
false	true	(AND): [Level(4), RoGCls(javax.crypto.Cipher)]	949	63	227	919	186	16.83%	0.157 s	19	35
false	true	(AND): [Level(5), RoGCls(javax.crypto.Cipher)]	949	63	227	919	186	16.83%	0.155 s	19	35
false	true	(AND): [Level(1), IISCls]	944	77	177	1086	19	1.72%	0.149 s	19	34
false	true	(AND): [Level(2), IISCls]	935	77	180	1067	38	3.44%	0.152 s	20	35
false	true	(AND): [Level(3), IISCls]	935	77	180	1067	38	3.44%	0.162 s	20	35
false	true	(AND): [Level(4), IISCls]	935	77	180	1067	38	3.44%	0.151 s	20	35
false	true	(AND): [Level(5), IISCls]	935	77	180	1067	38	3.44%	0.150 s	20	35
false	true	(AND): [Level(1), IISOrSubPkg]	824	229	501	1139	-34	-3.08%	0.180 s	24	44
false	true	(AND): [Level(2), IISOrSubPkg]	786	229	553	1098	7	0.63%	0.195 s	25	46
false	true	(AND): [Level(3), IISOrSubPkg]	783	229	553	1093	12	1.09%	0.198 s	25	46
false	true	(AND): [Level(4), IISOrSubPkg]	783	229	553	1093	12	1.09%	0.204 s	25	46
false	true	(AND): [Level(5), IISOrSubPkg]	783	229	553	1093	12	1.09%	0.200 s	25	46
false	true	(OR): [Level(1), RoGPkg(javax.crypto)]	759	273	937	960	145	13.12%	0.220 s	27	50
false	true	(OR): [Level(2), RoGPkg(javax.crypto)]	742	273	955	937	168	15.20%	0.222 s	27	51
false	true	(OR): [Level(3), RoGPkg(javax.crypto)]	740	273	952	937	168	15.20%	0.236 s	27	51
false	true	(OR): [Level(4), RoGPkg(javax.crypto)]	739	273	947	936	169	15.29%	0.233 s	27	51
false	true	(OR): [Level(5), RoGPkg(javax.crypto)]	739	273	947	936	169	15.29%	0.222 s	27	51
false	true	(OR): [Level(1), RoGCls(javax.crypto.Cipher)]	800	273	814	980	125	11.31%	0.205 s	26	47
false	true	(OR): [Level(2), RoGCls(javax.crypto.Cipher)]	749	273	952	948	157	14.21%	0.220 s	27	51
false	true	(OR): [Level(3), RoGCls(javax.crypto.Cipher)]	742	273	964	945	160	14.48%	0.230 s	27	51
false	true	(OR): [Level(4), RoGCls(javax.crypto.Cipher)]	739	273	947	936	169	15.29%	0.222 s	27	51
false	true	(OR): [Level(5), RoGCls(javax.crypto.Cipher)]	739	273	947	936	169	15.29%	0.232 s	27	51
false	true	(OR): [Level(1), IISCls]	797	273	778	1136	-31	-2.81%	0.207 s	26	49
false	true	(OR): [Level(2), IISCls]	755	273	932	1011	94	8.51%	0.218 s	27	51
false	true	(OR): [Level(3), IISCls]	741	273	947	944	161	14.57%	0.224 s	27	51
false	true	(OR): [Level(4), IISCls]	739	273	947	936	169	15.29%	0.230 s	27	51
false	true	(OR): [Level(5), IISCls]	739	273	947	936	169	15.29%	0.228 s	27	51
false	true	(OR): [Level(1), IISOrSubPkg]	782	273	818	1105	0	0.00%	0.216 s	27	50
false	true	(OR): [Level(2), IISOrSubPkg]	752	273	934	1003	102	9.23%	0.222 s	27	51
false	true	(OR): [Level(3), IISOrSubPkg]	741	273	947	944	161	14.57%	0.221 s	27	51
false	true	(OR): [Level(4), IISOrSubPkg]	739	273	947	936	169	15.29%	0.229 s	27	51
false	true	(OR): [Level(5), IISOrSubPkg]	739	273	947	936	169	15.29%	0.229 s	27	51
false	true	(AND): [RoGPkg(javax.crypto), IISOrSubPkg]	843	169	414	1093	12	1.09%	0.180 s	22	41
false	true	(AND): [RoGPkg(javax.crypto), IISCls]	949	63	159	1068	37	3.35%	0.151 s	19	34
false	true	(OR): [RoGCls(javax.crypto.Cipher), IISOrSubPkg]	773	239	660	927	178	16.11%	0.211 s	25	47
false	true	(OR): [RoGCls(javax.crypto.Cipher), IISCls]	893	119	379	921	184	16.65%	0.172 s	21	38
true	true	AlwaysInline	874	138	428	924	181	16.38%	0.235 s	22	41
true	true	RoGPkg(javax.crypto)	926	86	304	919	186	16.83%	0.198 s	20	36
true	true	RoGCls(javax.crypto.Cipher)	968	44	192	912	193	17.47%	0.156 s	19	34
true	true	IISOrSubPkg	949	63	128	1055	50	4.52%	0.202 s	19	35
true	true	IISCls	991	21	39	1049	56	5.07%	0.151 s	18	32

filter	cg	heuristic	t	ees	ings	v	diff	impr	time	n	e
true	true	Level(1)	955	61	125	1052	53	4.80%	0.196 s	19	34
true	true	Level(2)	901	117	350	969	136	12.31%	0.220 s	21	40
true	true	Level(3)	880	135	419	931	174	15.75%	0.227 s	22	41
true	true	Level(4)	874	138	428	924	181	16.38%	0.233 s	22	41
true	true	Level(5)	874	138	428	924	181	16.38%	0.224 s	22	41
true	true	(AND): [Level(1), RoGPkg(javax.crypto)]	973	41	95	1062	43	3.89%	0.176 s	19	33
true	true	(AND): [Level(2), RoGPkg(javax.crypto)]	935	81	270	970	135	12.22%	0.192 s	20	36
true	true	(AND): [Level(3), RoGPkg(javax.crypto)]	927	86	306	925	180	16.29%	0.186 s	20	36
true	true	(AND): [Level(4), RoGPkg(javax.crypto)]	926	86	304	919	186	16.83%	0.186 s	20	36
true	true	(AND): [Level(5), RoGPkg(javax.crypto)]	926	86	304	919	186	16.83%	0.185 s	20	36
true	true	(AND): [Level(1), RoGCls(javax.crypto.Cipher)]	995	18	55	1062	43	3.89%	0.154 s	18	32
true	true	(AND): [Level(2), RoGCls(javax.crypto.Cipher)]	975	40	172	961	144	13.03%	0.157 s	19	34
true	true	(AND): [Level(3), RoGCls(javax.crypto.Cipher)]	968	44	192	912	193	17.47%	0.157 s	19	34
true	true	(AND): [Level(4), RoGCls(javax.crypto.Cipher)]	968	44	192	912	193	17.47%	0.162 s	19	34
true	true	(AND): [Level(5), RoGCls(javax.crypto.Cipher)]	968	44	192	912	193	17.47%	0.156 s	19	34
true	true	(AND): [Level(1), IISCls]	994	18	32	1054	51	4.62%	0.153 s	18	32
true	true	(AND): [Level(2), IISCls]	991	21	39	1049	56	5.07%	0.153 s	18	32
true	true	(AND): [Level(3), IISCls]	991	21	39	1049	56	5.07%	0.151 s	18	32
true	true	(AND): [Level(4), IISCls]	991	21	39	1049	56	5.07%	0.159 s	18	32
true	true	(AND): [Level(5), IISCls]	991	21	39	1049	56	5.07%	0.152 s	18	32
true	true	(AND): [Level(1), IISOrSubPkg]	971	43	76	1056	49	4.43%	0.184 s	19	33
true	true	(AND): [Level(2), IISOrSubPkg]	950	63	126	1059	46	4.16%	0.197 s	20	35
true	true	(AND): [Level(3), IISOrSubPkg]	949	63	128	1055	50	4.52%	0.203 s	19	35
true	true	(AND): [Level(4), IISOrSubPkg]	949	63	128	1055	50	4.52%	0.208 s	19	35
true	true	(AND): [Level(5), IISOrSubPkg]	949	63	128	1055	50	4.52%	0.208 s	19	35
true	true	(OR): [Level(1), RoGPkg(javax.crypto)]	887	127	396	933	172	15.57%	0.225 s	21	40
true	true	(OR): [Level(2), RoGPkg(javax.crypto)]	877	136	428	925	180	16.29%	0.232 s	22	41
true	true	(OR): [Level(3), RoGPkg(javax.crypto)]	875	138	433	925	180	16.29%	0.240 s	22	41
true	true	(OR): [Level(4), RoGPkg(javax.crypto)]	874	138	428	924	181	16.38%	0.235 s	22	41
true	true	(OR): [Level(5), RoGPkg(javax.crypto)]	874	138	428	924	181	16.38%	0.232 s	22	41
true	true	(OR): [Level(1), RoGCls(javax.crypto.Cipher)]	906	110	313	934	171	15.48%	0.206 s	21	38
true	true	(OR): [Level(2), RoGCls(javax.crypto.Cipher)]	880	135	417	933	172	15.57%	0.224 s	22	41
true	true	(OR): [Level(3), RoGCls(javax.crypto.Cipher)]	877	138	445	933	172	15.57%	0.225 s	22	42
true	true	(OR): [Level(4), RoGCls(javax.crypto.Cipher)]	874	138	428	924	181	16.38%	0.227 s	22	41
true	true	(OR): [Level(5), RoGCls(javax.crypto.Cipher)]	874	138	428	924	181	16.38%	0.228 s	22	41
true	true	(OR): [Level(1), IISCls]	935	81	175	1059	46	4.16%	0.210 s	20	35
true	true	(OR): [Level(2), IISCls]	898	118	349	958	147	13.30%	0.222 s	21	40
true	true	(OR): [Level(3), IISCls]	878	135	412	924	181	16.38%	0.226 s	22	41
true	true	(OR): [Level(4), IISCls]	874	138	428	924	181	16.38%	0.225 s	22	41
true	true	(OR): [Level(5), IISCls]	874	138	428	924	181	16.38%	0.225 s	22	41
true	true	(OR): [Level(1), IISOrSubPkg]	927	88	202	1053	52	4.71%	0.216 s	20	36
true	true	(OR): [Level(2), IISOrSubPkg]	897	118	351	954	151	13.67%	0.232 s	21	40
true	true	(OR): [Level(3), IISOrSubPkg]	878	135	412	924	181	16.38%	0.227 s	22	41
true	true	(OR): [Level(4), IISOrSubPkg]	874	138	428	924	181	16.38%	0.226 s	22	41
true	true	(OR): [Level(5), IISOrSubPkg]	874	138	428	924	181	16.38%	0.227 s	22	41
true	true	(AND): [RoGPkg(javax.crypto), IISOrSubPkg]	971	41	85	1072	33	2.99%	0.183 s	19	33
true	true	(AND): [RoGPkg(javax.crypto), IISCls]	995	17	30	1056	49	4.43%	0.153 s	18	32
true	true	(OR): [RoGCls(javax.crypto.Cipher), IISOrSubPkg]	911	101	317	911	194	17.56%	0.213 s	21	39
true	true	(OR): [RoGCls(javax.crypto.Cipher), IISCls]	952	60	226	906	199	18.01%	0.179 s	19	35