Master Thesis

Graph-based Classification for Detecting Instances of Bug Patterns

Giacomo Iadarola

Andrew Habib
Advising assistant

Prof. Dr. Michael Pradel
Software Lab
TU Darmstadt

September 2018

TECHNISCHE
UNIVERSITÄT
DARMSTADT

SOLA
SoftwareLab

Abstract

Hundreds of software handle data, money transactions and information every day, for every person in the world. Thus, also the smallest vulnerability may lead to a domino effect. In order to mitigate these bad consequences as much as possible, there is a considerable demand for bug finder tools that can help developers debug their code and improve security and correctness. Nevertheless, none of them is perfect, and most of the available tools can detect only generic errors that contradict common behavior or make use of a list of pre-defined rules and patterns that may constitute a vulnerability. These tools cannot easily be extended to more specific bug patterns without a tedious and complicated study on the bug itself and its causes. We propose an approach to developing a generic bug finder tool that uses machine learning and trains a model able to classify buggy and non-buggy code by looking at a dataset of buggy examples for a specific bug pattern. Our approach applies static analyses to represent source code as graphs and then uses a multilayer perceptron model to perform the classification task. We report the results of our experiments in detecting Null Pointer Exceptions in Java codes. The evaluation results are promising and confirm that machine learning can help improve code security and develop better bug finder tools.

Zusammenfassung

Hunderte von Softwareapplikationen verarbeiten tglich Daten, fhren Geldtransaktionen aus und behandeln Informationen fr jede Person auf der Welt. Somit kann auch die kleinste Sicherheitslcke zu einem Dominoeffekt fhren. Um diese nachteiligen Folgen so weit wie mglich zu mildern, gibt es eine betrchtliche Nachfrage nach Bug-Finder-Tools, die den Entwicklern helfen knnen, ihren Code zu debuggen und die Codesicherheit zu verbessern. Allerdings ist keine von ihnen perfekt und die meisten der verfgbaren Tools knnen nur allgemeine Fehler erkennen, die blichem Verhalten widersprechen, oder sie verwenden eine Liste vordefinierter Regeln und Muster, die eine Sicherheitslcke auslsen knnen. Sie knnen nicht ohne weiteres auf ein spezifischeres Fehlermuster erweitert werden, denn sie bentigen eine langwierige und komplizierte Untersuchung des Fehlers und seiner Ursachen. Wir schlagen einen Ansatz zur Entwicklung eines generischen Bug-Finder-Tools vor, der maschinelles Lernen verwendet und ein Modell trainiert, das in der Lage ist, fehlerhaften und nicht-fehlerhaften Code zu klassifizieren, indem ein Dataset fehlerhafter Beispiele fr ein spezifisches Bug-Muster betrachtet wird. Unser Ansatz wendet statische Analysen an, um Quellcode als Graphen darzustellen und verwendet dann ein mehrschichtiges Perzeptronmodell, um die Klassifizierungsaufgabe durchzufhren. Wir melden das Ergebnis unseres Experiments mit dem Null Pointer Exception Fehler aus Java. Unser maschinelles Lernmodell war in der Lage, zwischen Code zu unterscheiden, der eine Null Pointer Exception auslsen knnte, ohne es eine Liste vordefinierter Muster zu liefern, die diesen Fehler auslsen knnten, sondern es stattdessen auf einem Datensatz fehlerhafter Beispiele zu trainieren. Die Evaluierungsergebnisse sind vielversprechend und besttigen, dass das maschinelle Lernen dabei helfen kann, die Codesicherheit zu verbessern und bessere Bugfinder-Tools zu entwickeln.

Contents

1 Introduction

We all live in a connected world, and software constitute the essential foundation of our society. The digital world profoundly influences our daily lives, through smartphones, PCs, smart clothes, domestic appliances and many other technologies that populate our cities. This connected network of apps and intelligent tools play a role in our choices, social activity, and daily routine. As for any other key role in the modern society, the digital world requires a high level of security. Moreover, it changes so fast that the research and ideation of the best security approaches and policies is one of the most significant open problems of our time.

All our information, decisions, requests and choices flow through thousands and thousands of code lines, which perform computations and then output a result, hopefully one that is useful for us. In a utopian world, everything goes perfectly fine, and each piece of software always does what it is programmed to do. In reality, software is full of bugs and errors which guide the execution to wrong paths and problems.

Software is still the product of human developers. Not even the most accurate debugging process can produce a bug-free code since it is inherently impossible to do so. Nevertheless, we should aim at the best quality for our systems and software, because even small mistake may produce significant damage through a domino effect. In December 2017, a Russian newly-launched weather satellite (costs around 58$ million) was lost due to an embarrassing programming error; the rocket was programmed with the wrong coordinates as if it was taking off from a different cosmodrome [69]. This fact confirms how much our society relies on software, and remind us we should spend more time improving their quality and security.

Human mistakes are not the only problem: when a the vulnerability is found by malicious hackers, they use it to exploit the software and perform actions that should not be allowed.

The first defense against software vulnerabilities is to test and debug the code to produce a more secure and reliable product. Nonetheless, debugging code is a tedious and challenging job. Recent work by Beller et al. [28] reports that developers estimate that testing takes 50% of their time, but they overestimate this percentage. In fact, the research proves that they do not spend more than 25% of their time testing. Indeed, this fact confirms that most software engineers are aware of the impact that testing has on their code and aim to test more. However, bugs are complex and nested, making the testing process slow and tiresome. Without guidance, the same developers that would like to test more end up testing less than they estimated.

Handling large software can be messy, and code is often produced by teams of several developers with different skills and coding styles. Therefore, the testing process needs to follow strict rules to be efficient. Developers are always looking for new methods that can help them debug their code.

1.1 The Problem

There is a huge demand for bug finder tools, and several tools and methods were proposed to help developers find bugs and fix them (see Chapter 8). The problem is enormous, and any step towards the automatizing of this process constitutes a considerable help in improving the software quality. Because of this, in addition to traditional software testing and manual detection of bugs by the developers, automated bug checkers that analyze source code and detect mistakes are becoming more and more popular.
Several kinds of bug finder are in use and they can be roughly classified into three categories:

Pattern-based. Pattern-based approaches use a pre-defined list of bug patterns and analyses to detect them [51].

Belief-based. Belief-based approaches infer "program beliefs" from one code location and pinpoint other code locations that seem to contradict the assumed beliefs [38, 61].

Anomaly-based. Anomaly-based approaches learn some properties and invariants from the code and then flag anomalies that may be due to bugs [65, 68, 78, 70].

The main limitation shared by all of these approaches is that they only work for specific kinds of bug patterns. Extending the set of supported profiles would require modifying the inner-working of the software. Most of these tools can pinpoint code locations that seem to contradict common behaviors or detect anomalies by comparing the code with a list of rules and pre-defined bug patterns that may lead to vulnerabilities. They are not able to look for a particular bug pattern without a list of "if condition" that helps the understanding how that bug is usually triggered. Additionally, acquiring detailed knowledge on bugs and generating a pre-defined list of rules is a tedious and complex task that requires many hours of work for testers. This limitation makes difficult to adapt existing tools to additional bug patterns without modifying their underlying analyses.
The goal of this thesis is to introduce a general bug finder that learns how to distinguish correct code from incorrect one. The training dataset contains different instances of the same bug pattern, so that the bug finder can learn and specialize itself on a specific bug and then classify code as buggy or not in regard to that particular bug pattern. Our approach is general and does not require any prior-knowledge of the code so that it can be applied to every kind of bugs. The main strength resides in the fact that the bug finder is specialized in one bug pattern at a time, and this improves the accuracy and efficiency in finding that particular error. Moreover, the approach can be extended to more bug patterns by providing a dataset of buggy examples for that specific bug; it does not require detailed knowledge on the bug itself or a pre-defined list of rules for detecting that particular vulnerability.
To achieve this goal, we developed a tool called GrapPa (from the first and last word of the title), which analyzes and then classifies Java source code as buggy relatively to a specific bug pattern. GrapPa aims to help developers check their code and find threats and vulnerabilities. It is based on several frameworks, libraries (Soot [16, 77] and Keras [11], see also Section 4.2 and 4.5), and external tools (CGMM [7], see also Section 4.3).

1.2 Thesis Structure

The thesis is structured as follows: Chapter 2 contains a general overview of the knowledge and theories required to fully understand our approach. The approach is then introduced in details in Chapter 3. Chapter 4 presents technical details about the implementation of the GrapPa tool. The results of the experiments we conducted are reported in Chapter 5, a short discussion is reported in Chapter 6. Chapter 7 presents some limitations, future improvements and ideas for our approach, while Chapter 8 reports some related works. Finally, the thesis concludes with a short summary of the entire work in Chapter 9.

2 Background

This section introduces the required knowledge to understand our approach.
The approach, introduced in Chapter 3, can be briefly summarized into three steps, and the following sections present a brief overview about the theory on which these steps are based:

Static Analysis. First, a static analysis extracts properties of each piece of source code and summarizes these properties into a Code Property graph (CPG), introduced in subsection 2.1.4. This graph merges concepts of classic program analysis, namely abstract syntax trees, control flow graphs and program dependence graphs, into a joint data structure. The subsections 2.1.1, 2.1.2, and 2.1.3 describe them in details.

Vectorization. Second, the graphs are vectorized by an unsupervised machine learning model, introduced in subsection 2.2.

Classification. Finally, two graph-based machine learning classifiers, introduced in Section 2.3, are trained with examples of graphs from buggy and non-buggy code, so that they learn how to distinguish those two kinds of graphs. Then, the trained models take as input previously unseen graphs, classify them and provide suggestions useful to detect vulnerabilities in the code.

2.1 Source Code Static Analysis

The static analysis of software is the analysis performed without actually executing that software, in contrast with dynamic analysis, which is performed while the software is running. This kind of analysis is either performed directly on the source code or some abstraction of the code. In our approach, we analyze directly the source code of the program.
Performing analysis on a piece of code means extracting properties which hold in some part of the code. This kind of analysis should be fast, extract useful information and have a high coverage, which can help detect bugs and errors in the code. Several methods were proposed, which perform sophisticated analysis and use the information obtained for different purposes, which vary from highlighting possible errors to formal methods that mathematically prove properties of a given program.
Static analysis is also used to extract the information necessary to represent the program into some data structure (i.e., a graph) which can highlight the interaction between its parts and so improves further analysis and studies. These representations were designed to analyze and optimize the

Listing 2.1: Jimple Example

```
1 public static void main(java.lang.String[])
2 {
3     java.lang.String[] args;
4     int x, temp$0, temp$1;
5
6     args := @parameter0: java.lang.String[];
7     x = 0;
8     if x < 3 goto label0;
9
10    goto label1;
11
12  label0:
13    nop;
14    temp$0 = x;
15    temp$1 = temp$0 + 1;
16    x = temp$1;
17
18  label1:
19    nop;
20    return;
21 }
```

code, so that it can be reproduced and enriched with further information. That is the case of our approach, which uses static analysis to extract information from a piece of code and then represents it as a graph, by drawing the dependencies and the relations between the different entities of code. Various graph representations of code have been developed in the field of program analysis. In particular, we focus on three classic representations, namely abstract syntax trees (AST), control flow graphs (CFG) and program dependence graphs (PDG) which form the basis for the CPG and so our graph representation.

We perform the analysis on code written in Jimple (Java sIMPLE), an intermediate three-address representation of a Java program, designed to be easier to optimize than Java bytecode [83]. Jimple includes only 15 different operations, while in comparison the java bytecode includes over 200 operations. Jimple is the intermediate language used by the Soot framework, used for static analysis by our tool GrapPa (see Section 4.2). The Listing 2.1 shows a basic example in Jimple, which will be referenced in the next subsection to introduce the AST, CFG, PDG and the CPG. Listing 2.2 shows the same code of Listing 2.1 but written in Java language instead.

Due to the short number of operations, the Java code is smaller than the same code converted to Jimple. However, the example code contains just a few statements and should be relatively easy to link a statement in Java to the same statement in Jimple. The snippet in Listing 2.2 contains a declaration at line 2, an if condition at line 4 and an assignment at line 6. These three statements can be linked to the statements in Listing 2.1 at line 4 and 7 for the declaration, 8 for the if condition and from line 14 to line 16 for the assignment. In the next subsections, the Listing 2.1 is used to introduce the graph representations and to guide the reader towards the comprehension of

Listing 2.2: Java Example

```
1 public static void main(String[] args)
2 {
3     int x = 0;
4         if (x < 3)
5         {
6             x++;
7         }
8 }
```

their strengths and weakness.

We introduce some definitions regarding concepts of classic program analysis. We represent source code as a graph, and we define a graph as a tuple $g = (\mathcal{V}, \mathcal{E})$ formed by a set of nodes $\mathcal{V}$ and a set of edges $\mathcal{E}$. The edges (s_i, d_i) are direct, where s_i represents the source node and d_i the destination node of the edge. A node v corresponds to a basic block, a single statement or a token. We define a control flow edge as a direct edge (s, d) if there is a flow of control from the statement represented by the node s to the statement represented by the node d in the source code. For instance, by looking at Listing 2.1, the statement `x=0;` at line 7 and the statement `if x < 3 goto label0;` at line 8 would be connected by a control flow edge in a graph representation, because they are one after the other in the flow execution of the program.

We define a data dependency edge (also data flow dependency edge) as a direct edge (s, d) between two statements represented by the node s and d when the statement d makes use of a variable defined by the statement s. For instance, there is a data dependencies between the statement at line 4 to the statement at line 7 in the Listing 2.1. Finally, we define a control dependency edge (also control flow dependency edge) as a direct edge (s, d) between two statements represented by the node s and d, when the execution of the statement d is conditionally guarded by s. For instance, all the statements between line 12 and line 16 in the Listing 2.1 are control dependent on the statement at line 8, because the if condition regulates their execution.

2.1.1 Abstract Syntax Tree

Abstract syntax trees are one type of graph representation for source code. Usually, an AST is the first intermediate representation produced by the code parser of compilers and form the basis for the generation of other code representations [86, 80].

It represents the abstract syntax structure of the source code and captures the essential information of the input in an ordered tree where inner nodes represent operators and leaf nodes correspond to operands. In computer science, ASTs are needed because languages are often ambiguous, while the ASTs help the compiler understand the input source code.

Figure 2.1 shows an extract of the AST generated by Listing 2.1. The complete AST is shown in Figure A.2 and Figure A.3 in the Appendix for layout reasons. The subgraph shown in Figure 2.1 comes from the assignment statement at line 7. The red node represents the statement, while the other nodes are syntax constructs of the language, and the leaves represent the tokens. By

traversing this kind of tree, the compiler and a tester are able to interpret the tokens and validate the syntax correctness. Each node contains a number in the name field, this is the unique ID of the node, which regulates the visit order of the AST (for instance, the ID of the statement node is 60). By sorting the node ID of the tokens, we can reconstruct the original statement of the code. For instance, we can sort the tokens of of Figure 2.1 and reconstruct the statement `x=0;`.

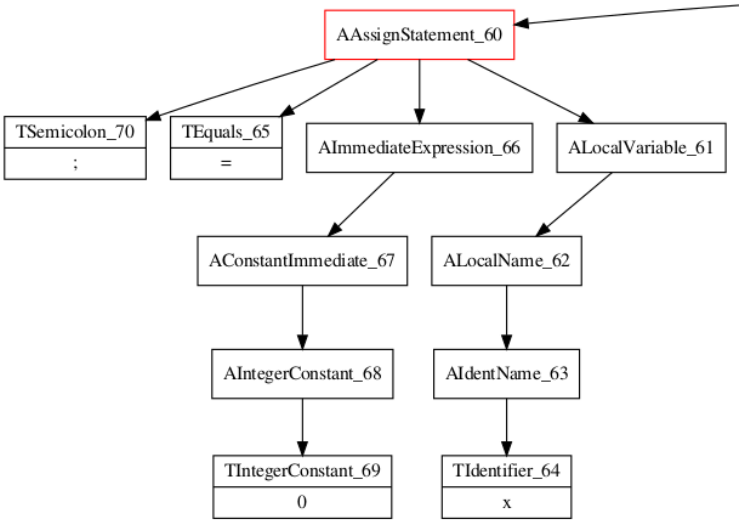


Figure 2.1: Extract of the AST Listing 2.1.

ASTs are well suited for simple code transformation, but are not applicable for more sophisticated code analysis, because make explicit neither the control flow nor data and control dependencies of the statements.

2.1.2 Control Flow Graph

Control flow graph represents all the paths that might be traversed during a program execution [81, 86]. A CFG has an entry node, through which control enters into the flow graph, and an exit node, through which all control flow leaves. Each node in the graph represents a single statement, and direct edges are used to represent a transition in the control flow. In short, it describes the order in which code statements are executed and also the conditions that needs to be met for a particular path of execution to be taken.
The Figure 2.2 shows the CFG for Listing 2.1. The two grey nodes at the beginning and the end of the graph constitute the Entry and Exit nodes. As we can see, the if statement produces a fork into the graph paths, where the path choice depends on the evaluation of the if condition.

CFG is a standard code representation used in program analysis. Compared to an AST, the CFG gives control flow information, but still fails to provide data and control dependencies.

2.1.3 Program Dependence Graph

A Program Dependence Graph is a graph representation that makes explicit both data and control dependencies for each operation in a program. The PDG used by our tool is an intra-procedural

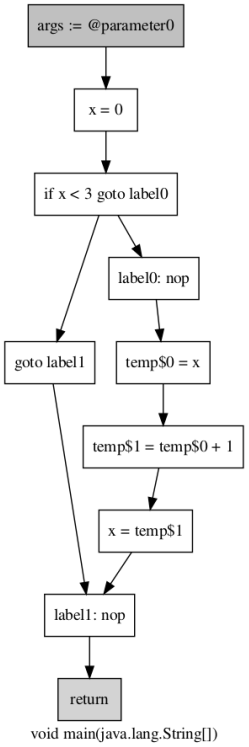


Figure 2.2: CFG of Listing 2.1.

dependence graph, which models dependencies within a procedure but does not say anything about external procedures called by the program.

The usual representation for the control dependencies of a program is the Control Flow Graph (CFG), as already discussed in subsection 2.1.2, even though it does not contain information about data dependencies. To address this problem, the PDG was presented by Ferrante J. [39]. The main motivation was to develop a program representation useful in optimizing compilers and detecting parallel operations. Indeed, a PDG is used to detect possible operations that can be parallelized to improve efficiency at runtime, but the model is also useful in other contexts, such as slicing.

The nodes of a PDG roughly correspond to the program statements, while the edges model dependencies in the program. A direct edge represents a dependence between two nodes, which can be classified as either control or data dependency. Figure 2.3 shows an extract of the PDG generated for Listing 2.1. The three nodes shown in Figure 2.3 correspond to the statements at line 7, 8 and 9, respectively. The yellow edges are the dependencies, marked as DATA or CONTROL. The node `AIfStatement_71` represents the if statement `if x < 3 goto label0;`, and has a data dependency to the node `AAssignStatement_60`, which represents the assignment statement `x = 0;`, because the if condition uses the variable x. Differently, the label statement `label0:` (node `ALabelStatement_95`) is control dependent on the if statement because its execution depends on the if condition.

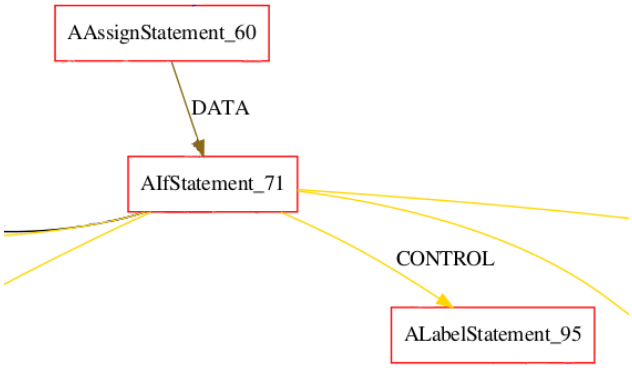


Figure 2.3: Extract of PDG generated for Listing 2.1.

2.1.4 Code Property Graph

A Code Property Graph is a source code representation that combines an AST, a CFG, and a PDG into one data structure. In the original paper [86], the authors present this novel representation as a solution for addressing the problem of graph mining. The main idea underlying this approach is that many vulnerabilities and errors can be discovered only by taking into account both the control flow and the data dependencies of the code. By combining different concepts of classic program analysis, a CPG makes possible to find common vulnerabilities using graph traversal.
First of all, the nodes of the CFG are copied to the CPG. The CFG nodes constitute the base of the CPG, in fact, all the AST and PDG elements are connected to them. Secondly, the AST edges and nodes are added to the CFG statement nodes in the CPG. Finally, the PDG edges (data dependencies and control dependencies) are added to the graph. The CPG is a multi-graph, meaning that two vertices may be connected by more than one edge, where each edge provides different information among control flow, data dependency or control dependency.
Figure 2.4 shows an extract of the CPG generated for Listing 2.1, the complete graph is reported in the Appendix as Figure A.4.

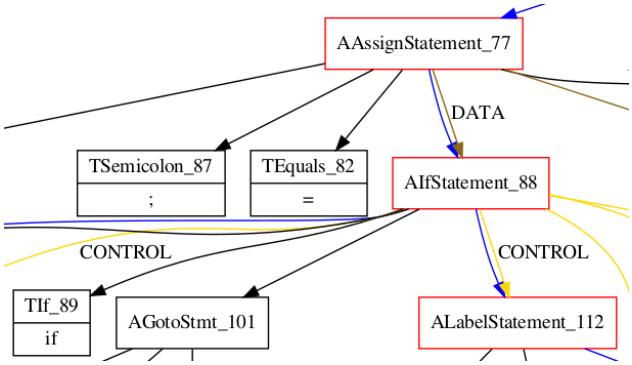


Figure 2.4: Extract of CPG generated for Listing 2.1.

The CPG extract and the PDG extract shown in Figure 2.3 represent the same three statements at line 7, 8 and 10 in Listing 2.1. By comparing these two figures, we can see the addition of the control flow edges taken from the CFG (the blue edges), and the AST nodes and edges (the black

nodes and edges).
The CPG combines the information provided by the AST, CFG, and PDG into a joint data structure, which makes the representation strong and sound for code analysis. Our tool GrapPa uses a modified version of the original CPG, which simplifies the graph structure to remove information that is redundant for our purpose. This simplified version is introduced in Section 3.2.

2.2 Contextual Graph Markov Model

Contextual Graph Markov Models (CGMM) is a recent approach to graph data processing and combines ideas from generative models and neural networks [27]. We use CGMMs to vectorize graphs [7].
Vectorization is an important step and strongly influences the final result of our solution. Each graph has a different dimension but has to be compressed into a one-dimensional array of fixed dimension. Hence, the machine learning model that performs the vectorization tries to preserve as much information as possible. The CGMM tool applies an unsupervised machine learning model to convert the data into vectors and addressing the vectorization task.
The CGMM approach takes a dataset of graphs as input and applies layers of hidden variables to encode their structural information, using diffusion from neighbor nodes. It produces an unsupervised encoder able to encode graphs of varying size and topology into a fixed dimension vector. The encoding obtained from the unsupervised CGMM can then be used as input for a standard classifier or regressor when performing supervised tasks.
The paper by Errica F. et al. [27] introduces this novel approach. The model is randomly initialized and then trained one layer at a time. At level one, the hidden states are assigned without considering any context, except for the vertex label. At the next iteration, vertexes have information concerning their direct neighbors. Then, for each successive iteration, the information propagates by one neighbor farther at a time. This process allows an effective context propagation from each node of the graph, provided that a sufficient number of layers is used. Finally, the encoding of the graph is produced as a vector of state frequency counts for each layer. The layers are concatenated into a fixed-size vector summarizing contributions from each one. The number of layers can be set as a parameter or calculated with a model selection: after the addition of a new layer, a supervised model is trained and tested using the current graph encoding as input, and a new layer is added only if the last one has a positive effect on the performance.

2.3 Graph-based classifier

Graphs are a well-know representation for classification tasks. Classifying graphs is a classical problem in several research fields, not just computer science (see Chapter 8). Our solution makes use of two different classifiers, called Random Forest (RF) and Multilayer Perceptron (MP).

2.3.1 Random Forest

The Random Forest model is a decision tree model introduced by Leo Breiman [29, 67]. A standard decision tree model algorithm constructs a tree graph of decisions and their possible consequences based on the training phase. Nodes contain conditional control statements, and the execution is guided by the input, which influences path decisions. The classification task is completed by traversing the decision tree using the input sample starting from the root node until it reaches a leaf, which contains the output.
The Random Forest model is included in the "ensemble learning methods", that generate classifiers (in this case, decision trees) and then aggregate and select the best one based on their result. One generation technique, on which the RF model is based, is called "bagging". It produces different trees that are independently constructed using a subset sample of the training data set. As final step, a simple majority vote is taken for prediction.
The construction of trees in the Breiman RF proposal differs from the bagging generation method by adding one more random step. In fact, for each node this last step generates the condition that split the graph into two sub-graphs by taking into account a set of predictors randomly chosen. In a classical decision tree model, each node separates the dataset using the best split criterion.
The addition of a random step improves the performance of the classifier compared to other neural networks and support vector machines [29], and make it less prone to overfitting.

2.3.2 Multilayer Perceptron

In our approach, we use a network model called multilayer perceptron (MLP) [49]. MLP is one of the most used feedforward artificial neural network models. The neural network consists of at least three layers of nodes, where each node represents a neuron and contains a nonlinear activation function. The first layer of nodes handles the input vector, and get the name of the input layer, while the last layer is called the output layer. Hidden layers, at least one for each MLP model, connect the input layer to the output layer. Empirical tests are performed to find the best hyper-parameters for the model in terms of number of layers and neurons. Figure 2.5 shows an example of a MLP structure with one hidden layer and one node in the output layer.

The layers can have different activation function and purposes. Among all the possibilities, we cite the ones used in our approach: the Rectified Linear Unit (Relu), the Dropout and the Signmoid function. The Rectified Linear Unit is defined as

$$relu(x) = \begin{cases} x & if x > 0 \\ 0 & if x \leq 0 \end{cases}$$

where x is the input tensor.
Dropout is a regularization technique aiming to reduce the complexity of the model and prevent overfitting. At training time, the dropout layer randomly selects neurons that are ignored, and sets a fraction `rate` of input units to 0 at each update. Consequently, the neurons contribution to the activation of downstream neurons is temporally removed on the forward pass and weight updates are not applied to the neuron on the backward pass. The `rate` of input units is usually set to the

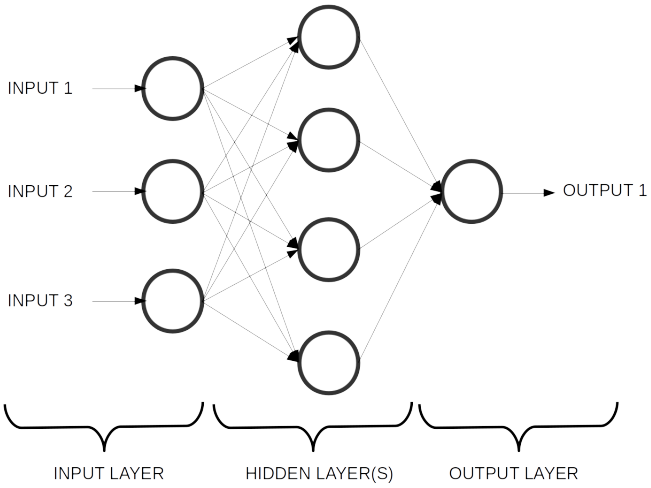


Figure 2.5: MLP example with one hidden layer and one output node.

default value of 0.5, so half of the input neurons, randomly selected, output 0, while the other half propagate the input value to the next layer.
The sigmoid function is defined as

$$S(x) = \frac{1}{1 + e^{-x}}$$

and outputs values in the range $[0, 1]$. It is commonly used as the activation function in the output layer.
The neural network is fully connected, which means that each node in one layer connects with a specific weight to every node in the following layer. Weights are taken into account during the training phase and are set by a supervised learning technique called backpropagation.
Briefly, this technique can be split into two steps: a forward pass and a backward pass. The goal is to optimize the weights so that the networks can learn how to classify inputs to outputs correctly. In the forward pass, an input is provided and then propagated through the hidden layers to the output layer, which outputs a result vector. Since the expected output (target output) is known, it can be compared to the actual output to calculate the total error. This step only applies the model to a specific input to get an output, and there is no learning involved. The second step (backward pass) provides the real training part. The goal of this second step is to update each weight of the network so that the actual output is closer to the target output and the total error is minimized. The actual output is then back-propagated from the output layer to the input layer and the weights are updated. The two steps are repeated for any sample in the training set, and then the model is considered trained. It can then be applied to previously unseen examples to classify them.

2.3.2.1 Representing Model Uncertainty

Machine learning models are widely used, however, they are usually not able to quantify the uncertainty of their predictions. Most of the times models show overconfidence in their predictions, and it is difficult to estimate the uncertainty in the output, even when it is critical to evaluate the

correctness of the final result [43, 62].
The paper written by Gal Y. and Ghahramani Z. [40] presents a technique for assessing the uncertainty in the model predictions. The key point is to apply Bayesian models to machine learning since the first one offers a mathematical ground that can be used to estimate uncertainty. Their approach suggests utilizing the Dropout layers to extract information about the model prediction. The dropout layer usually filters the propagation of the data from the previous layer to the following one with a specific rate, but only in the training phase. Once the model is trained and does inference, all the neurons of the dropout layers contribute to the output. The approach from Gal Y. and Ghahramani Z. uses the dropout technique in the inference step as well. The model runs the prediction several times, and for each one turns off a subset of neurons in the dropout layers. By doing so, the final output is a list of several predictions, an array of values. The standard deviation of these values can provide an estimation of the uncertainty of the result. When the standard deviation is small, it means that the predictions were similar for each subset of neurons used for the inference, and so the model was confident in the prediction for that input.

3 Approach

This chapter reports our solution to the problem introduced in Section 1.1. The main idea in our approach is to develop a way of classifying a source code as buggy with no prior-knowledge of it. This process can then be implemented in a tool which provides useful suggestions to the developer regarding possible threats and vulnerabilities in the code.
We propose to achieve this goal by training a machine learning model on several examples of known buggy and non-buggy codes represented as graphs, and affected by the same bug pattern. We then use the model to classify previously unseen codes.
The general idea can be roughly summarized in three steps:

Static Analysis. The first step uses static analysis to extract properties from each piece of source code and summarizes those properties into a simplified version of a Code Property Graph. This step is the only step language dependent and requires a parser for the language of the source code.

Training. In the second step, a graph-based classifier is trained with examples of graphs from buggy and non-buggy codes, so that it learns how to distinguish those two kind of graphs. We use two different machine learning models, a Multilayer Perceptron and a Random Forest model. Both of them require a vector as input data, so the graphs generated in the first step needs to be vectorized before training the model. The model is trained with a specific dataset that contains buggy examples of just one specific bug pattern. In detail, for each bug pattern, a different dataset is available, and then different models are trained and specialized.

Testing. The third and last step concerns the classification of previously unseen source codes and the validation of the entire approach. The models trained in step two now determine whether the unseen code suffers from the bug pattern by classifying the code as buggy together with a likelihood value and an uncertainty measure of the prediction.

The three steps are explained in Section 3.2, Section 3.3, and Section 3.4, respectively.
The next Section introduces one more auxiliary step, that revealed necessary for our purpose as we needed data to test and validate our approach. Having a big and variegated dataset is extremely important in machine learning, and our approach needs thousands of buggy examples to provide useful results.
While producing bugs-free code is not possible, we need examples that are as close as possible to that, or at least that are free from the specific bug we want to test. We need a dataset that contains buggy and non-buggy example of the same source code. The study and generation of such

a dataset is an interesting topic on its own and requires months of experiments and tests. We focus this work on static analysis and graph classification, and thus choose a simple solution that is able to produce a desirable dataset of codes in a short amount of time. We perform mutations on code from well-tested and known open source projects to generate bugs and we then collect all those codes (the original source code and the mutated one) in our dataset.

3.0.1 Notations

We introduce the notations used in this Chapter for representing graphs. In the approach, we consider the problem of learning from a population of directed and cyclic graphs $\mathcal{G} = (g_1, ..., g_G)$. A graph $g = (\mathcal{V}, \mathcal{E}, LN, LE)$ is defined by a set of nodes $\mathcal{V}$ and a set of edges $\mathcal{E}$. The set of edges $\mathcal{E} = \{(s_1, d_1), ..., (s_m, d_m)\}$ contains m direct edges (s_i, d_i), where s_i represents the source node and d_i the destination node of the edge. Every direct edge (s_i, d_i) is associated to a label a_{s_i,d_i}, which is taken from a finite set of integers $\{0, 1, ..., LE\}$, where LE is the maximum label integer. Similarly, every node v is associated to a label x_v from a finite set of integers $\{0, ..., LN\}$, where LN is the maximum label integer. We introduce the concept of incoming and outcoming edges of a node, defined as the subset of nodes such that $Inc(v) = \{u \in \mathcal{V} | (u, v) \in \mathcal{E}\}$ and $Out(v) = \{u \in \mathcal{V} | (v, u) \in \mathcal{E}\}$, respectively, and the concept of vector $X = \{x_1, ..., x_k\}$, as a one-dimensional array of scalar value x_i with dimension k.
In order to make clear some steps of the approach, we would like to clarify the meaning of some words used in this Chapter. When referring to "token", we mean the use of the word in the compiler design. When the lexical analyzer reads source code, it produces a list of tokens: such tokens are variable names, symbols, statements and language keywords that constitute the source code. For instance, the code `int num = 3 + 1;` produces seven tokens ($\{int, num, =, 3, +, 1, ;\}$). Moreover, we are taking into account two subsets of this token set, the literals, the set of numbers contained in the source code (both the integers and the floating-point numbers) and the identifiers, defined as all the strings in the source code that are not language reserved. For instance, the code `String foo = "The answer is" + "42";` would produce a set of identifiers $\{foo, "The\ answer\ is", "42"\}$, where 42 is an identifier because it was cast to a string.

3.1 Generate Bugs in Java Code

Some datasets of bugs are available online (see Chapter 8), but none of them is big enough to train a machine learning model. The closest example of a dataset that fits our requirements is Defects4J [57]: it contains 395 known and labeled bugs from several open-source projects but still not enough samples for our use case. For each bug, Defects4J provides a buggy code and a fixed program version. Nevertheless, we are collecting specific bug patterns and generating one dataset for each of them. When grouping bugs by type we are left with only a few examples on Defects4J for each of them.
Our solution uses the Major mutation framework [56] (see also Section 4.1) to mutate the code randomly with the purpose of generating bugs. We select well-tested open source projects that are well-known for the reliability of their test suite, this is a crucial point of our approach. Buggy

codes are created by modifying the code and injecting small artificial faults into the program; the faults are generated by the mutations. Examples of mutations are the replacement of arithmetic operators, the manipulation of branch conditions, or the deletion of a whole program instruction. Our proposal for generating buggy code can be described as follow:

Open Source Project Selection. We need the source code, so we have to look for open source projects available online. Furthermore, the selected open projects need to have a strong test suite, on which we can test the validity of the injected bugs.

Mutation. We run the Major mutation framework on the selected projects and collect all the mutated code. We apply arithmetic, logical, relational, conditional operator replacement, expression value replacement, literal value replacement, and statement deletion. The mutations affect both literals and identifiers of the code, and entire statements too (a list of all the applied mutations with examples is shown in Figure A.1 in the Appendix). The Major framework applies the mutation to the AST of the compiler to reduce the likelihood of compilation errors, but the compiler may still fail in some cases. The mutated codes that do not compile are discarded.

Test on the Mutated Code. The entire corpus of the mutated code is tested. The process is applied to one mutated code at a time. First of all, the original code file is replaced by the mutated one in the project. Then, the entire project is recompiled and tested using the provided test suite.

Analysis of Test Result. The test log is analyzed to find out which buggy behavior we got. Each mutated code is either discarded or classified as buggy code. In case there is no test failure, we can assume that the mutation does not affect the computation or, more likely, the test suite does not cover a specific test to recognize this modification to the code, so we discard the mutated code. On the other hand, if some tests do not pass, we group those mutated codes by the kind of error or exception raised. The selected mutated codes constitutes the buggy examples and the original source code the non-buggy examples.

The last two steps of this approach are explained in pseudo-code in the Algorithm 1, where `project_code` represents a pile of all the source codes of the selected open source project. The mutation task, represented by the function `apply_mutation(sourceCode)`, applies different kind of mutations and returns a list of mutated codes.

The procedure returns as many datasets as errors and exceptions encountered by the test suite. We analyze the list of errors and select which bug pattern we want to investigate. For instance, in our experiments we selected three different bug patterns: Null Pointers, Array Index Out of Bounds, and String Index Out of Bounds. For each of them, a dataset with original codes and mutated ones was created. The so constructed corpus of code constitutes the base for the next steps of the approach.

Algorithm 1 Mutating code and collecting bugs

```
while project_code.size() ≠ 0 do
  original_code ← project_code.pop()
  mutated_code ← apply_mutation(original_code)
  while mutated_code.size() ≠ 0 do
    code_to_test ← mutated_code.pop()
    X ← compile(code_to_test) {returns 0 if compilation fails, otherwise class file}
    if X = 0 then
      continue {discard mutated code because compilation failed}
    end if
    Y ← apply_test_suite(X) {Y contains the error raised up, if any}
    if Y.equals("Test Succeed!") then
      continue {discard mutated code because mutation does not introduce any error}
    else
      code_in_dataset(code_to_test, Y) {Put the mutated code in the dataset of the Y error}
    end if
  end while
end while
```

3.2 Static Analysis to Generate Graphs

Our approach represents source codes as graphs. The key idea is to improve the information provided by reading a source code file. For instance, the graph representation can contain additional information regarding data and control dependencies between statements. It should also be easier for humans to detect errors in the source code by reading a graph than a sequence of code lines, at least for small examples. Unfortunately, graphs are usually too complicated to be understood by humans. Our approach uses machine learning models to address this problem. We also simplify the graph representation as much as possible while preserving the information useful for our purpose. We then present the graphs to a machine learning model, hoping that the additional information provided by the graph can help the model classify the inputs efficiently.

In this Section, we present the static analysis used to extract properties of the code and represent then as a graph. We chose to represent the code as a Code Property Graph, as presented in Section 2.1.4, but we slightly changed its representation to better satisfy our needs.

We chose CPG among others graph structures because of the information that it contains. Bugs can be complex and nested in the code,and we need a considerable amount of information such as execution flow and dependencies between statement to detect them. The CPG structure offers a useful, variegated and rich representation: it contains information regarding control and data dependencies of the statements, and the AST nodes offer a specific overview of the tokens that constitute the statements. Therefore, the graph preserves information regarding the execution flow but also data about a specific token in the code. For instance, if particular literals (e.g., a typo) cause a bug, the information is preserved in the structure, and a developer may find it. Shortly, the representation appears general but also detailed, and it is able to detect mistakes and errors of different types.

Nevertheless, the original version of CPG appears redundant for our purposes, and we propose to

cut out some nodes to make the structure less general and more specialized to our requirements. In fact, the structure also preserves the "intermediate AST nodes", nodes that connect the statements with the tokens. Figure 3.1 shows an example of these nodes in the AST. Because of those nodes, few lines of code generate a graph with several nodes (see Figure A.4 in the Appendix for Listing 2.1). All those "intermediate nodes" are redundant for our purpose because the mutations only affect the tokes. They are directly dependent on the statements from which they derive; the differences between two separate instances of the same statement reside in the tokens, as different identifiers or literals values. For instance, the statements `int foo1 = 4` and `int foo2 = 2` would have exactly the same AST structures, except for the content of the tokens nodes, $\{foo1, 4\}$ and $\{foo2, 2\}$, respectively. Therefore, we propose to simplify the CPG structure to produce a lighter one that preserves the same information of the original one with regard to our purposes.
Precisely, we are interested in preserving the statement nodes and the token nodes (the ones affected by the mutations), while excluding all the others. By recalling the concept of incoming and outcoming edges, defined as the subset of nodes such that $Inc(v) = \{u \in \mathcal{V} | (u, v) \in \mathcal{E}\}$ and $Out(v) = \{u \in \mathcal{V} | (v, u) \in \mathcal{E}\}$, we can define the "intermediate nodes" as a subset of $\mathcal{V}$:

$$statement\_nodes = \{x \in \mathcal{V} | \exists (u, v) \in Inc(x).u = x \vee v = x \wedge a_{u,v} = control\_flow\_edge\_label\}$$

$$token\_nodes = \{x \in \mathcal{V} || Inc(x)| = 1 \wedge |Out(x)| = 0\}$$

and by exclusion

$$intermediate\_nodes = \{x \in \mathcal{V} | x \notin statement\_nodes \wedge x \notin token\_nodes\}$$

where $|Inc(x)|$ represents the cardinality of the $Inc(x)$ set.

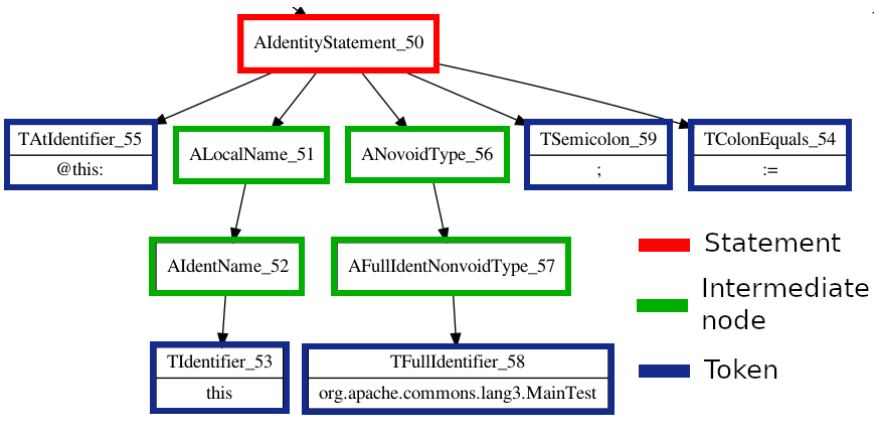
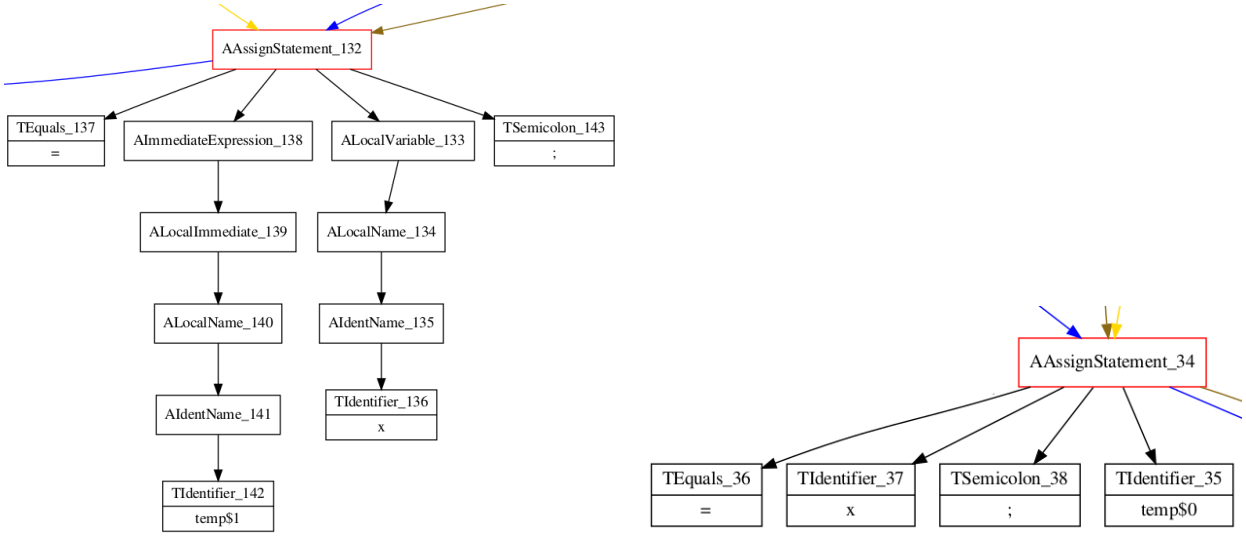


Figure 3.1: Example of intermediate AST nodes.

Our approach uses a simplified version of the CPG, where the intermediate AST nodes between the statement and the tokes are removed. Figure A.5 in the Appendix shows the simplified version of the CPG presented in Chapter 2, while Figure 3.2 shows an example of the differences between the original CPG and the simplified one; the two subfigures are extracted by the complete graphs

shown in the Appendix (see Figure A.4 and Figure A.5). The simplified CPG is lighter and smaller than the complete one but preserves all the information essential for our task. In fact, all the statements and the edges that describe dependencies are still present, and so are the token nodes containing the literals and identifiers values.



(a) Complete CPG

(b) Simplified CPG

Figure 3.2: Example of differences between Figure A.4 and Figure A.5 in the Appendix.

The simplified CPG allows the machine learning model to work with a graph which is almost half the size of the original one (e.g., the two graphs in the Appendix have 104 and 51 nodes respectively) and that still has the same information useful in detecting bugs.
For each source code in the dataset, a static analysis implemented with the Soot framework (see also Chapter 4) generates an AST, a CFG, and a PDG for every method of the class. These three graph structures are then merged and modified to produce a simplified version of the CPG. Finally, each method has a CPG that represent its source code.
The static analysis applied is intra-procedural, it applies to single methods and only uses the information available in the code lines of a method. In contrast with the inter-procedural analyses, the intra-procedural analyses do not consider relationships between different methods and functions and cannot represent information coming from other methods of the project.
The Algorithm 2 explains in pseudo-code the steps to produce a CPG for each method of an input source code.

Each method is extracted from the input source code file before static analysis is performed on it: the AST, CFG, and PDG are created and later merged to generate a CPG. The CPG is then simplified by removing the intermediate AST nodes.

3.3 Graph Vectorization

To apply machine learning models to our datasets, we need to represent the graphs in a suitable form. Our approach converts the graphs into vectors of fixed size, which can be taken as input by a

Algorithm 2 Generate CPGs for each method of a source code "MyClass"

```
list_of_methods ← extract_methods(MyClass) {split source code into a list made by its methods}
while list_of_methods.size ≠ 0 do
    X ← list_of_methods.pop()
    ast ← generate_ast(X)
    cfg ← generate_cfg(X)
    pdg ← generate_pdg(X)
    cpg ← generate_cpg(ast, cfg, pdg)
    simplify_cpg(cpg)
    list_of_cpgs.put(cpg)
end while
```

machine learning model that classifies them. We adopt the CGMM approach (see Section 4.3), so the approach uses an unsupervised machine learning model to convert the data into fixed dimension vectors.

To vectorize a graph, we need to represent it in a standard format: we label the nodes and the edges and represent the entire structure in the format required by the machine learning model that performs the vectorization. There are only four different types of edges in our graph representation: AST edges, control dependency edges, data dependency edges, and control flow edges. Recalling our notation introduced in Section 3.0.1, this means that $LE = 3$ and the set of possible edge labels become $\{0, 1, 2, 3\}$. On the contrary, labeling the nodes is not easy: although most of them are unique and can be mapped to a specific label (e.g., the statements), token nodes need to be managed carefully. Since the bug may occur in a typo (e.g., a variable set to 1 instead of 2), the graph has to preserve the difference between two nodes both containing, for instance, an integer number but with different values. Consequently, we cannot simply enumerate all the possible node labels because we should label all the possible literals and identifiers with a different number, which means enumerating two infinite sets. A naive approach would be to enumerate and label all the literals and identifiers of the project. Still, we would have to label thousands of variables.

Our approach uses a "Top N variables" strategy to list the most important literals and identifiers in the project and labels each of them with a specific number. This approach reduces the number of labels while still keeping relevant information. In particular, we collect all the identifiers and literals and count the number of occurrences for each of them. We then sort this frequency list and select the first X literals and identifiers which cover the $N\%$ of the entire variables occurrences. To each of this X variables, we assign different label. All the other variables, which constitute the $(100 - N)\%$ of the occurrences, are represented by just one label, despite their value. The key idea is to give priority in the labeling operation to those values that occur many times in the dataset: we assume they are more important in the project. For instance, after the sorting operation, we can find out that the literal 1 occurs two hundred times in the project while the integer 424242 only appears two times, so the number 1 deserves to have a unique label more than 424242, because it identifies a widely used variable. The $(100 - N)\%$ of the entire variables occurrences represents identifiers and literals that occur just a few times in the project.

Formally, the labeling problem is addressed by a map function $map(x_v) = label$ that maps each node

label to an integer number. The first 114 numbers are reserved for statements and language reserved words and symbols (e.g., colons, parentheses, operators). When the input dataset that contains all the graphs is processed all the identifiers and literals are counted and sorted by frequency. Finally, the top variables which cover the $N\%$ of the entire variables occurrences are selected and the map function is ready to label every node of the dataset. Algorithm 3 explains the counting and sorting operation of the top N variables when defining the map function in pseudo-code. The function `count_label.increment(identifOrLiteral)` of the dictionary structure $count\_label = \{(X_i, Y_i)\}$ increments Y_k by one where $(X_k, Y_k) \in count\_label \wedge (X_k = identifier \vee X_k = literal)$.

Algorithm 3 Counting and selecting the Top N variables in the input dataset, where N = 90

```
count_label.initialize() {Dictionary (X,Y) where X is a String/Identifiers and Y an integer}
total_ideAndLit ← 0
label ← 114 {starting label, number smaller than 114 are reserved}
while dataset.size ≠ 0 do
  graph ← dataset.pop()
  visit_graph ← graph.iterator()
  while visit_graph.hasNext() do
    node ← visit_graph.next()
    if node.isIdentifiersOrLiterals() then
      if count_label.contains(node.getContent()) then
        count_label.increment(node.getContent())
      else
        count_label.add(node.getContent(), 1)
      end if
    end if
    total_ideAndLit ← total_ideAndLit + 1
  end while
end while
count_label.sortByValue() {Sort the dictionary by the value (the integer)}
total_ideAndLit ← total_ideAndLit * 0.9 {90% of the total ide and lit number}
while total_ideAndLit ≥ 0 do
  temp_ideLit ← count_label.getFirstAndRemove()
  map_labels.add(temp_ideLit.getKey(), label)
  total_ideAndLit ← total_ideAndLit − temp_ideLit.getValue()
  label ← label + 1
end while
while count_label.size ≠ 0 do
  temp_ideLit ← count_label.getFirstAndRemove()
  map_labels.add(temp_ideLit.getKey(), label)
end while
return label
```

The Top N variable algorithm has a complexity in time of $O(G * d + l)$ where G is the number of graphs, l the number of literals and identifiers and assuming that each graph has the same number of nodes d.

The graphs are written on file as adjacent lists and then taken as input by the CGMM tool. First of all, a model is trained for each dataset of graphs. Therefore, we generate one model for each

bug pattern. The trained models are then used to convert graphs to vector. Each model outputs vectors $X = \{x_1, ..., x_k\}$, one vector per graph, each of size k. The dimension k is regulated by the number of layers of the CGMM model and it is fixed for each vector of the same graph dataset.

3.4 Machine Learning Classifiers

The last step of the approach regards the machine learning model that performs the classification task. As explained in the previous section, the CGMM tool converts the graphs into vectors that can be used by a machine learning model for training and validation tasks. Our approach uses a MLP.
First of all, a MLP model is trained using the entire dataset to generate a trained model, one model for each bug pattern dataset. Then, the trained models are stored in memory and later used to classify previously unseen graphs.
In conclusion, the approach generates several trained models, one for each bug pattern considered, and then applies the model on new projects and previously unseen vectors to classify them and give advice and suggestions to the developers regarding possible threats and vulnerabilities. The following subsection reports the steps for the classification of previously unseen vectors.

3.4.1 Classification

Taking a set of graphs as input, each represented by a vector, our approach classifies each graph with a value in the range $[0, 1]$, where one means buggy and zero means non-buggy. We then adopt the approach of Gal Y. and Ghahramani Z. explained in Section 2.3.2.1 to evaluate the uncertainty of the model for each prediction. The model runs R more times, using the dropout technique for the inference phase, so that the output is a vector of predictions of dimension R. This vector contains predictions of the model for the input graph, one prediction for each of its R different neurons configuration due to the dropout.
We define the uncertainty value for a specific vector *vect* as:

$$uncertainty(vect) = |pred(vect) - avg\_pred\_dropout(vect)| + std\_pred\_dropout(vect)$$

where $pred(vect)$ represents the standard prediction of the model, $avg\_pred\_dropout(vect)$ represents the average of the R predictions of the model with the dropout and $std\_pred\_dropout(vect)$ the standard deviation of the R predictions of the model with the dropout.
Then, the approach narrows down the set of input graphs and removes the ones where the uncertainty of the model is bigger than a threshold value T. The threshold function for removing the vectors is defined as:

$$filter\_vectors(vect) = \begin{cases} remove & if\ uncertainty(vect) > T \\ store & otherwise \end{cases}$$

where T is the uncertainty precision or threshold.

Intuitively, the function *uncertainty*(*vect*) outputs a small value if the difference between the prediction without the dropout and the average of the dropout predictions is small, and also if the standard deviation of the predictions with the dropout is small. In this event, the model with the dropout and the one without the dropout output similar values, and moreover all the R predictions are similar as stated by the small standard deviation, which means that the model is confident on that classification.

Finally, the approach outputs a subset of the input graphs. This subset contains the graphs on which the model has an uncertainty value smaller than the threshold T. For each graph, the prediction value generated by the model without the dropout is provided. By doing so, the model suggests to the developer which graphs (methods) need to be checked carefully for avoiding vulnerabilities (the ones that are classified close to 1), and which graphs look more secure for that bug pattern (the ones classified close to 0)

Algorithm 4 shows the steps we just explained in pseudo-code, where *model* represents one of the available trained models, specialized in recognizing one specific bug pattern. The algorithm presents the computation for one single vector but it can be easily generalized for a dataset of vectors.

Algorithm 4 Classifying vectors and calculating uncertainty values

$pred \leftarrow model.predict(vector)$
$array\_drop\_preds \leftarrow model.predict\_drop(vector, rate)$ {rate is the number of dropout runs}
$avg\_drop\_pred \leftarrow average(array\_drop\_preds)$
$std\_drop\_pred \leftarrow dts(array\_drop\_preds)$
$uncertainty \leftarrow calc\_uncer(pred, array\_drop\_preds, std\_drop\_pred)$
if $uncertainty < T$ **then**
  return $pred$
end if

4 Implementation

Our tool GrapPa implements the approach presented in the previous chapter. The following Sections present the GrapPa 's components and introduce the frameworks and external tools on which the implementation is based. The tool is available online on Github [53].
The implementation of our approach can be split into four modules. Each component uses a different framework or external tool, is built on the top of the previous one and provides input for the next one. Figure 4.1 shows an overview of these components and briefly summarizes their operations, as presented in Chapter 3.
It is worth noting that the tool GrapPa covers only the last three modules since the first one (the code mutation) was a necessary step to generate the dataset for training the models, but the tool reuses the trained models and does not need to recover the datasets and train the models again.

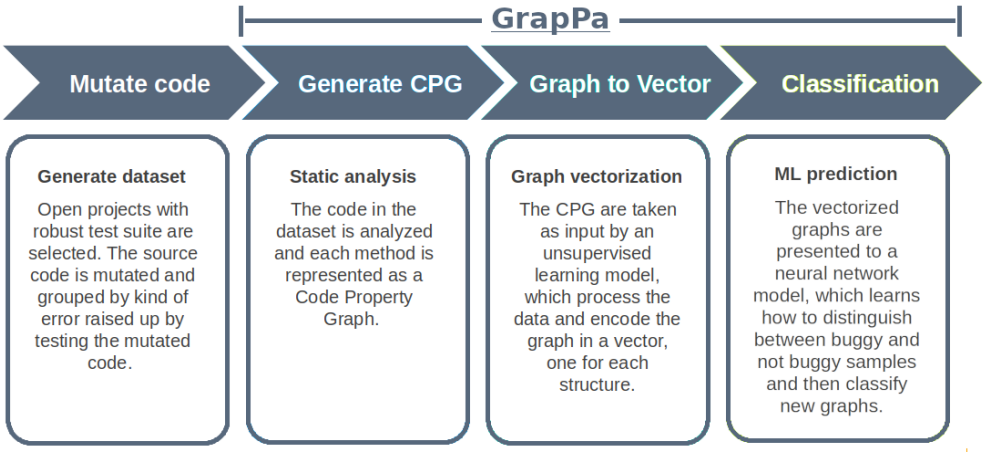


Figure 4.1: Overview of the implementation components.

The first component is implemented using the Major mutation framework and is introduced in Section 4.1. The second one makes use of the Soot framework, introduced in Section 4.2, and provides the input for the third component, which uses the tool CGMM, introduced in Section 4.3. Finally, the last module that performs the classification task relies on two different machine learning model, implemented using the WEKA and Keras frameworks (see Section 4.4 and Section 4.5).

4.0.1 Open Project Selection

GrapPa uses trained models to analyze and classify inputs. These models were trained on datasets of data extracted from open source projects. This step constitutes the first one of the tool imple-

mentation, but it is not included in the its features since the trained models are stored in the tool and reuse for future predictions and classification tasks. Nevertheless, the procedure can be applied to any other dataset and so extend the tool with more trained models. The model selection and code mutation procedures are reported in Chapter 5 for clarity and replicability.

4.1 MAJOR

Major mutation framework is an open source project that enables efficient mutation analysis of software systems written in Java. It is available online [12] and was introduced by J. Ren [56].
Major main features are a modified version of a Java compiler based on Java7, which takes source codes as input, mutates and then compiles them, and a modified version of JUnit test, to analyze the mutations on the test suites of the input project and collect the results. Moreover, it has its domain-specific language to configure the mutation process in detail.
We applied all the mutations offered by the framework on the open source projects selected. Figure A.1 in the Appendix reports the possible mutations of Major mutation framework.

4.2 SOOT

Soot is a framework developed by the Sable Research Group at McGill University [77]. It supports the analysis and transformation of Java bytecode for optimization tasks. Soot is free software, written in Java, and available online [16].
Soot provides four different intermediate representations for representing Java bytecode. Among these, our tool uses Jimple, a typed 3-address intermediate representation that forms a simplified version of Java source code. The framework also contains detailed program analyses, such as call graph analysis, on which we develop our graph analysis.
To apply Soot graph analyses, Java bytecode is converted to Jimple, and then the framework has a strict class hierarchy to handle and analyze these codes. The fundamental Soot object is the `Body`, which stores the code for a method. Namely, the `Body` object for the Jimple intermediate representation is called `JimpleBody`. The `Body` contains three "chains", list-like data structure. Each chain contains information regarding statements of the method stored in the `Body`. The most interesting part of a `Body` is its chain of `Unit`s, which is the actual code contained in the `Body`, one `Unit` per statement, linked and sorted as in the original code.
One more important concepts of Soot regards packs and phases. The Soot phases regulate the code transformation and analyses. First, Soot applies the `jb` pack to every single method body, which converts the Java bytecode to the Jimple representation. Then, Soot applies four whole-programs packs, which perform transformation and optimization on the Jimple code, and in particular one can add a `SceneTransformer` object to these packs and introduce a new analysis. Our tool implements a `SceneTransformer` in the whole-jimple transformation pack (`wjtp`) to perform analyses to generate ASTs, CFGs, PDGs and finally the CPGs.
The CPG in the tool GrapPa is implemented through the `CodePropertyGraph` class, that contains

Listing 4.1: Attributes of the CPGNode class

```
public class CPGNode
{
    public enum NodeTypes {//fixed set of elements for variable Type
        AST_NODE,
        EXTRA_NODE,
        CFG_NODE
    }

    private NodeTypes nodeType;
    private String name;
    private int nodeId;
    private String content;
    private Set<CPGEdge> edgesOut;
    private Set<CPGEdge> edgesIn;

}
```

two data structures to store the elements of the CPG: the nodes, implemented in the `CPGNode` class, and the edges, implemented in the `CPGEdge` class. The attributes of these two classes are reported in Listing 4.1 and 4.2.

By using the option `-process-dir <project_path>`, the tool GrapPa takes the project path as input and performs the following operations to represent every method as a CPG:

STEP 1 - Convert Java to Jimple. The Soot analysis takes the project files as input, and the `jb` pack parses and stores each method encountered in a `Body b` object. Then, our `SceneTransformer` gets the `Body b` objects and applies analyses on them.

STEP 2 - Generate the AST. We implement a class called `NedoAnalysisAST` that extends the `DepthFirstAdapter` class in the `soot.jimple.parser.analysis` package.
`NedoAnalysisAST` parses the Jimple code contained in the `Body b` object and generate an AST. Then, a `CodePropertyGraph cpg` object is initialized and the AST is visited. For each AST node, a new `CPGNode` class is instantiated, the AST node information is copied in the `CPGNode` attributes and then the CPG node is added to the `CodePropertyGraph cpg`. In most of the cases, these `CPGNode`s have node type `AST_NODE`, but the analysis also recognizes the statement nodes and initializes them as `CFG_NODE`s (see Figure 2.1, where the red node represents the `CFG_NODE` in the AST struct). This step is crucial, because the `CFG_NODE`s are mapped to the CFG and PDG statements node in the next steps, and mark them makes easy to combine the three different graph representations into the CPG.

STEP 3 - Generate the CFG. The Soot framework provides an implementation of a CFG trough the `ExceptionalUnitGraph(Body b)` class, which takes the `Body b` created in STEP 1 as input and returns a `UnitGraph cfg` object. The CFG is visited and each node is mapped to a `CFG_NODE` in the `CodePropertyGraph cpg`. For each CFG edge from `UnitBox X` to

Listing 4.2: Attributes of the CPGEdge class

```
public class CPGEdge
{
    public enum EdgeTypes {//fixed set of elements for variable Type
        AST_EDGE,
        CFG_EDGE_C,
        PDG_EDGE_C,
        PDG_EDGE_D
    }

    private EdgeTypes edgeType;
    private CPGNode source;
    private CPGNode dest;

}
```

`UnitBox Y` (where `UnitBox` is a node in the `UnitGraph` class), a new `CPGEdge` is added to the `CodePropertyGraph cpg` object, where the source and destination nodes represent the mapped `CPGNode`s for the `UnitBox X` and `UnitBox Y`, respectively.

STEP 4 - Generate the PDG. In the same way of STEP 2, the Soot framework provides an implementation of a PDG trough the `HashMutablePDG(UnitGraph u)` class, in the Ferrante version [39].
It returns a `ProgramDependenceGraph pdg` object by taking as input the `UnitGraph cfg` created in STEP 2. Then, the PDG edges are combined into the CPG struct in the same way of the CFG edges presented in STEP 2. The `ProgramDependenceGraph pdg` graph is traversed and each visited edge is added to the `CodePropertyGraph cpg` as a `CPGEdge` with type `PDG_EDGE_C`. Unfortunatly, the PDG implementations available in Soot does not provide information regarding data dependency. In order to add also the data dependency edges in our graph, GrapPa uses another analysis provided by Soot, which detects all the DU-pairs in a `UnitGraph` object. A DU-pair is a couple of statement (A, B), where A corresponds to a declaration and B to a statement that uses the variable declared in A. Thus, the DU-pairs correspond to the definition of data dependency. The `SimpleLiveLocals(UnitGraph u)` class takes the `UnitGraph cfg` generated at STEP 2 as input and returns all the DU-pairs of the `UnitGraph` object. For each DU-pairs, a new `CPDEdge` of type `PDG_EDGE_D` is created and added to the CPG.

STEP 5 - Simplify the CPG. So far, the CPG created corresponds to the version presented in the Yamaguchi F. paper [86]. GrapPa implements one more step and simplifies the graph, as explained in Section 3.2. The `CodePropertyGraph pdg` object is visited and all the "intermediate AST nodes" are removed. We formally define these nodes as the set of `CPGNode`s of type `AST_NODE` that have at least one outcoming edge (they are not leaves). These nodes are removed and the leaves (the tokens) are linked to the closest `CPGNode` with an `AST_EDGE`.

Listing 4.3: adjlist file example where the incoming edges are tuple (SourceId EdgeLabel)

```
#nodeId nodeLabel adjacencyList_incomingEdges
0 0
1 0 (142 1) (169 1) (121 1)
2 7 (0 1)
3 85 (2 0)
4 554 (2 0)
5 9 (2 3) (192 1)
.........
```

The so created CFG can be used for further analyses and transformations. Furthermore, the `CodePropertyGraph` class provides a method (`getCPGtoString()`) that stores the graph in memory as a text file. Thus, the graph can be loaded again by passing the text file to the `CodePropertyGraph` constructor.

4.3 CGMM

GrapPa incorporates and extends the tool CGMM [7] for vectorizing the CPGs. The `CPG2CGMM` class takes a `CodePropertyGraph` object as input and returns a text file in the ".adjlist" CGMM format that can be passed to CGMM component and vectorized. In detail, the CGMM component requires the graphs to be described as an adjacent list of incoming edges. The ".adjlist" file contains one node per line, and each line contains the node id, the node label, and the adjacency list of incoming edges formed by tuples (X, Y) where X is the source node id and Y the label edge. Listing 4.3 shows an ".adjlist" file extract.

By taking a set of graphs as input, the CGMM tool trains a model and produces the unsupervised encoder which can be used to vectorize more graphs. GrapPa contains three CGMM models trained on our datasets, and more models can be easily added. The CGMM component of GrapPa performs two main operations: the model training, and the graph vectorization.

Training. The file `MAssessment.py` takes as input a set of graphs labeled either as buggy or not buggy, and returns a trained model, that can be used to vectorize previously unseen graphs.

Vectorization. The file `MVectorization.py` takes a trained model and a dataset of ".adjlist" files as input and returns a text file that contains a matrix, a list of vectors, one vector for each ".adjlist" file in the dataset.

Further information is reported in the GrapPa documentation.

4.4 Weka

Weka is a collection of machine learning algorithms for data mining tasks [84]. It contains tools for data preparation, classification, regression, clustering, association rules mining, and visualization. Weka is developed and supported by the Machine Learning Group at the University of Waikato

Listing 4.4: Python code to create the MLP used by GrapPa

```
1 model = Sequential()
2 model.add(Dense(512, input_dim=data.shape[1], activation='relu'))
3 model.add(Dropout(0.5))
4 model.add(Dense(512, activation='relu'))
5 model.add(Dropout(0.5))
6 model.add(Dense(1, activation='sigmoid'))
```

and is free software available online [19].
Weka provides a complete graphical user interface and allows to test datasets on different machine learning models and approaches. By importing the datasets, in the weka format ".arff", the user can select different models and settings, do experiments on the datasets and compare the results. GrapPa does not integrate Weka but provides scripts to convert a dataset of vectors to an ".arff" file, which can be uploaded in Weka to perform experiments. Besides, a runnable version of Weka is stored as a ".jar" file in the project source folder.
We use Weka and several models to find the best hyper-parameters for converting our datasets of graphs to datasets of vectors using CGMM. We achieved the best result with the Random Forest algorithm; the results are reported in Chapter 5.
Weka can also be run from a terminal. The `weka.jar` library contained in the GrapPa project folder can be used for testing Weka and replicate our experiments. The following command shows our configurations for running a 10-fold cross-validation on a dataset converted into a ".arff"' file for the Random Forest algorithm.

```
1  java -cp "path/to/weka.jar" weka.classifiers.trees.RandomForest \
2    -P 100 -I 100 -num-slots 1 -K 0 -M 1.0 -V 0.001 -S 1 \
3    -t <path/to/dataset.arff>
```

4.5 Keras

Keras is an open source neural network library written in Python [11, 82]. It contains numerous implementations of commonly used neural networks that can be easily modified. Keras is conceived as an interface rather than a complete framework and runs on the top of other open-source software and libraries (e.g., TensorFlow [18]). GrapPa integrates Keras and uses its API to create, train and test neural network models. In detail, GrapPa uses Keras as an interface on the top of the machine learning framework TensorFlow.
The neural network model used by GrapPa is a MLP, with three hidden layers, 512 neurons per layer and one neuron in the output layer. Listing 4.4 shows the python code to create the model, while Figure 4.2 draws the structure of the MLP implemented in GrapPa .

The model is created as a `Sequential` model, so a linear stack of layers. At line 2 of Listing 4.4 the input layer is added to the model, identified as L_0 in Figure 4.2. The input layer is a dense layer, where each neuron is connected to every other neuron in the next layer. The input layer

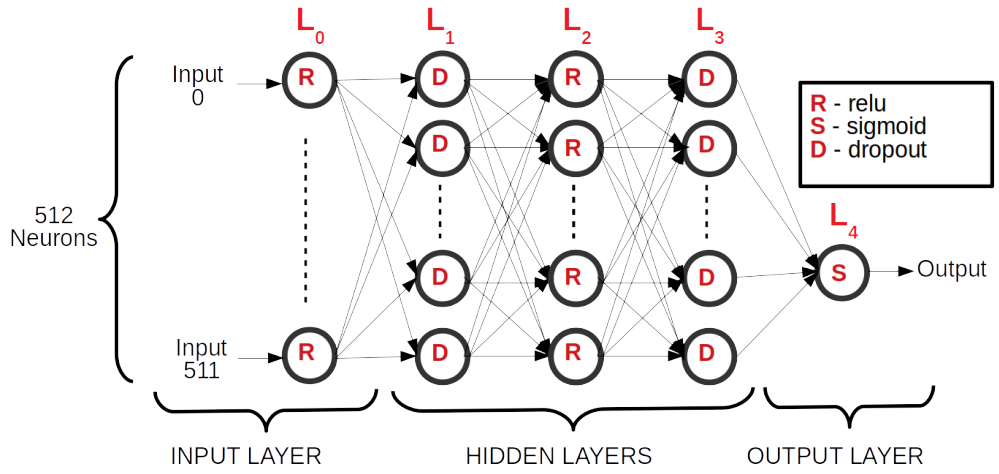


Figure 4.2: Structure of the MLP implemented in GrapPa .

needs to know what input shape it should expect; this information is passed by the argument `input_dim=data.shape[1]`, where the variable `data` contains the loaded dataset. The following layers do not need this information because they do automatic shape inference. The argument `activation='relu'` set the activation function for the input layer neurons to a Rectified Linear Unit. Line 3 adds the first hidden layer L_1, a dropout layer. The `rate` of input units is set to the default value of 0.5, so half of the input neurons randomly selected, output 0 in the training phase, while the other half propagate the input value to the next layer. The second hidden layer L_2, at line 4 of Listing 4.4, is again a dense layer with 512 neurons and a rectified linear unit as the activation function. Then, the third hidden layer L_3 is another dropout layer with `rate=0.5`. The last layer L_4, the output layer, has just one neuron.
Our goal is to classify an input vector (a graph) as either buggy or not buggy, so we are performing a binary classification. Thus, the model output has one dimension, which corresponds to the output of the neuron in the last layer. The activation function of the last layer is a sigmoid function, and so the model outputs values in the range $[0, 1]$, where one means buggy and zero means not-buggy. The neural network in GrapPa is used for two main tasks:

Training. The Python script `MLP_MAssessment.py` takes a dataset as input, generates a MLP and trains the model with the dataset provided. Finally, it outputs a trained MLP model. The so trained model can be later used for prediction on new datasets.

Prediction. The Python script `MLP_LoadAndPredict.py` takes a trained model and a vector as inputs, and performs predictions on the vector using the provided model. The script can also take a dataset of vectors as input and provides predictions for each vector of the dataset. The script `selectMethods.sh` takes the predictions provided by `MLP_LoadAndPredict.py` and applies the uncertainty threshold to select only the vectors where the model is confident in its prediction. Finally, for each input vector the model outputs two values: a prediction value in range $[0, 1]$, which constitutes the likelihood of the vector to be affected by the bug pattern on which the model is trained, and an uncertainty value in range $[0, 1.8)$, which reports the confidence of the model on the prediction.

5 Evaluation

5.1 Collecting Data

We selected open projects available online to create the datasets needed for our experiments. We chose projects by evaluating their test suites. We needed projects as diverse as possible in terms of functionalities and features, and with a robust test suite that allows us to validate our mutations without having prior-knowledge on the code or having to do manual checks.
We chose to use the Apache Common libraries for generating the datasets because their test suites are generally recognized as stable and robust. Furthermore, they contain several libraries, and they constitute a good range of different operations that our model can learn.
We selected the Apache Commons Lang [4], Codec [1], Email [2], Math [5], and IO [3] libraries. Since the Major mutation framework only provides a Java7 compiler, we were not able to compile some of the most updated apache common libraries, in those cases we had to use previous compatible versions.
Table 5.1 reports information regarding the selected open source projects.

Name of the library	Version used	Number of classes
Apache commons lang	3.4	282
Apache commons math	3.6.1	1618
Apache commons IO	2.6	233
Apache commons codec	1.11	124
Apache commons email	1.5	49

Table 5.1: Overview of the open source projects used.

Each library was mutated and analyzed to collect buggy code examples. The steps can be summarized as follows, where we use the Apache Common Lang library as an example:

Mutation. The Apache Common Lang 3.6 library was compiled using the Apache Maven software project management [6]. We edited the maven "pom.xml" configuration file to set the Major compiler as the default. The project was then compiled with the mutation option enabled, and the mutated codes were collected. In Major domain-specific language, this operation is achieved by setting the variables `mvn compile "-DmutEn=true" "-DmutType=ALL"`. We generated more than thirty thousand mutated codes just from the apache commons lang library.

Testing. We tested the mutated code using the test suite provided by the apache commons lang

library. We collected the outputs and analyzed them, and we then grouped the mutated codes by error types.

Create the dataset. By looking at the error frequency table, we chose the error types that we were interested in analyzing. Among these errors, we picked up three of them: `Null Pointer`, `ArrayIndexOutOfBounds` and `StringIndexOutOfBounds`; those were the three most common errors we encountered. Finally, three datasets were created from the mutated code that raised those errors, one per error.

Details on the three datasets are shown in Table 5.2. Inside every dataset, for each mutated code file, the original code file is also provided. The mutated codes we selected and included in the datasets represent buggy code examples, while original codes represent non-buggy code examples.

Dataset	**Number of buggy methods**	**Total size of the dataset**
Null Pointer	4848	9696
Array Index Out Of Bounds	3258	6516
String Index Out Of Bounds	2517	5034

Table 5.2: Contents of the three datasets.

5.2 Building the CPGs

We applied our static analysis to each dataset and generated the graphs. The intra-procedural static analysis extracts the methods of the Java classes from each buggy and non-buggy code, and then creates a CPG for each method.
As for labeling the nodes, we adopt the "Top N variables" approach explained in Chapter 3, with $N = 90\%$. Table 5.3 shows the number X of variables labeled for each of the three datasets. This approach is efficient because allows a 90% coverage of all the variables by labeling less than the 20% of the project variables.

Bug Pattern	**90% Coverage, First X Variables**		**Total Variables**		
	literals	identifiers	literals	identifiers	total
Null Pointer	66	347	664	2667	3331
Array Out Of Bounds	64	440	574	2205	2779
String Out Of Bounds	75	386	656	2025	2681

Table 5.3: Overview of the labels for the three bug patterns considered.

Table 5.3 shows the label statistics for the three considered bug patterns. For instance, for `Null Pointer` bug, the dataset has 3331 different variables (split into 664 literals and 2667 identifiers), but just 413 labels (66 for literals and 347 for identifiers) are sufficient to cover 90% of occurrences in the project; most of the variables appear several times in different methods and classes (e.g., the integer number 1 or 2 is likely to occur more times in the project code than unusual number like 4242). For each of this top N variables, we map the identifier or the literal to one different label, all the other variables are represented by a single label, despite their value. The "excluded variables" only represent 10% of the variables occurrences and labeling them would

increase the number of labels by a factor of five because they are sparse. Once the map function for labeling was created, we started generating the graphs and the datasets.
Taking into account the `Null Pointer` dataset, the average size of methods is 22 code lines and the average graph size is around 600 graph elements (nodes and edges). The generated graphs have sizes proportional to the method sizes, as shown by Figure 5.1.

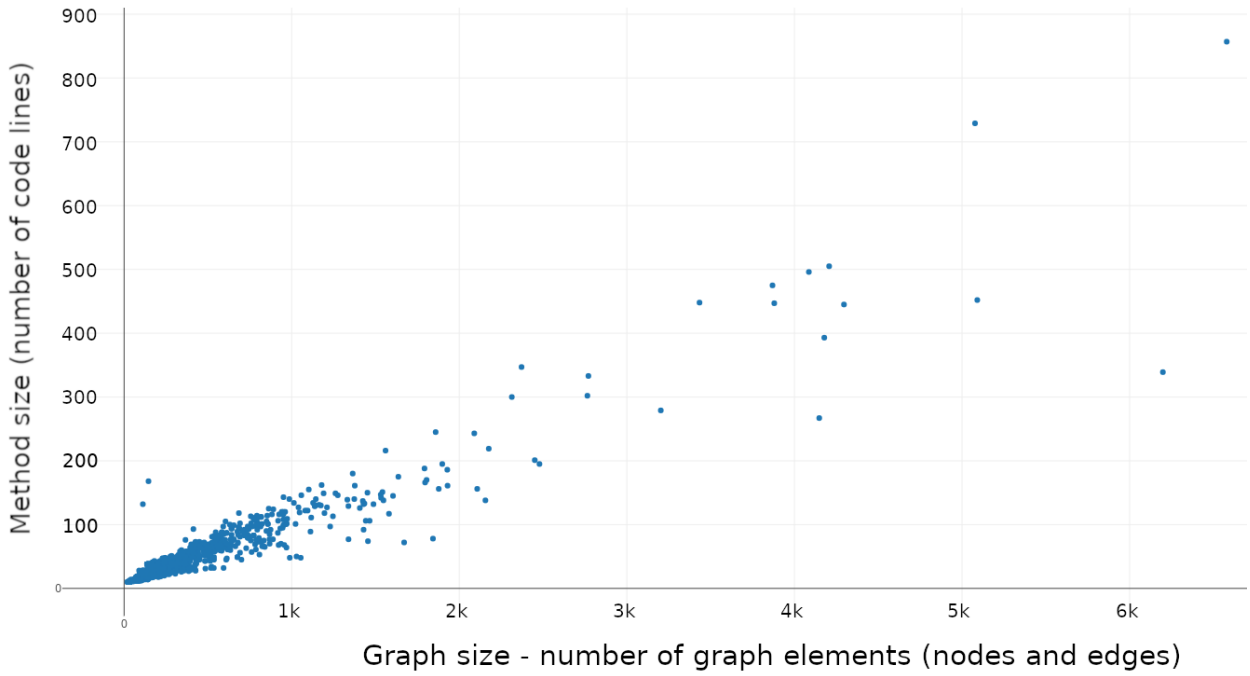


Figure 5.1: Size of methods compare to the size of the generated graphs, created with [13].

We created three different datasets of graphs, one for each bug pattern considered. In the following sections, we report the experiments for the `Null Pointer` dataset.

5.3 Training and Validation

We used Weka to find the best settings for the CGMM model.
The `Null Pointer` dataset was tested on different approaches available in the framework, with 10-cross-fold validations. Furthermore, we perform a model assessment to find the best hyper-parameters for the CGMM encoder. In detail, we tested different sizes for the vectors generated by the CGMM approach. The vector size is controlled by the number of layers of the CGMM encoder and the variable C, which represents the number of possible latent states in CGMM. Table 5.4 shows the validation result of the `Null Pointer` dataset in the Weka framework; it shows the accuracy of several models on the vectorized dataset with different layers and C values. The size of the dataset vector is calculated as $size\_vector = C * Layers$.
We used the ZeroR algorithm as a baseline, and Listing 5.1 reports the Weka settings for each of the applied models.

Layers	C	ZeroR	DecisionTable	IBk	J48	RandomForest
2	20	50.52(0.03)	75.74(1.56)	79.91(1.09)	80.92(1.07)	84.42(0.76)
4	20	50.52(0.03)	74.90(1.74)	80.24(1.18)	81.94(1.40)	85.18(1.13)
6	20	50.52(0.03)	74.17(1.75)	81.93(1.33)	82.76(1.40)	86.10(1.20)
8	20	50.52(0.03)	74.68(1.67)	81.19(0.81)	82.56(1.25)	85.68(0.71)
10	20	50.52(0.03)	76.23(1.51)	82.31(1.33)	83.15(1.28)	86.31(1.33)
2	40	50.52(0.03)	76.03(1.64)	79.27(1.29)	82.21(1.05)	84.88(1.00)
4	40	50.52(0.03)	76.20(1.47)	80.53(1.36)	83.05(1.41)	86.10(1.15)
6	40	50.52(0.03)	75.91(1.62)	81.04(1.53)	83.00(1.29)	85.78(0.95)
8	40	50.52(0.03)	75.71(1.62)	82.04(1.06)	83.41(1.32)	86.23(1.16)

Table 5.4: Result of CGMM model selection, taking into account several hyper-parameters and the models' accuracy in validation.

Listing 5.1: Settings used for models applied in Weka

```
(1) rules.ZeroR ''
(2) rules.DecisionTable '-X 1 -S "BestFirst␣-D␣1␣-N␣5"'
(3) lazy.IBk '-K 2 -W 0 -A "weka.core.neighboursearch.LinearNNSearch" /
    -A "weka.core.EuclideanDistance␣-R␣first-last"'
(4) trees.J48 '-C 0.25 -M 2'
(5) trees.RandomForest '-P 100 -I 100 -num-slots 1 /
    -K 0 -M 1.0 -V 0.001 -S 1'
```

The Random Forest algorithm shown the best result in accuracy. We selected the dataset with $C = 40$ and 8 layers, so with a vector dimension of 320. Table 5.5 shows the results of the experiment performed on this dataset: on the left we report the confusion matrix for the Random Forest model and on the right the precision, recall, and F1 scores.

Confusion Matrix		Predicted	
		Negative	**Positive**
Actual	**Negative**	421	69
	Positive	65	415

Accuracy	86.23
Precision	0.8574
Recall	0.8646
F1	0.8610

Table 5.5: Confusion matrix and results for the Random Forest model and dataset with 8 layers and $C = 40$.

The selected dataset with $C = 40$ and 8 layers was used to perform a model assessment for the MLP. By applying a 10-cross-fold validation with $batch\_size = 128$ and $epochs = 50$ we tested different hyper-parameters and configurations of layers to find the best settings. Table 5.6 shows the results, where the Relu and Dropout layer columns report the number of Relu layers and Dropout layers in the model, respectively.

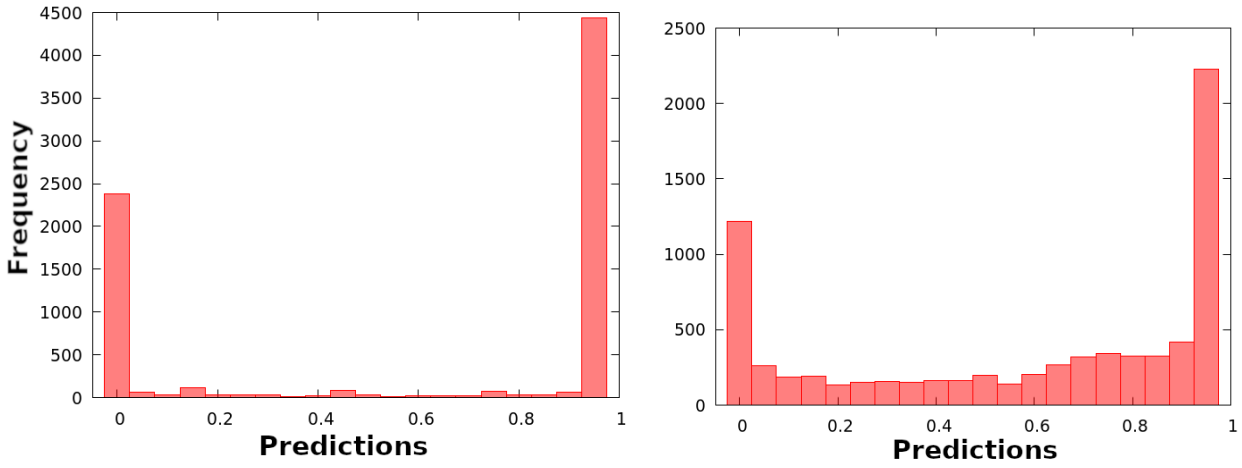
We selected the MLP with 512 neurons per layer, 2 Relu layers, and 2 Dropout layers, the same MLP configuration shown in Figure 4.2 in the previous Chapter. We trained the MLP model using the entire training set and we stored the model so that we could use it later to classify previously unseen samples.

Neurons per layer	Relu layers	Dropout layers	LOSS	ACC
64	2	2	0.2693	0.8798
256	2	2	0.2389	0.8989
512	2	2	0.2378	0.9012
1024	2	2	0.2370	0.8843
512	3	3	0.3353	0.8850

Table 5.6: Result of 10-cross-fold validation for MLP with different hyper-parameters.

5.4 Test

The MLP models stored in the GrapPa tool were used to test the approach on a test dataset, a set of graphs that were not included in the training dataset. This section reports the test results. The charts and histograms of this Chapter are created using Gnuplot [9].
We selected the JFreeChart [10] library as the test dataset. JFreeChart is a Java chart library which contains 1019 Java files and 7555 Java methods. We used the trained `Null Pointer` model and applied the tool GrapPa on the entire JFreeChart project.
For each of the 7555 methods, the model returned a value in the range $[0, 1]$, which represents the likelihood of that method raising a Null Pointer Exception. We run then the model 30 more times applying the dropout technique (each run had a subset of neurons turned off); we ended up with lists of 30 values in the range $[0, 1]$, one list for each method. Consequently, we computed the average of the 30 predictions and finally, we got two values for each method: the prediction without the dropout and the average of the predictions with the dropout. Figure 5.2 shows the predictions made by the model, the subfigure on the left reports the predictions of the model without the dropout, while the subfigure on the right reports the average of the predictions of the model with dropout.
The model without the dropout classified most of the vectors either as zero or one, while the model with the dropout showed more variety in the predictions, even though most of the predictions were still one or zero.



(a) Predictions without dropout

(b) Average predictions with dropout

Figure 5.2: Comparison between the two prediction approaches.

We calculated the uncertainty values for each prediction using the formula reported in Chapter 3. Figure 5.3 reports the average uncertainty values grouped by ranges of prediction. We can see that the model is confident when the prediction value is one or zero, where the uncertainty is small, while in the range $[0.2, 0.8]$ we have higher uncertainty values. Table 5.7 reports the same values of Figure 5.3.

range of prediction	**Average uncertainty**
$P = 0$	0.0001
$0 \leq P < 0.2$	0.1887
$0.2 \leq P < 0.4$	0.4173
$0.4 \leq P < 0.6$	0.4556
$0.6 \leq P < 0.8$	0.4110
$0.8 \leq P < 1$	0.1980
$P = 1$	0.0001

Table 5.7: Average of the Uncertainty values grouped by range of predictions.

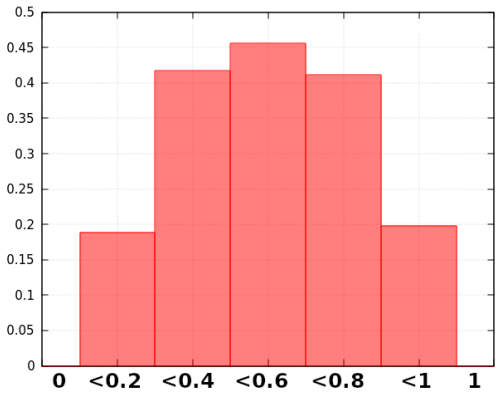


Figure 5.3: Average of the Uncertainty values grouped by range of predictions.

Figure 5.4 shows the frequency of the uncertainty values for the test dataset. We removed all the vectors and predictions with an uncertainty value bigger than 0.05. By doing so, we reduced the starting set of 7555 methods to 2675 methods.

We split the remaining 2675 vectors into three sets: the first set contained the vectors classified with zero, the second set contained the vectors classified with one and the third set all the other vectors. Then, we randomly extracted 40 vectors from the "zero classified" set and 40 from the "one classified" set. We manually checked the methods related to these vectors to confirm the validity of the predictions for the Null Pointer Exception (NPE). Table 5.8 shows the results.

Subset of predictions	**Manual inspection agrees with model prediction**	
	Positive cases	Negative cases
Possible NPE	60%	40%
NPE not-possible	80%	20%

Table 5.8: Manual inspection results.

We agreed with the model predictions in 70% of the cases. As far as the zero classified methods

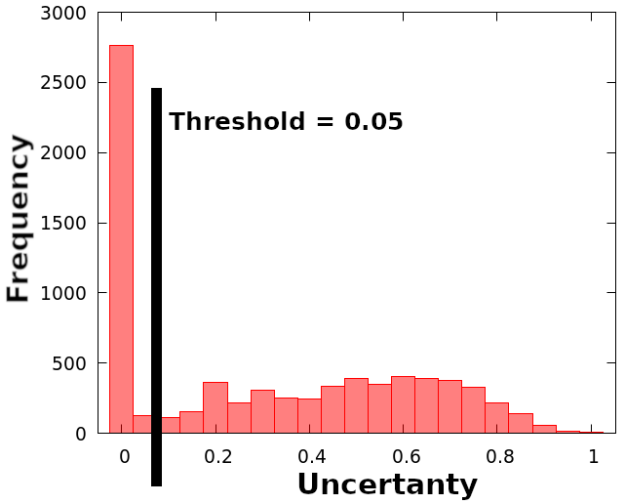


Figure 5.4: Histogram of the uncertainty values for the test project.

are concerned (methods classified as non-buggy), we double checked that it was not possible for any possible variable value to raise a Null Pointer Exception. We did not take into account comments or annotations during the manual verification, in order to evaluate with the same information available to the model. We confirmed that in 60% of the cases a Null Pointer Exception was not possible, as predicted by the model, while in the other 40% of the cases a NPE might have been possible. On the contrary, we agreed with the model in the 80% of the cases for the one classified methods (methods classified as buggy). The complete list of checked methods is reported in Table A.1 and Table A.2 in the Appendix.
For instance, Listing 5.2 shows the method `XYDataImageAnnotation.draw` classified as buggy by the model and we agreed with this classification since at line 6 `plot.getOrientation();` is accessed without checking if `plot` is null.

Listing 5.2: XYDataImageAnnotation.draw method was classified as Buggy by the model and we agreed

```
1  public void draw(Graphics2D g2, XYPlot plot, Rectangle2D dataArea,
2                   ValueAxis domainAxis, ValueAxis rangeAxis,
3                   int rendererIndex,
4                   PlotRenderingInfo info)
5  {
6      PlotOrientation orientation = plot.getOrientation();
7      AxisLocation xAxisLocation = plot.getDomainAxisLocation();
8      AxisLocation yAxisLocation = plot.getRangeAxisLocation();
9      RectangleEdge xEdge = Plot.resolveDomainAxisLocation(xAxisLocation,
10             orientation);
11     ...
12 }
```

One more example is reported in Listing 5.3, that shows the method `DialPlot.getValue` classified as non-buggy by the model. We agreed with this classification since all the variables are checked for null value before using them.

Listing 5.3: DialPlot.getValue method was classified as non-Buggy by the model and we agreed

```
1 public double getValue(int datasetIndex)
2 {
3     double result = Double.NaN;
4     ValueDataset dataset = getDataset(datasetIndex);
5     if (dataset != null) {
6         Number n = dataset.getValue();
7         if (n != null) {
8             result = n.doubleValue();
9         }
10    }
11    return result;
12 }
```

On the contrary, we did not agree with the model in some other cases. For instance, the method `StrokeMap.writeObject` showed in Listing 5.4 was classified as non-buggy by the model but we did not agree with this classification since at line 3 the method `stream.defaultWriteObject();` is accessed without checking if `stream` is null. One more example that can also be considered a corner case is reported in Listing 5.5, where the method `FlowArrangement.add` was classified as buggy even though does not contain code at all.

Listing 5.4: StrokeMap.writeObject method was classified as non-Buggy by the model but we did not agree

```
1 private void writeObject(ObjectOutputStream stream) throws IOException
2 {
3     stream.defaultWriteObject();
4     stream.writeInt(this.store.size());
5     Set keys = this.store.keySet();
6     ...
7 }
```

Listing 5.5: FlowArrangement.add method was classified as buggy by the model but we did not agree

```
1 public void add(Block block, Object key)
2 {
3     // since the flow layout is relatively straightforward,
4     // no information needs to be recorded here
5 }
```

In order to compare our approach with other bug finders, we run FindBugs [8] on the JFreeChart project and filter the results, looking only at the Null Pointer warnings raised by the tool. FindBugs ranks each warning with a value between 1 and 20, where 1 has the greatest priority and means that the vulnerability has the highest probability of generating bugs. FindBugs reported four methods as vulnerable for Null Pointer errors, and we compare them to the results provided by GrapPa for those four methods. The comparison is shown in Table 5.9 and Table 5.10.

Class.Method	**FindBugs Error**	**Bug Rank (out of 20)**
ContourPlot.contourRenderer	Load of known null value	19
ContourPlot.pointRenderer	Load of known null value	19
JDBCXYDataset.executeQuery	Possible null pointer dereference	8
ColorBar.draw	Null passed for non-null parameter	8

Table 5.9: Methods selected by FindBugs that may have vulnerabilities related to Null Pointer Exception.

Class.Method	**GrapPa Output**		**Manual check agreed**
	Prediction value	Uncertainty value	
ContourPlot.contourRenderer	0.0000	0.0001	NO
ContourPlot.pointRenderer	1.0000	0.0473	YES
JDBCXYDataset.executeQuery	1.0000	0	YES
ColorBar.draw	0.6587	0.6115	NO

Table 5.10: Comparison of GrapPa output concerning methods selected by FindBugs.

We also manually checked the four methods selected by FindBugs and we agreed with the warnings. As we can see by comparing the two tables, the GrapPa tool agrees with FindBugs on the vulnerability in the classes `ContourPlot.pointRenderer` and `JDBCXYDataset.executeQuery`, since the output is 1 (classified as buggy) and the uncertainty value small (above the threshold set at 0.05 for both cases). On the contrary, the other two classes selected by FindBugs were marked as non-buggy (`ContourPlot.contourRenderer`) or buggy with a prediction value of 0.6 but an high uncertainty value of 0.6115 (`ColorBar.draw`).

6 Discussion

In this chapter, we report a short discussion about the results presented in Section 5.
Starting the analysis from the entire set of classes of a project, our approach narrows down the methods of a project and helps developers focus their attention on some specific cases and classes that may raise exceptions and errors. Nevertheless, the approach, in its actual form, cannot be compared to other bug finders, since it produces suggestions rather than finding specific bugs and pinpoint them.
The test results, especially when the vectors are classified as non-buggy, show that the trained machine learning model is able to detect the possibility of Null Pointer Exception (accuracy 80%). This means that the model is able to learn from the dataset, but its accuracy is still not precise enough to be really useful as a bug finder tool.
Our results show that the model can learn how to distinguish buggy code from non-buggy one just by looking at examples. Our approach can learn bug patterns without explicitly knowing what the bug is, and without us providing the model a list of rules or patterns related to the vulnerability. This idea makes our approach and the tool easy to extend to new bug patterns since the model needs only a dataset of examples to extract information about the bug behavior.
One of the biggest challenges in developing new approaches able to catch bugs is having to create a list of rules and cases that we think trigger that bug. Designing such a list is a tedious and complex task, and the generic bug finders that automatize this process usually can only detect errors that contradict common behavior, but are not able to identify specific bug patterns without a predefined set of rules. Our approach shows that a machine learning model can overcome this problem by looking at a dataset of examples. Nevertheless, generating and collecting dataset of examples is also a difficult task and we hope that our approach may help future research on this promising idea.
By comparing the results achieved in the validation and test phases by the model, we can see that the validation results are higher than the test results. The MLP in the 10-cross-fold validation achieves 90% of accuracy (see Table 5.6), but this result was not confirmed in the manual verification, where we agreed with the model predictions in just 70% of the considered cases. We think that the model may have learned how to distinguish between mutated and original codes better than actual buggy and non-buggy codes. The differences between the source code files in our dataset were due to the applied mutations. Thus, the synthetic bugs generated may have influenced the training of the model. This intuition would require more experiments to be confirmed.
One more contribution of this work is represented by the three generated datasets (see Table 5.2). Those datasets contain thousands of examples on specific bug patterns and contain both buggy

and clean code. Moreover, by comparing the two different kind of files (the mutated one and the original one), we can precisely detect and pinpoint the bugs, which allows further analysis of the bugs causes. We published the datasets online [52], so that they can be used for further research and extended with more examples.

In conclusion, ours is a novel approach that applies techniques that have been introduced only recently. As far as we know, the CGMM approach was never applied to this kind of problems and we confirmed it to be well suited for vectorizing graphs of different sizes and topology. Furthermore, our proposed variant of the CPG exploits the strength of this data structure but lightens the complexity of the graph, so that it is less redundant for a machine learning model that tries to extract repetitive pattern and sub-structure from it.

Finally our approach is general and can be applied to different programming languages and bug patterns. The results in validation are promising and we would like to keep working on the approach to improve it and provide more contributions to the research field.

7 Limitations and Future Work

In this chapter we present the limitations of the current approach and some ideas on how to improve it.

7.1 Dataset

The dataset is crucial in any machine learning approach. The dataset constitutes the starting point for training the model, and the efficiency and correctness of its predictions are strongly related to the variety and meaningfulness of the dataset. Errors in training data may lead to misclassifications in the test phase. In future work, we would like to apply some related researches to our approach, for instance, the algorithm proposed by Chakarov et al. [33], to evaluate how much the presence of errors in training data has affected the result of our classification task.
In our approach we generate a dataset by mutating the code to inject bugs. This step allows to collect buggy codes without having to manually look for them, but the "validity" of these bugs may be questionable since many of them can be either trivial or equivalent.
We defined a "real bug" as a point in the source code that raises up an unexpected error in testing and is detected by the the test suite. We considered the test suite to be equivalent to the developers expectations on the code. We trusted the robustness of test suites to collect only bugs as close as possible to real ones. Nevertheless, the research would have been more precise if it was based on datasets of bugs extracted from real errors and mistakes. The Defects4j project [57] is a good example of bug dataset, where each bug is labeled and classified by the kind of error, but it does not contain enough examples to train a machine learning model efficiently.
We are aware that this selection and generation of bugs cannot be validated with certainty. The generated bug may be unusual and uncommon, something that no developers will never write, and consequently useless for a machine learning model to learn from real bug examples. Nevertheless, the robustness of the test suite reduces the damage of this problem. Moreover, the necessity of having thousands of buggy codes generated in a short time showed no solutions other than this simple but effective approach.
As far as the bug generation approach is concerned, future improvements would involve improving the datasets variety. We would increase the number of open projects included in the dataset generation process. More projects mean more data and a better representation of code variety for the machine learning model. More data also means more bug patterns available to study, which would improve the number of bug patterns classified by the tool.
Since the manual classification and labeling of bugs is a tedious and hard task, we would like to

also improve the bug generation approach. By improving the effectiveness of bug generation, we would also enhance the efficiency of every approach that makes use of machine learning, because it would have better datasets for training models. Due to this lack of bug datasets, we were not able to extend our spectrum of analyzed bug patterns; research on bug generation would drastically improves approaches similar to ours. More and more data is required for deep learning studies, and we need to find even better techniques to fulfill this need; otherwise, data may become a bottle-neck for future analyses.

7.2 Approach

We could improve our approach by including code annotations in our analysis. Since the intra-procedures analyses are limited to the method analyzed, the approach may classify as buggy any variable provided by an external method, even if that specific variable is already well-tested on the source method. Code annotations and comments written by the developer are useful in that sense. For instance, even though finding null pointer exceptions is a difficult problem, with annotations (e.g., mark what can and can not be null in the method requirements) our approach would have one more hint to improve classification. By including annotations in the approach, and letting the model know about the comment information, we would also reduce the false-positive cases. For instance, Listing 7.1 shows a method classified as buggy by the model because the intra-procedural analysis does not allow the model to explore the method at line 3, which checks if the variable `dataset` is null. This additional check would have helped classify the method as non-buggy instead of buggy.

Listing 7.1: DatasetUtilities.findMinimumRangeValue method classified as buggy by the model but with an inter-procedural analysis the model would recognize the null check at line 3

```
1 public static Number findMinimumRangeValue(CategoryDataset dataset)
2 {
3     ParamChecks.nullNotPermitted(dataset, "dataset");
4     if (dataset instanceof RangeInfo) {
5         RangeInfo info = (RangeInfo) dataset;
6         return new Double(info.getRangeLowerBound(true));
7     }
8     // hasn't implemented RangeInfo, so we'll have to iterate...
9     else {
10        double minimum = Double.POSITIVE_INFINITY;
11        int seriesCount = dataset.getRowCount();
12    ...
13    }
14 }
```

We would also like to apply our approach in different research fields, such as bug predictions and mutants selection.

One more improvement concerns bug localization. We would apply slicing techniques to the method classified as buggy and then present the sliced codes to the model again. If the model still classifies the code as buggy, we can reduce the number of statements and variables that contribute to the bug classification. By repeating this operation for each variable, we would have a probability for each line to be the cause of the error, and so we could present developers with critical alerts pointing to the variables that are more likely to introduce vulnerabilities.

8 Related Work

8.1 Source Code Mutation

Mutation analysis is a well-known and largely used approach to assess the quality of test suites or testing techniques. Mutants are used to measure how good the tests are by observing and comparing the runtime behavior of the non-mutated and mutated programs and quantify the quality of the test suite as the percentage of mutants that the tests can kill. This idea was originally proposed by DeMillo et al. [36] and Hamlet [48].
Several works are available in the literature that studies the validity of this approach to asses test suite quality. Andrews et al. [25] use a middle-sized industrial program with a comprehensive pool of test cases and known faults and compare the behavior of test suites on mutants and real faults; the result suggests that the generated mutants are similar to the real faults. The paper by Ren et al. [58] presents research on the validity of mutants as a substitute for real faults. The authors located real faults that were previously found and fixed by analyzing the open source program's version control. Then, they obtained developer-written test suites, both for the faulty and the fixed program version. Finally, they generated mutants and performed mutation analysis on the fixed program version, and conducted experiments using the real faults, the mutants, and the test suites to compare the differences in the analyses. Their results show a significant correlation between mutant detection and real fault detection, that confirm the idea of the mutants as a valid substitute for real faults.
The recent work presented by Cheng et al. [34] uses a dataset of mutated codes for their experiments. The goal is to find bugs in machine learning programs and algorithm implementations. The authors mutated machine learning programs code to inject bugs in them. Then, they compare the performances of the original algorithms and the mutated ones. The result shows that the 15%-36% of the mutants did not significantly perform worse than the reference classifiers.
Related to our approach for choosing the mutants, the paper by Brown D. [30] introduces a technique to create potential faults that are similar to changes made by actual programmers. The approach consists in generating mutants and then test them on the test suite of the software project, thus select a subset of all the mutants that raise up errors following distance evaluation criteria from real developers changes. The approach aims to generate bugs that are as difficult to kill as ones made by programmers in real-world cases.
The paper by Dolan-Gavitt B. et al. [37] introduces `LAVA`, a dynamic taint analysis-based technique that injects bugs into program source code. The dynamic analysis allows linking each bug with concrete inputs that trigger it. The approach aims to generate a ground-truth corpus of code for

testing bug finders with a dataset of "realistic" bugs, in the sense of bugs that can be triggered with concrete input values.

8.2 Source Code Representation

The thousands of code lines that constitute the source code of software contain information that can be extracted to perform analyses and improve the software itself. Data structures and representations are proposed in the literature, for instance, the survey by Allamanis M. et al [23] presents a taxonomy for navigating in this research field and reports the latest steps taken in developing models that exploit patterns and extract information from source code. The authors also support a website [22] to search and navigate the literature in this area.
The paper by Phan A. et al [71] explores the information contained in programs' abstract syntax trees by using two different tree-based convolutional neural networks and a support vector machine. The approach proposed by Koc G. et al [60] tries to reduce the number of false positive warnings in bug finder tools by applying machine learning models to extracted information from the program structure.

8.3 Bug Finders

Static analysis tools are widely used to detect bugs and errors in software and raise awareness about correctness and security. Developers spend a considerable amount of time looking for bugs, which is an inefficient task. To alleviate this problem, many bug finders were proposed, based on different kind of analyses and approaches. This chapter presents different methods to find bugs that are related to our work.
There are popular tools available in the literature, some of them are also supported and used by the major software companies. This is the case of Google's Error Prone [21] and Facebook's Infer [32]. The paper by Prause C. et al. [73] compares six automated software verification tools on real-world spacecraft software to investigate their effectiveness and efficiency.
One of the most used tools is `FindBugs` [51, 8], which use simple static analysis techniques for finding bugs based on specific bug pattern. It defines a bug pattern as a "code idiom that is likely to be an error". The tool collects more than 50 different bug patterns, and for each of them use simple static analysis to verify if that specific bug affects the inspected code. For instance, `FindBugs` also offers a NullPointerException bug detector which looks for instructions where a null value might be dereferenced using intraprocedural dataflow analysis. It worths noting that `FindBugs` does not support Java 1.9, which is supported by `SpotBugs` [17], that is considered the successor of `FindBugs`.
`PMD` [14, 74] is another tool which inspects Java code using a rules-based approach. It includes a series of rules or patterns considered common in Java application, and more rules can be defined and added to the tool. `PMD` is used to point out inefficient structures such as unused local variables, duplicate import statements, or empty try and catch blocks.
These two mentioned tools (`FindBugs` and `PMD`) are both pattern-based approaches, which use a

pre-defined and extendable list of bug patterns and analyses to detect bugs. `SonarQube` [41, 15] instead adopted a different approach. It is is a quality framework tool, and provides different views over multiple aspects and characteristics of the source code to output the overall quality essence. `SonarQube` supports analysis of the source code files and shows general information as metrics and coverage test information.

Some bug finders infer properties which hold at some code location and pinpoints other code locations that seem to contradict them. The framework presented by Kremeken T. et al. [61] is based on this idea. First, it incorporates information from program analysis and annotations into a graph, with no prior knowledge of the code. Then, they infer the specification of the program by analyzing the graphs and extracting information. This approach can catch the "programmer beliefs", so facts implied by code (for instance, deference of a pointer suggests a belief that is not null). Finally, they provide the specification for the program, which can be used with any rules-based bug finder to locate code location that seems to contradict the inferred belief.

The anomaly-based approach is another proposed method for bug finders available in the literature. These tools learn relations between variables and methods which hold at some code location and flag anomalies in other code location that seem to contradict them. For instance, the `PR-Miner` tool proposed by Li Z. and Zhou Y. [65] extracts association rules from code, so generate links to item $X \Rightarrow Y$ with probability c, which means that if a path contains X, it also includes Y with probability c. Thus, `PR-Miner` finds bugs by detecting violations of these association rules. The main idea is that the programming rules usually hold for most cases and violations happen only occasionally. If some probability c of an association rule is high, the path that violates that rule indicates a potential outlier. A similar idea can be found in the approach adopted by Acharya M. et al. [20], which extracts partial order from API patterns by analyzing the user usage. The authors collect information from the API client code, for instance in which order the code applies the functionalities provided. Thus, they infer from this information the correct and effective API usage based on the idea that the most used approach is the correct one. The proposal of Nguyen T. et al. [70] follows the same path. The method, called `GrouMiner`, mines the usage patterns of objects and classes using a graph-based algorithm. The code is represented as a labeled, directed and acyclic graph, and the usage patterns are considered as sub-graph frequently traversed. Then, they mine all the sub-graphs and extract patterns usage for every object, which can be included in the documentation or used by a rule-based bug finder to point out location that violates these patterns.

The framework `DeepBugs` by Pradel M. and Sen K. [72] presents a learning approach to name-based bug detection. The authors generate examples of incorrect code by transforming a corpus of code and then train a classifier on these correct and incorrect examples. Finally, the trained model was able to distinguish correct from incorrect codes in real-world codes as well.

An approach from the bug prediction research field is the procedure proposed by Kim S. et al. [59], that classifies new software changes as buggy when they are similar to prior buggy changes, extracted from the revision history of a software project. The work by Rahman et al. [50] presents the tool `BugCache`, which inspects the code and rank files by the number of times that a fixing commit was required for that specific file. If a file needs to be fixed continuously, it is probably

inherently buggy or troublesome to handle by the developers, and so requires more attention and tests. This approach was also adopted by Google security engineers [63].
Other approaches presented in the literature make use of dynamic analyses as fuzzing, symbolic execution or combination of them such as the concolic execution. The recent work by Liang H. et al. [66] presents an overview of fuzzing tools and a discussion on the situation of the state-of-the-art. The tool `KLEE`, introduced by Cadar C. et al. [31], runs the analyzed program at first on randomly-constructed input and then use symbolic values and constraint solving model to find more concrete values able to increase the program coverage. The paper by Artzi S. et al. [26] presents `Apollo`, an automated technique for finding errors in HTML-generating web applications. Their approach uses techniques of dynamic test generation and combines concrete, symbolic execution and constraint solving. Jayaraman K. et al. [54] adopted a similar approach for the tool `JFuzz`, a testing tool for Java programs, which starts from a seed input and then use concolic execution to reach high program coverage. The tool `Driller`, introduced by Stephens N. et al. [76], tries to combine fuzzing and concolic execution to improve efficiency. It uses concolic execution to generate inputs that satisfy checks to cover new code branches and then uses fuzzing to explore as many paths as possible avoiding the path explosion of the concolic analysis.

8.4 Graph-based Classifier

We can find several applications of machine learning model on static analysis in the literature.
In the security field, some works apply machine learning model to classify android apps [75, 42]. For instance, Gascon H. et al. [42] present a procedure for detection of Android malware based on structural similarities between the app. They extract function call graphs from android applications and use a support vector machine to distinguish between benign and malicious application.
A recent work presented by Habib A. and Pradel M. [47] use a static analysis that extracts a graph representation from Java classes and then use a combination of graph kernel method and support vector machine to classify the class either as thread-safe or thread-unsafe.
Neural networks model on graphs were adopted by several approaches [45, 64, 35, 44] and have been used in several applications, such as link prediction and classification [46], prediction of variable names given their usage [24], text classification [55, 79], and detection of cross-platform binary code similarity [85].

9 Conclusion

In this work, we present a novel and general approach that makes use of machine learning to train a model on distinguishing between buggy and non-buggy code for a particular bug pattern. We represent the source codes as graphs to provide more information to the machine learning model, such as control and data dependencies between statements. We also mutate the source code of some open source projects to inject bugs and then generate the dataset required to train the machine learning model.

The approach makes use of recent works, as the Code Property Graph [86] for the static analysis, the Contextual Graph Markov Model [27] to convert graphs into vectors, and uses the dropout technique to represent the models uncertainty [40]. Finally, a Multilayer Perceptron model is trained and performs the classification task. We also present a simplified version of the Code Property Graph, which revealed useful for our purposes.

The experiments we conducted to validate the approach show that the trained model was able to distinguish with an accuracy of 70% between methods that may rise a Null Pointer Exception from methods where the Null Pointer Exception was not possible. This shows that machine learning can help develop better bug finder tools. The main strength of our approach resides in the possibility to easily extend the set of bug patterns that it is able to recognize. The approach does not need prior-knowledge on the analyzed code or a list of pre-defined rules to detect a particular bug. By providing a dataset of buggy examples for a specific bug pattern, the approach trains a machine learning on this dataset and produces a model which is able to classify buggy methods for that particular bug pattern.

We implemented the approach in a tool called GrapPa : it already contains three trained models for detecting three different errors in Java code (Null Pointer Exception, Array Index Out of Bounds and String Index Out of Bounds). The GrapPa tool [53] and the three datasets of synthetic bugs [52] are published online to allow further research and analyses.

Acknowledgments

I would like to thank Professor Michael Pradel and Andrew Habib Ph.D., for their guidance, patience and valuable training and advise throughout the project.
This thesis concludes my Master's degree studies, and I am thankful to all the people who supported me during the last two years, both at the University of Twente and the University of Darmstadt, and with whom we had good times. Many people are worth being mentioned, I name without preference order Elia, Alex, Suz, Teresa, Matteo, Come, and Kwadjo.
This project would not have been possible without the support of my parents, my brother, Chiara, and especially my good friends Giorgio, Federico M., Federico E., Jacopo, Letizia, Iacopo, Mauro, Federico F., Nicola, and Pietro. There is no doubt that it would have been impossible to complete this thesis and easily overcome these last months without their great help.
Grazie bimbi, I owe you.

A Appendix

Mutation operator		Example
AOR	(Arithmetic Operator Replacement)	`a + b` ⟼ `a - b`
LOR	(Logical Operator Replacement)	`a ^ b` ⟼ `a | b`
COR	(Conditional Operator Replacement)	`a || b` ⟼ `a && b`
ROR	(Relational Operator Replacement)	`a == b` ⟼ `a >= b`
SOR	(Shift Operator Replacement)	`a >> b` ⟼ `a << b`
ORU	(Operator Replacement Unary)	`-a` ⟼ `~a`
EVR	(Expression Value Replacement) Replaces an expression (in an otherwise unmutated statement) with a default value.	`return a` ⟼ `return 0` `int a = b` ⟼ `int a = 0`

LVR	(Literal Value Replacement)
	Replaces a literal with a default value:
	• A numerical literal is replaced with a positive number, a negative number, and zero.
	• A boolean literal is replaced with its logical complement.
	• A String literal is replaced with the empty String.
		`0` ⟼ `1`
		`1` ⟼ `-1`
		`1` ⟼ `0`
		`true` ⟼ `false`
		`false` ⟼ `true`
		`"Hello"` ⟼ `""`

STD	(STatement Deletion)
	Deletes (omits) a single statement:
	• `return` statement
	• `break` statement
	• `continue` statement
	• Method call
	• Assignment
	• Pre/post increment
	• Pre/post decrement
		`return a` ⟼ `<no-op>`
		`break` ⟼ `<no-op>`
		`continue` ⟼ `<no-op>`
		`foo(a,b)` ⟼ `<no-op>`
		`a = b` ⟼ `<no-op>`
		`++a` ⟼ `<no-op>`
		`--a` ⟼ `<no-op>`

Figure A.1: List of mutations available in Major, as reported in the documentation [12].

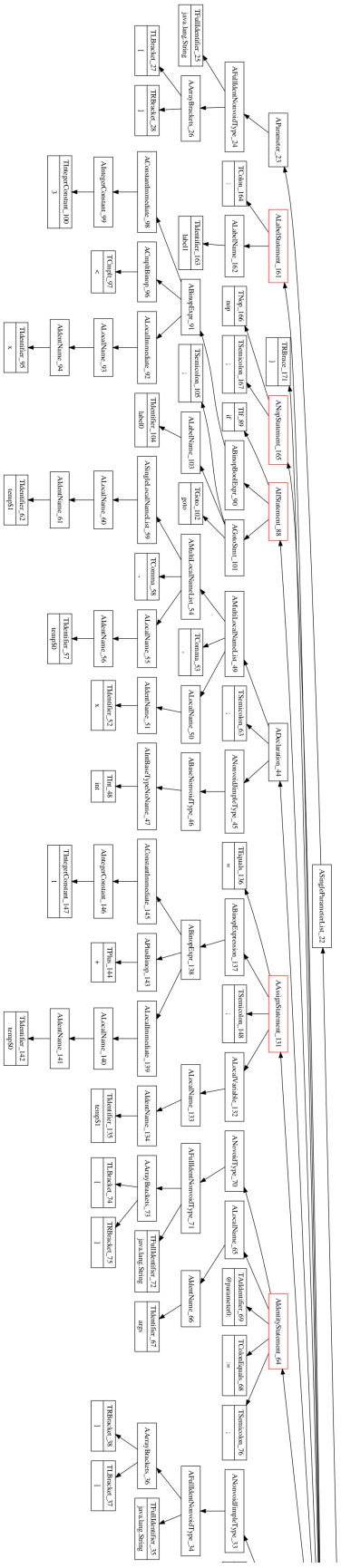


Figure A.2: Complete AST of Listing 2.1, left part.

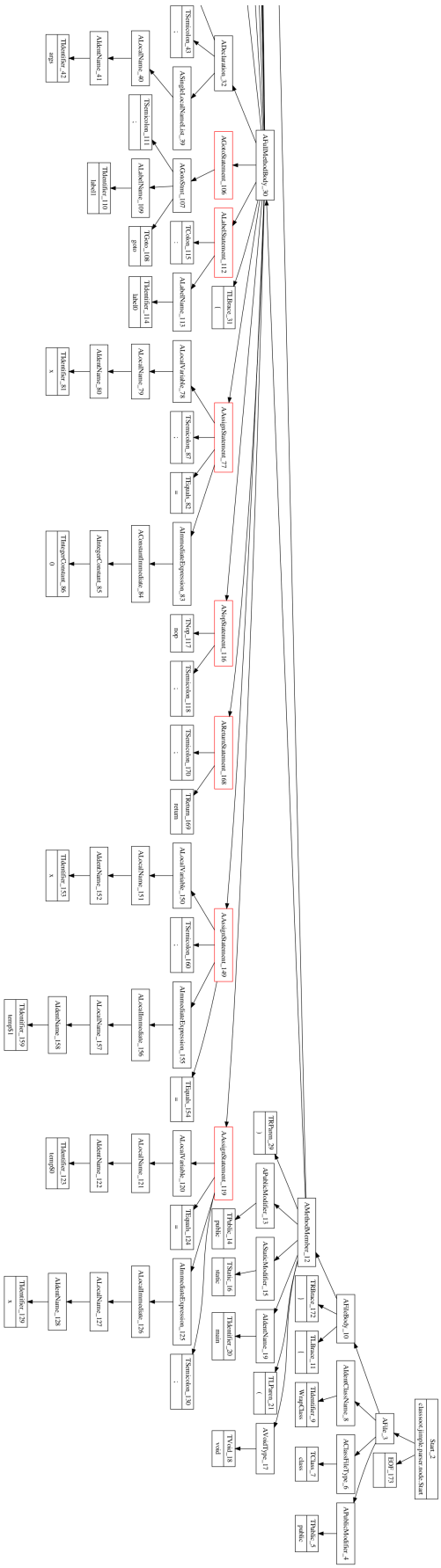


Figure A.3: Complete AST of Listing 2.1, right part.

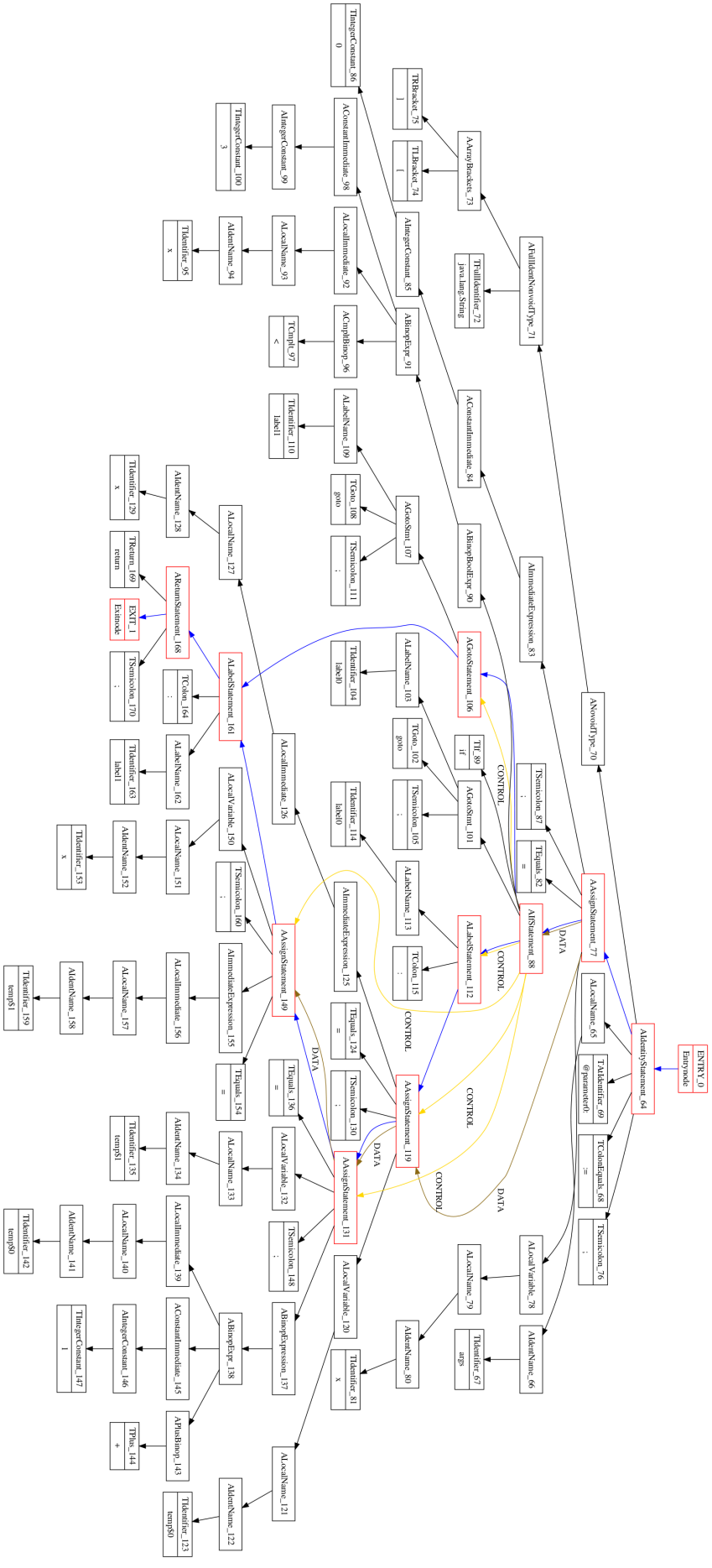


Figure A.4: Complete CPG of Listing 2.1.

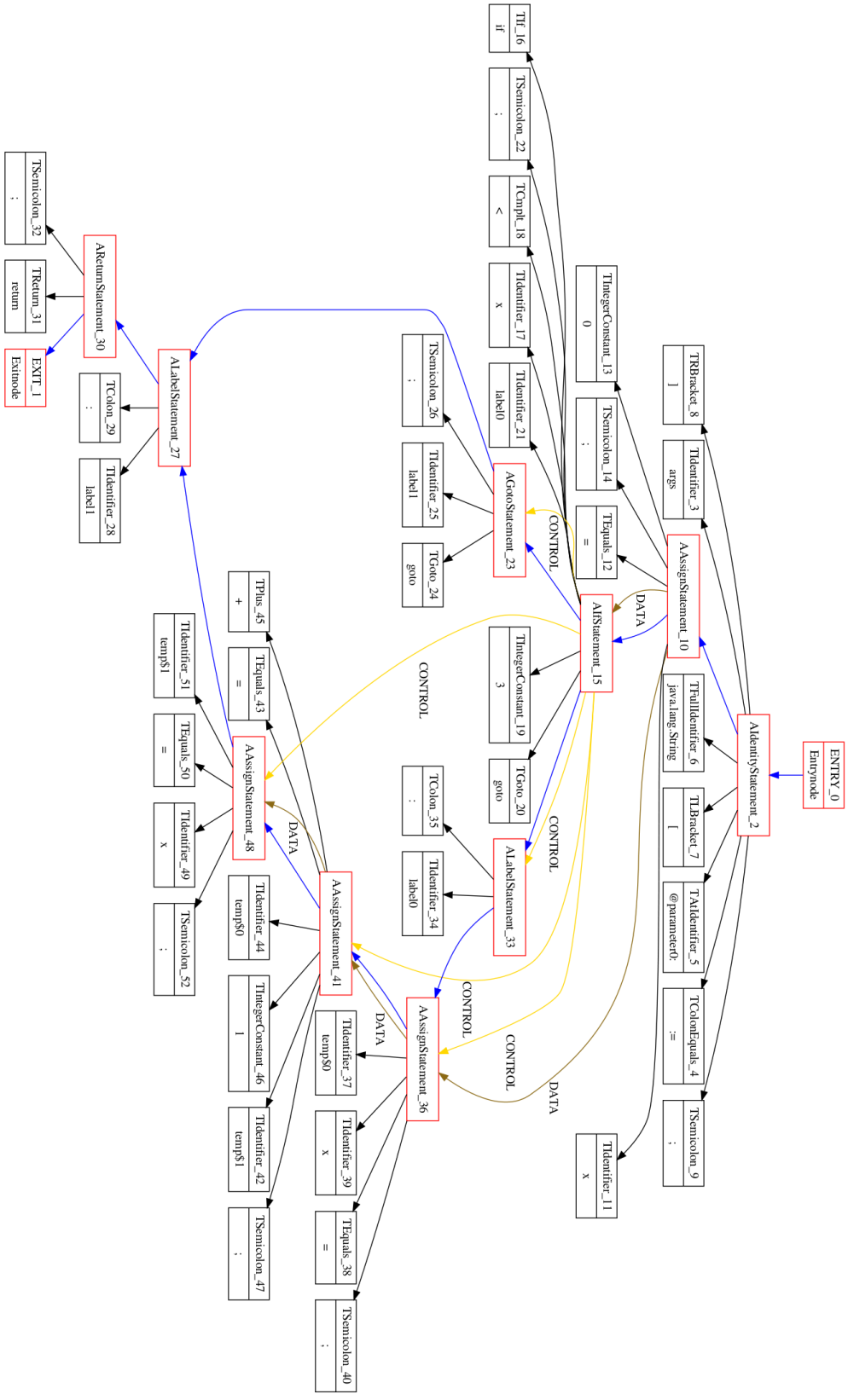


Figure A.5: Complete CPG of Listing 2.1, simplified version.

Class_method	Manual Check Agreed
DefaultKeyedValueDataset_clone	YES
StandardCategorySeriesLabelGenerator_equals	YES
LineAndShapeRenderer_getBaseShapesFilled	YES
XYBoxAnnotation_writeObject	NO
DynamicTimeSeriesCollection_getStartX	NO
XYImageAnnotation_clone	YES
CategoryAxis_removeCategoryLabelURL	NO
DefaultOHLCDataset_getOpen	YES
LegendGraphic_isShapeOutlineVisible	YES
ThermometerPlot_getBulbDiameter	YES
BarRenderer_drawItem	YES
XYIntervalSeriesCollection_getSeriesCount	YES
AbstractCategoryItemRenderer_getBaseToolTipGenerator	YES
StandardChartTheme_equals	YES
MatrixSeries_equals	YES
DialPointer_isClippedToWindow	YES
DefaultNumberAxisEditor_setAxisProperties	NO
PeriodAxis_reserveSpace	NO
JFreeChart_handleClick	YES
AbstractRenderer_getItemPaint	YES
PolarPlot_setAngleTickUnit	YES
ContourPlot_isRangeCrosshairLockedOnData	YES
FXGraphics2D_fillOval	YES
DialPlot_getValue	YES
DefaultBoxAndWhiskerCategoryDataset_getRangeLowerBound	YES
StrokeMap_writeObject	NO
OHLCSeries_getPeriod	YES
SlidingCategoryDataset_getRowKey	NO
DefaultPolarItemRenderer_drawAngularGridLines	NO
LegendItem_setOutlineStroke	YES
ChartProgressEvent_getChart	YES
ChartPanel_getHorizontalTraceLine	YES
AxisEntity_hashCode	YES
Title_setVerticalAlignment	YES
PiePlot_setURLGenerator	YES
CategoryPlot_drawRangeGridlines	YES
TimeSeriesCollection_getDomainOrder	YES
StatisticalBarRenderer_findRangeBounds	YES

Table A.1: Manual check on zero (non-buggy) classified methods for Null Pointer exception.

Class_method	Manual Check Agreed
PinNeedle_hashCode	NO
XYDataImageAnnotation_draw	YES
DatasetUtilities_findMinimumRangeValue	YES
SpiderWebPlot_setDataExtractOrder	YES
VectorDataItem_getYValue	YES
NumberTickUnitSource_previous	YES
AbstractPieLabelDistributor_getItemCount	YES
BlockContainer_getArrangement	NO
CategoryPointerAnnotation_writeObject	NO
JThermometer_setValue	YES
DataUtilities_equal	NO
DataUtilities_calculateRowTotal	YES
ChartPanel_scale	YES
CategoryPointerAnnotation_getArrowLength	YES
StackedXYAreaRenderer_writeObject	YES
DefaultCategoryDataset_incrementValue	YES
CategoryTick_getCategory	NO
KeyedObjects2D_getColumnCount	YES
PeriodAxisLabelInfo_getDateFormat	YES
CombinedRangeXYPlot_handleClick	YES
LegendGraphic_getLineStroke	NO
AbstractCategoryItemRenderer_getLegendItem	YES
CombinedDomainXYPlot_getPlotType	NO
CombinedDomainCategoryPlot_getGap	NO
ChartRenderingInfo_readObject	YES
SlidingGanttCategoryDataset_getPercentComplete	YES
PiePlot_setLabelLinkMargin	NO
CustomXYToolTipGenerator_equals	YES
FlowArrangement_add	NO
DefaultTableXYDataset_getIntervalPositionFactor	YES
CandlestickRenderer_getVolumePaint	NO
TitleEntity_getTitle	NO
XYSeries_clear	YES
DefaultKeyedValueDataset_getValue	NO
PlotOrientation_toString	NO
XYPointerAnnotation_setTipRadius	NO
DialTextAnnotation_getLabel	NO
DefaultIntervalXYDataset_getXValue	YES
XYSeriesCollection_getSeriesIndex	YES
JFreeChart_setAntiAlias	YES

Table A.2: Manual check on one (buggy) classified methods for Null Pointer exception.

Bibliography

[1] Apache commons codec. https://commons.apache.org/proper/commons-codec/. Accessed: 31-08-2018.

[2] Apache commons email. https://commons.apache.org/proper/commons-email/. Accessed: 31-08-2018.

[3] Apache commons io. https://commons.apache.org/proper/commons-io/. Accessed: 31-08-2018.

[4] Apache commons lang. https://commons.apache.org/proper/commons-lang/. Accessed: 31-08-2018.

[5] Apache commons math. https://commons.apache.org/proper/commons-math/. Accessed: 31-08-2018.

[6] Apache maven. https://maven.apache.org/. Accessed: 31-08-2018.

[7] Contextual graph markov model. https://github.com/diningphil/CGMM. Accessed: 14-08-2018.

[8] Findbugs - find bugs in java programs. http://findbugs.sourceforge.net/. Accessed: 14-09-2018.

[9] Gnuplot. http://www.gnuplot.info/. Accessed: 19-09-2018.

[10] Jfreechart - java chart library. http://www.jfree.org/jfreechart/. Accessed: 18-09-2018.

[11] Keras: The python deep learning library. https://keras.io/. Accessed: 14-08-2018.

[12] The major mutation framework. http://mutation-testing.org/. Accessed: 31-08-2018.

[13] Plotly - modern visualization for the data era. https://plot.ly/create/#/. Accessed: 19-09-2018.

[14] Pmd - an extensible cross-language static code analyzer. https://pmd.github.io/. Accessed: 14-09-2018.

[15] Sonarqube - continuous code quality. https://www.sonarqube.org/. Accessed: 14-09-2018.

[16] Soot java optimization framework. https://sable.github.io/soot/. Accessed: 14-08-2018.

[17] Spotbugs - find bugs in java programs. `https://spotbugs.github.io/`. Accessed: 15-09-2018.

[18] Tensorflow: machine learning framework. `https://www.tensorflow.org/`. Accessed: 14-08-2018.

[19] Weka 3: Data mining software in java. `https://www.cs.waikato.ac.nz/ml/weka/`. Accessed: 02-09-2018.

[20] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining api patterns as partial orders from source code: from usage scenarios to specifications. 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*, pages 25–34. ACM, 2007.

[21] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan. Building useful program analysis tools using an extensible java compiler. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 14–23. IEEE, 2012.

[22] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. Machine learning on source code. `https://ml4code.github.io/`. Accessed: 30-09-2018.

[23] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.

[24] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[25] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, (8):608–624, 2006.

[26] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 261–272. ACM, 2008.

[27] D. Bacciu, F. Errica, and A. Micheli. Contextual graph markov model: A deep and generative approach to graph processing. *arXiv preprint arXiv:1805.10636*, 2018.

[28] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 179–190. ACM, 2015.

[29] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[30] D. B. Brown, M. Vaughn, B. Liblit, and T. Reps. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 511–522. ACM, 2017.

[31] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[32] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. OHearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11. Springer, 2015.

[33] A. Chakarov, A. Nori, S. Rajamani, S. Sen, and D. Vijaykeerthy. Debugging machine learning tasks. *arXiv preprint arXiv:1603.07292*, 2016.

[34] D. Cheng, C. Cao, C. Xu, and X. Ma. Manifesting bugs in machine learning code: An explorative study with mutation testing. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 313–324. IEEE, 2018.

[35] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pages 3844–3852, 2016.

[36] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[37] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 110–121. IEEE, 2016.

[38] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 57–72. ACM, 2001.

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

[40] Y. Gal and Z. Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059, 2016.

[41] J. García-Munoz, M. García-Valls, and J. Escribano-Barreno. Improved metrics handling in sonarqube for software quality monitoring. In *Distributed Computing and Artificial Intelligence, 13th International Conference*, pages 463–470. Springer, 2016.

[42] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013.

[43] Z. Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452, 2015.

[44] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.

[45] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, volume 2, pages 729–734. IEEE, 2005.

[46] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[47] A. Habib and M. Pradel. Is this class thread-safe? inferring documentation using graph-based learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 41–52. ACM, 2018.

[48] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software engineering*, (4):279–290, 1977.

[49] S. Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[50] F. R. D. P. A. Hindle and E. B. P. Devanbu. Bugcache for inspections: Hit or miss? 2011.

[51] D. Hovemeyer and W. Pugh. Finding bugs is easy. *Acm sigplan notices*, 39(12):92–106, 2004.

[52] G. Iadarola. Buggy examples datasets - github repository. `https://github.com/Djack1010/BUG_DB`. Accessed: 16-09-2018.

[53] G. Iadarola. Grappa - github repository. `https://github.com/Djack1010/GrapPa`. Accessed: 16-09-2018.

[54] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jfuzz: A concolic whitebox fuzzer for java. 2009.

[55] C. Jiang, F. Coenen, R. Sanderson, and M. Zito. Text classification using graph mining-based feature extraction. In *Research and Development in Intelligent Systems XXVI*, pages 21–34. Springer, 2010.

[56] R. Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 433–436. ACM, 2014.

[57] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[58] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665. ACM, 2014.

[59] S. Kim, E. J. Whitehead Jr, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[60] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter. Learning a classifier for false positive error reports emitted by static code analysis tools. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 35–42. ACM, 2017.

[61] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 161–176. USENIX Association, 2006.

[62] M. Krzywinski and N. Altman. Points of significance: importance of being uncertain, 2013.

[63] C. Lewis and R. Ou. Bug prediction at google. `http://google-engtools.blogspot.com/2011/12/bug-prediction-at-google.html`. Accessed: 25-09-2018.

[64] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[65] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 306–315. ACM, 2005.

[66] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[67] A. Liaw, M. Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[68] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 103–116. ACM, 2007.

[69] B. News. Failed satellite programmed with "wrong co-ordinates". `https://www.bbc.com/news/technology-42502571`. Accessed: 16-09-2018.

[70] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. 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*, pages 383–392. ACM, 2009.

[71] A. V. Phan, P. N. Chau, M. Le Nguyen, and L. T. Bui. Automatically classifying source code using tree-based approaches. *Data & Knowledge Engineering*, 114:12–25, 2018.

[72] M. Pradel and K. Sen. Deepbugs: A learning approach to name-based bug detection. *arXiv preprint arXiv:1805.11683*, 2018.

[73] C. Prause, R. Gerlich, and R. Gerlich. Evaluating automated software verification tools. In *Software Testing, Verification and Validation (ICST), 2018 IEEE 11th International Conference on*, pages 343–353. IEEE, 2018.

[74] D. Rubio. Pmd: A code analyzer for java programmers. `https://www.linux.com/news/pmd-code-analyzer-java-programmers`. Accessed: 14-09-2018.

[75] A. Shabtai, Y. Fledel, and Y. Elovici. Automated static code analysis for classifying android applications using machine learning. In *2010 International Conference on Computational Intelligence and Security*, pages 329–333. IEEE, 2010.

[76] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[77] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *International conference on compiler construction*, pages 18–34. Springer, 2000.

[78] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. *Automated Software Engineering*, 18(3-4):263–292, 2011.

[79] N. Widmann and S. Verberne. Graph-based semi-supervised learning for text classification. In *Proceedings of the ACM SIGIR International Conference on Theory of Information Retrieval*, pages 59–66. ACM, 2017.

[80] Wikipedia contributors. Abstract syntax tree — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Abstract_syntax_tree&oldid=849296366`, 2018. [Online; accessed 21-August-2018].

[81] Wikipedia contributors. Control flow graph — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Control_flow_graph&oldid=838424516`, 2018. [Online; accessed 21-August-2018].

[82] Wikipedia contributors. Keras — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Keras&oldid=848819205`, 2018. [Online; accessed 2-September-2018].

[83] Wikipedia contributors. Soot (software) — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Soot_(software)&oldid=856521825`, 2018. [Online; accessed 31-August-2018].

[84] Wikipedia contributors. Weka (machine learning) — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Weka_(machine_learning)&oldid=857164380`, 2018. [Online; accessed 2-September-2018].

[85] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376. ACM, 2017.

[86] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 590–604. IEEE, 2014.

Erklärung zur Abschlussarbeit gemäß §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Giacomo Iadarola, die vorliegende Master-Thesis / Bachelor-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu fhrt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Thesis Statement pursuant to §23 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Giacomo Iadarola, have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.

I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.

In the submitted thesis the written copies and the electronic version for archiving are identical in content.

Datum/Date	**Unterschrift/Signature**