Automated Vulnerability Prediction in Software Systems and Lightweight Identification of Design Patterns in Source Code

Jeffy Jahfar Poozhithara

A Masters Thesis submitted in partial fulfillment of the requirements of the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2021

Committee

Hazeline Asuncion

Brent Lagesse

Erika Parsons

Program Authorized to Offer Degree:

Computing & Software Systems

University of Washington

Abstract

Automated Vulnerability Prediction in Software Systems and Lightweight Identification of Design Patterns in Source Code

Jeffy Jahfar Poozhithara

Chair of the Supervisory Committee:
Associate Professor Hazeline Asuncion
Computer Science and Systems

Software development companies put a heavy investment in fixing security vulnerabilities in their products after code development. This demands an automated mechanism to identify security vulnerabilities during and after software development. One approach is to include possible solutions like security design patterns during design. This reduces system-wide architectural changes required and enables efficient documentation and maintenance of the software systems. Further, identifying which design patterns already exist in source code can help maintenance engineers determine if new requirements can be satisfied. The current techniques for design pattern identification require either manually labeling training datasets or manually specifying rules or queries for each pattern. As part of this research, we took a two-pronged approach: 1. Pre-implementation: predict vulnerabilities before any source code is written, to increase awareness of possible risks while developing the system. 2. Post-implementation: check the source code to identify any missing security patterns, based on the identified vulnerabilities.

For the first approach, we created a Keyword Extraction-based Vulnerability Identification System (KEVIS) that uses natural language processing techniques to extract keywords and n-grams from software documentation to predict security vulnerabilities in software systems. We analyzed the correlation of certain keywords and n-grams with the occurrence of

various security vulnerabilities as well as the correlation between different vulnerabilities. Additionally, we analyzed the performance of classification algorithms (Logistic Regression, Support Vector Machines, K-Nearest Neighbors, Multi-level perception, and Random Forest) in the prediction. To enable the analysis, we also created a dataset by mapping over 200,000 vulnerability reports on the CVE website with technical/functional documentation of 3602 products. The preliminary analysis shows that the performance of KEVIS is comparable or better than the prediction using source code as well as other static analysis methods.

For the second approach, we introduced PatternScout, a technique for automatically generating SPARQL queries by parsing UML diagrams of design patterns, ensuring that pattern characteristics are matched. We discuss key concepts and the design of PatternScout. Our results indicate that PatternScout can automatically generate queries for the three types of design patterns (i.e., creational, behavioral, structural), with accuracy that is comparable, or perform better than, existing techniques.

Due to the difference in concepts used for both approaches and ease of explanation, the background, literature review, method, results, and discussions corresponding to each approach is discussed separately in their own sections (Approach 1 - Automated Vulnerability Prediction in Software Systems, and Approach 2 - Lightweight Identification of Design Patterns in Source Code, respectively).

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

The time and money invested by product development companies in fixing security vulnerabilities after releasing software systems into production environments are significantly high. Some projects undergo continuous revisions for up to 12 years incorporating perfective, corrective, and adaptive changes to the software project [65]. Many software projects undergo security testing after deployment in pre-production environments. Vulnerabilities detected at this point can lead to multiple revisions and rewriting of software code including regression testing causing much delay in the timeline beyond the initial plan. According to their analysis of various security flaws, it was demonstrated that every security vulnerability in the software could have been eliminated prior to release via proactive software maintenance engineering. A reliable mechanism to list all potential vulnerabilities that are likely to be encountered based on the functional specifications would help in incorporating design patterns before beginning the development stage. Early identification of software vulnerabilities is essential in software engineering and can help reduce not only costs but also prevent loss of reputation and damaging litigations for a software firm[68]. Such an approach can also support mitigating vulnerabilities during the design phase (i.e., through the use of security design patterns). This will reduce system-wide architectural changes required post-development and enable efficient documentation and maintenance of the software systems.

While there is a large body of work on defect prediction, the body of work on vulnerability prediction is smaller. Vulnerabilities differ from defects in that there are many fewer security flaws in code compared to defects, and defect prediction techniques do not directly transfer to the task of vulnerability prediction [68]. Although vulnerabilities are a specific type

of software defect, the problem of finding vulnerabilities in software differs in significant ways from the more general problem of finding defects. The most obvious difference is quantitative: there are typically many more defects than vulnerabilities reported. The rarity of vulnerabilities means that the classes of vulnerable and neutral software components are severely unbalanced, which increases the difficulty of building effective prediction models [123].

In comparative studies of different machine learning-based vulnerability prediction methods, it was observed that some details such as datasets, always have a significant influence on the experimental results [146]. One of our contributions in this study is an extensive dataset mapping product documentation text with the CVE dataset thereby enabling the application of natural language processing on the product documentation for more in-depth vulnerability analysis.

Deep learning techniques including Recurrent Neural Networks (RNNs) [62], [133], [117], [85], Convolutional Neural Networks (CNNs) [79], [57], [130], and Deep Belief Networks (DBNs) [131], [141] have been successful in image and natural language processing. While it is tempting to use deep learning to detect vulnerabilities, there is a "domain gap": deep learning is born to cope with data with natural vector representations (for example, pixels in images). In contrast, software programs do not have such vector representations [83]. To overcome this challenge, many methods have converted source code into representations that can be used as inputs to these machine learning algorithms (for example, Abstract Syntax Tree[83]). [35] shows that the documentation alone may already contain information for predicting the presence of some logic flaws, even before the code is analyzed. In this study, we try to use the technical and functional specification documents of software systems, which are readily available as text (.txt, .pdf, .html files) to find the correlation between keywords or n-grams in these files with the vulnerabilities that were reported in the corresponding software product. This approach is referred to as *Approach 1: Automated Vulnerability Prediction in Software Systems* for the remainder of the paper.

Design patterns are general-purpose solutions to recurring software engineering problems.

Significant work has gone into identifying and cataloging design patterns and security design patterns [21] [56] [47], but these patterns assume that a developer is working on the design phase of the software. Many times, however, a maintenance engineer works on an existing codebase, and it is unclear which design patterns already exist in the source code.

The ability to identify existing patterns in source code is especially important for legacy code that needs to meet new security requirements. For example, security engineers must be able to quickly understand which, if any, existing security mechanisms (i.e., security design patterns) have been designed into the existing codebase. The identified security design patterns can then be compared with security requirements.

Since design patterns assist with satisfying requirements, maintenance engineers need to determine which patterns already exist in the code. Finding design patterns can be time-consuming, due to manual work required to reverse engineer the code [128]. Techniques for automated detection also have challenges. Mining techniques involve the time-consuming task of manual labeling training dataset and manual checking results, due to the false-positive results [53] [126] [144]. On the other hand, detection using rules [94] and queries [100] also involve the manual specification of design patterns and suffer from false-negative results, as they lack the capability of handling design pattern variations. This challenge is especially pronounced in security design patterns, which have a higher level of variability than object-oriented design patterns [127] [33].

Meanwhile, Semantic Web technologies provide a means of encapsulating rich information about source code, able to capture design- and code-level concepts such as class relationships, visibility, and properties, which are not captured in graph representations of source code (e.g., abstract syntax trees). In Semantic Web, information is represented as a graph known as Resource Description Framework (RDF). There are several query technologies to extract information from RDF; among these SPARQL is the most popularly used. Running queries on RDF yields highly accurate results, as these essentially only retrieve results with the matching graph pattern. However, as we mentioned, this type of technique could also suffer from false negatives, as the results obtained are very specific. Furthermore, using semantic

web query languages, like SPARQL, is difficult because the user needs to learn not only the query language syntax, but also the vocabulary and relationships in the data [120] [70] [102]. This is why researchers have developed automated techniques for generating SPARQL queries [58] [64] [27], but none of these techniques cater to the domain of software.

To address these challenges, we developed PatternScout which leverages the advantages of Semantic Web technologies while overcoming its difficulties. First, we overcome the limitation of the difficulty of using SPARQL queries by automatically generating queries from UML Class and Sequence diagrams. This is a feasible approach since repositories of common design patterns have already been created [21] [56] and they already include UML Class diagrams in their descriptions. This also applies to security design patterns as many of them also include Class diagrams [47] [76]. Second, we overcome the limitation of low recall by creating a catalog of known design pattern variations [105] and running through all different variations when examining source code. This approach is referred to as *Approach 2: Lightweight Identification of Design Patterns Source Code* for the remainder of the paper.

The research questions for Approach 1 are as follows: First, is there a significant correlation between keywords/n-grams extracted from software documentation (technical/functional/user guides) with security vulnerabilities reported in the system post-development? Second, are different security vulnerabilities significantly correlated with each such that detection of one vulnerability can be used as a warning for the other. Third, how do different binary classification algorithms perform in terms of predicting security vulnerabilities by training on keywords and n-grams extracted from software documentation? Fourth, does a perceptron model or multilayered neural network perform better in predicting security vulnerabilities using the same features? Finally, can the dataset created as part of this research be used to create a pre-trained model to predict security vulnerabilities for new software products before their development?

Software companies create a technical and functional specification document before beginning to develop the product. They also release documentation/manuals/user guides after the release of the project into production. CWE [5] and CVE [7] maintain a registry of vul-

nerabilities reported in different software products over the years. This also includes details of the different versions of the vulnerability that were detected and its severity. By applying natural language processing techniques to the documentation, we analyzed the correlation of certain keywords and n-grams with the occurrence of various security vulnerabilities as well as the correlation between different vulnerabilities. To enable this analysis, we created a dataset of technical/functional documentation of software systems and the security vulnerabilities that were reported in each system over the years. Additionally, we analyzed the performance of binary classification algorithms (Logistic Regression, Support Vector Machines, Random Forest Classifier, K-Nearest Neighbors Classifier, and Multi-level perceptron) in predicting security vulnerabilities in software systems using keywords and n-grams extracted from software documentation.

Our preliminary results for Approach 1 are as follows: The model was able to achieve an average accuracy of 84.5%, precision 81%, recall 82%, and an f1-score 0.81 using data from 3602 products. We identified that the Rapid Automated Keyword Extraction (RAKE) [109] algorithm was able to identify keywords that drastically reduced computational complexity without compromising on performance compared to features extracted using a TF-IDF Vectorizer. Due to the imbalanced nature of the dataset, a balanced class weighted Random Forest Classifier was found to be the most effective binary classifier for predicting vulnerabilities using documentation text. Further, we were also able to identify highly correlated vulnerabilities (reported together in software systems) by analyzing over 209000 vulnerability reports.

We know that security design patterns can be used to address security vulnerabilities [47]. Hence, by predicting vulnerabilities that can come up during development or testing, developers can decide the design patterns to be incorporated while making architectural decisions. This would help in significantly reducing development and post-production maintenance related to security vulnerability fixes. Future prospects of this research include creating a design pattern recommendation system that can suggest security design patterns to be included in source code based on the vulnerabilities predicted.

Our main contribution as part of Approach 2 is a novel technique to generate SPARQL queries that can correctly identify design patterns. Compared to other methods, our approach is more lightweight because it does not involve any manual training and does not require defining rules or queries. The only requirement is a design pattern to find (in the form of a Class diagram). The second contribution is that PatternScout incorporates behavioral aspects of a pattern in addition to structural characteristics by incorporating stereotypes, filters, and sequence diagrams when necessary. The third contribution is a repository of SPARQL queries that contains 23 GoF design patterns and their 107 variants [105]. Finally, we offer lessons learned to improve accuracy.

We assessed PatternScout using experiments to measure accuracy and performance measures. Our accuracy measures indicate that they are also comparable, or perform better, than existing techniques (e.g., [100], [26], [22], [125], [30]). Our performance measures indicate that execution time is related to the number of patterns detected.

This paper is organized as follows. We start with a comparison of our automated vulnerability prediction method with existing static analysis and source code analysis techniques. This is followed by a comparison of PatternScout with existing methodologies is discussed in Section 2. We then discuss the natural language processing techniques, binary classification algorithms, and vulnerability databases used for Approach 1, and the modeling and Semantic Web technologies used for Approach 2 (Section 3). This is followed by the Design and Evaluation of each approach (Section 5). Further, we discuss our evaluations, limitations, and lessons learned in Sections 6, 7, and 7.2.4 respectively.

Chapter 2

LITERATURE REVIEW

2.1 Approach 1

While many researchers have used vulnerability databases like CVE, CWE, NVD for predicting vulnerabilities [146][49][92][84][82][80], we are the first to apply machine learning techniques on technical/functional documentation along with these databases as the ground truth for predicting vulnerabilities in the design phase.

The construction of a security vulnerability identification system based on text classification was introduced in [115]. The paper summarizes the system requirements and establishes a machine learning-based vulnerability text classifier that uses the descriptions of vulnerabilities on CWE. It introduces the word segmentation, feature extraction, classification, and verification processing of vulnerability description text. Their contributions are mainly in two aspects: One is to standardize the unified description of vulnerability information, which lays a solid foundation for vulnerability analysis. The other is to explore the research methods of a vulnerability identification system for information security and establish a vulnerability text classifier based on machine learning, which can provide a reference for the research of similar systems in the future. While our study relies on Vulnerability Databases like CWE and uses machine learning-based text classification, our approach differs from their method in that their inputs are vulnerability descriptions whereas ours are specification documents.

Researchers have used CVE, CWE, NVD datasets for studying vulnerability characteristics [80] and life cycle. For example, [49] proposes an automated system that uses free-form descriptions of newly-disclosed one-day vulnerabilities to extract the most probable affected software from the description, when a vulnerability is a high-value asset used sparingly to attack high-value targets. Identifying which systems are vulnerable can be achieved by ex-

tracting relevant keywords from the free-form vulnerability description and forwarding them to an alert service monitoring specific keywords related to these systems (such as names of public software used in the system). In our study, we use keyword extraction on software documentation to match against vulnerability reported in software products on CVE.

There are many studies that analyze the influence of factors for vulnerability detection [146] [81] [114] [78] [121] [138]. They have conducted comparative studies to evaluate the impact of different factors including the classification algorithms, feature extraction methods, the class imbalance methods, vectorization methods for vulnerability detection, the user-defined name replacement and the source of datasets. These analyses are focused on identifying vulnerabilities in source code by using the code snippet samples in these datasets. For example, [83] uses the abstract syntax tree of the code for analysis. These machine learning-based vulnerability detection can be divided into traditional machine learning-based detection and deep learning-based detection. Previous studies on traditional machine learning-based vulnerability detection include the studies [52] [18] [74] [75] [77] [113] [36] [38] [61] [104] [91] [34] [139]. Most of the existing techniques for the vulnerability prediction investigated the correlation between vulnerabilities and component characteristic(s) like code churn, code complexity, dependencies, code coverage [147], developer activity metrics [118], import statements [93], code gadgets (number of semantically related lines of code) [84],organizational measures and actual dependencies while [68] uses the analysis of raw source code as text. In this research, we build on top of these methods by developing a model that can be used to predict vulnerabilities even before the development of the system by removing the dependency on source code.

[97] proposes a hybrid technique based on combining N-gram analysis and feature selection algorithms for predicting vulnerable software components where features are defined as continuous sequences of tokens in source code files, i.e, Java class files. While these methods perform well in detecting security issues using source code, they do not enable making architectural and design decisions to avoid these security vulnerabilities before the development phase.

On a macroscopic level, KEVIS is similar to the Logic-flaw predictor in [35] in terms of using automated document analysis leveraging Natural language processing techniques. They proposed that the detection of logic flaws, which currently relies on program analysis that leverages the functionality information reported in the program's documentation, can be made more efficient by executing automated text analysis on syndication developer's guide. They use dependency parsing and word embedding to automatically recover semantic information from the wrapper's integration instructions documented by the developer's guide. They compared it against the finite state machine (FSM) of the payment service encapsulated to infer the relation between the syndication payment process and the payment FSM, maps important payment states to the related instructions in the guide, and recovers the parameters for required security checks. However, this method is restricted to the domain of payment systems. Moreover, the method was only tested on 5 payment services that had the necessary syndication guides. Our method is applicable for software systems across various domains and was tested on 3602 products and 209000 vulnerability reports. While methods like [35] require domain-specific pre-training, our pre-trained model can be executed over any software system with a functional/technical specification document or user guide.

Many of the existing techniques focus on a specific domain (e.g payment services [35]) or programming language (PHP [123], Java [97], C/C++ [146]) in their predictions. Our focus was to create a predictor that can be used across domains, even before the software is built when the architectural decisions are being made (including the programming languages to be used), once the functional and non-functional requirements are created.

The machine learning-based techniques that rely on source code suffer from another disadvantage: overfitting. Cross-project vulnerability prediction, where a vulnerability prediction model is trained on one project and tested on another project (or another codebase), depends on the notion that certain underlying attributes of source code will universally indicate the presence of vulnerabilities, regardless of which codebase or project contains these vulnerabilities. However, there is the potential that some attributes will indicate the presence of vulnerabilities in one project but not in another. This suggests that in practice, the

cross-project prediction will have to contend with two kinds of overfitting–overfitting to particular characteristics of software modules (which can also occur in within-project prediction) and overfitting to characteristics of individual projects (a problem unique to cross-project prediction) [123].

Static analysis is the process of evaluating a system or component based on its form, structure, content, or documentation, which does not require program execution [71]. The basic idea of static analysis is that it checks program text statically to discover defects or weaknesses in the program; especially those that can lead to vulnerabilities. Manual code inspection is a traditional static analysis technique, but it requires spending lots of time and inspectors to have prior knowledge about the vulnerabilities to detect. There are tools to support static analysis, but they do not make static analysis fully automated as the outputs still need to be verified by a person [86]. Also, some vulnerabilities in programs can never be found by static analysis [96]. Static analysis can only approximate programs' behaviors. The former tells us we can not think the system to have no vulnerability when the output of the static analysis is like that" no vulnerability found". The latter implicates results of static analysis are not perfect. False-negative and false-positive are two typical problems. Because static analysis is undecidable, false negative is strictly unavoidable. While impacts of false-negative are more serious than false positive, false positive is more disgusting. False-positive requires humans to verify and confirm the outputs of static analysis, which notes static analysis must work with a human. Another characteristic of static analysis is its conservatism, meaning that its results are not precise enough and maybe not useful enough [50]. Most of the existing static analysis techniques are aimed at source code [86].

2.2 Approach 2

Earlier approaches for detecting design patterns in source code ranged from sub-graph matching [145], [143], [63] and ontology based techniques [100], [98] to using machine learning techniques [126], [53],[144], [125]. A detailed meta-analysis of various design pattern mining approaches is discussed in [46]. We focus our discussion here on techniques that leverage the

Semantic Web.

RDF triples can be manually constructed from UML diagrams [124]. Automated construction of RDF triples from Abstract Syntax Tree (AST) is enabled in CodeOntology [25]. Our approach, meanwhile, generates SPARQL queries automatically from Class and Sequence diagrams to find design patterns. While previous methods use UML diagrams to create RDF triples, our approach creates queries. Automatic generation of SPARQL queries is needed, due to high learning curve associated with creating these queries [102][120][70].

There are ontology-based approaches to identifying design patterns (e.g., [100], [98]). One approach uses Semantic Web technologies to automatically detect design patterns [100]. This technique only supports variants that have the same number of target components. A closely related work also detects design patterns using Class and Sequence diagrams [98]. This technique does not detect patterns from source code but design diagrams. Both techniques require the manual creation of queries or rules for each pattern to be identified. Our technique requires the presence of Class or Sequence diagrams, many of which already exist in literature.

Other detection techniques are as follows. Matching design patterns have been performed using UML diagrams [28] and using text-based similarity check and verification of roles [112]. Many techniques use source code metrics and machine learning to detect patterns [126] [144] [53] [125]. An advantage of PatternScout is that it can accommodate variations in implementation without compromising on the accuracy and it does not require manual training for each pattern. The pre-processing required is the generation of RDF graphs of source code (for example, see Preparation Time in Table 6.4). We also generated SPARQL queries of variations of design patterns identified in literature [105], to minimize any pre-processing time. Finally, approaches also exist to detect patterns from documentation [42]. Our technique detects patterns from the source code.

Chapter 3
BACKGROUND

3.1 Approach 1

In this section, we provide background on various concepts we use for creating KEVIS.

3.1.1 Natural language processing

In our research, we utilized two NLP techniques to automatically analyze the documentation: tokenization and word embedding, as explained below.

Keyword Extraction

Tokenization is the process of splitting text data (sentences, paragraphs, documents) into words/unigrams or n-grams. There are numerous word segmentation or tokenization methods available for text mining. For extracting keywords from the documentation files, the Rapid Automatic Keyword Extraction Algorithm (RAKE)[110] [109] was chosen. We used the implementation called RAKE NLTK from the Natural Language Toolkit (NLTK) library [14]. Compared to other keyword extraction algorithms like TextRank (Undirected, co-occurrence window), Ngram with tag, NP chunks with tag, and Pattern with tag, RAKE has performed better or comparable in identifying correct keywords. It ensures that stopwords are not part of the keywords extracted. Stopwords are words frequently encountered in text data. For text data in the English language, this list consists of words like *a, the, at, of* etc. It also provides flexibility in passing a custom stoplist if we wish to exclude a different set of stopwords. Based on this benchmark evaluation, RAKE was found to have effectively extracted keywords and outperformed the state of the art in terms of precision, efficiency, and simplicity [109].

Word Embedding

Word Embedding is a set of language modeling and feature learning techniques that map text (words or phrases) from a vocabulary to high-dimensional vectors of real numbers. Such mapping can be implemented in different ways. The state-of-the-art word embedding tool, Word2vec [90], initializes word representations by random values and uses as its input a joint probability distribution of words' context by applying a continuous Bag-of-Words or a skip-gram model. This distribution is then utilized during the training of a classifier.

3.1.2 Binary Classifiers

We selected the optimal classifier for our model by comparing the performance of the following 5 classifiers. For all classifiers, a random state variable was set for the reproducibility of the results. For each classifier, a grid search was conducted evaluating the performance of different hyperparameter combinations. A grid search is a K-fold cross-validation of all possible permutations of parameters specified in the parameter grid. In each case, the highest performing hyperparameter combination after a 5 fold cross-validation was selected to be used in the benchmarking between classifiers. The highest performing hyperparameter combination of each classifier is mentioned in the respective section.

K-Nearest Neighbor Classification

Neighbors-based classification is a type of instance-based learning or non-generalizing learning that does not attempt to construct a general internal model, but simply stores instances of the training data [59]. Classification is computed from a simple majority vote of the nearest neighbors of each point. A query point is assigned the data class which has the most representatives within the nearest neighbors of the point. The k-neighbors classification [135] is the most commonly used neighbors-based classification technique. The optimal choice of the value k is highly data-dependent: in general, a larger k suppresses the effects of noise but makes the classification boundaries less distinct. In this study, we set the value of k=5 and

the distance metric as Minkowski distance with a uniform weighting of all neighborhoods.

Logistic Regression

Logistic regression (LR), also known as the log-linear classifier computes probabilities describing the possible outcomes of a single trial using a logistic function. In our evaluations, we used the LR model with l2 penalization with regularization parameter (C) set to 0.01, the maximum number of iterations allowed for the solvers to converge set to 1000 using a Limited-memory BFGS [54] solver.

Support Vector Machines

Support vector machines [37] (SVMs) are a set of supervised learning methods used for classification, regression, and outliers detection. The main advantages of support vector machines and why it was selected for our analysis was that it is effective in high dimensional spaces, even in cases where the number of dimensions is greater than the number of samples. As it uses a subset of training points in the decision function (called support vectors), it is also memory efficient. When the number of features is much greater than the number of samples, choosing Kernel functions and regularization terms is crucial to avoid over-fitting. For our evaluations, we used the regularization parameter C=0.3, RBF kernel, and the balanced class weight mode to automatically adjust weights inversely proportional to class frequencies in the input data.

Multi-level Perceptron

Multi-layer Perceptron (MLP) [134] is a supervised learning algorithm that can learn a non-linear function approximator for either classification or regression. It is different from logistic regression, in that between the input and the output layer, there can be one or more non-linear layers, called hidden layers. The reason we chose MLP as a candidate classifier is its capability to learn non-linear models. However, MLP with hidden layers has a non-convex

loss function where there exists more than one local minimum. Therefore different random weight initializations can lead to different validation accuracy. MLP requires tuning several hyperparameters such as the number of hidden neurons, layers, and iterations. In our evaluations we used 2 hidden layers with 5 hidden units, the maximum number of iterations allowed for the solvers to converge set to 1000 with rectified linear unit function(relu) activation.

Random Forest Classifier

A random forest (RF) is a meta estimator that fits several decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting [55]. RF has received increasing attention in recent years due to its ability to produce excellent classification results with a rapid processing speed [103]. The main training options in the RF model are the use of the maximum number of trees, the variable number required for the split search, and the variant for the sampling process. RF was chosen as a candidate classifier in our analysis as it is known to reduce overfitting in decision trees and to improve accuracy. In our analysis, we set the number of trees in the forest to be 100 with a maximum depth of 10. The class weighting was set to a balanced subsample as a countermeasure for the class imbalance problem discussed in Section 5.1.4.

3.1.3 Vulnerability Databases

At present, some security service organizations around the world have established their own vulnerability libraries [115]. Common network vulnerability libraries mainly include CVE (Common Vulnerabilities and Exposures) [5], NVD (US National Vulnerability Database) [13], and CNVD (China National Vulnerability Database) [2]. Among them, each CVE vulnerability has a unique name corresponding to the CVE dictionary, which helps users distinguish vulnerabilities from vulnerability databases and detection tools [60]. If the vulnerability in the information security monitoring report belongs to a vulnerability in the CVE database table, then the corresponding patch solution can be obtained through the

vulnerability name to solve the information security problem in time. For example, a CVE vulnerability number is CVE-2008-1046.

3.1.4 Vulnerability Classification

Vulnerability classification refers to the classification of vulnerabilities. From the perspective of mathematical thinking, the vulnerability classification process is a mapping process. It classifies vulnerabilities that need to be classified into existing vulnerability categories according to a certain mapping relationship. The vulnerability category refers to the type of vulnerability, which is divided into categories based on attributes such as the cause of the vulnerability, the scope of action, the technology used, and the location characteristics [115].

CWE provides information about weaknesses for over 900 different software and hardware quality and security issues. A hierarchical system of five types of abstraction is utilized to provide clarity and understanding of the relationships between weaknesses. Four well-defined hierarchical types are reserved for weaknesses, from most abstract to most specific: Pillar, Class, Base, and Variant. These abstraction types correlate with the type of information contained within the CWEs as described by different dimensions: behavior, property, technology, language, and resource [7]. A sample hierarchy visualizing the vulnerability classification is shown in Figure 3.1.

3.2 Approach 2

In this section, we provide background on various technologies we use for PatternScout.

3.2.1 Modeling

UML is a general-purpose modeling language intended to provide a standard way to visualize the design of a system [31]. A UML Class Diagram has components like Classes and Interfaces which in turn contain attributes and operations. Classes are connected via relationships like Generalization, Association, Composition, Collaboration, and Interface Realization.

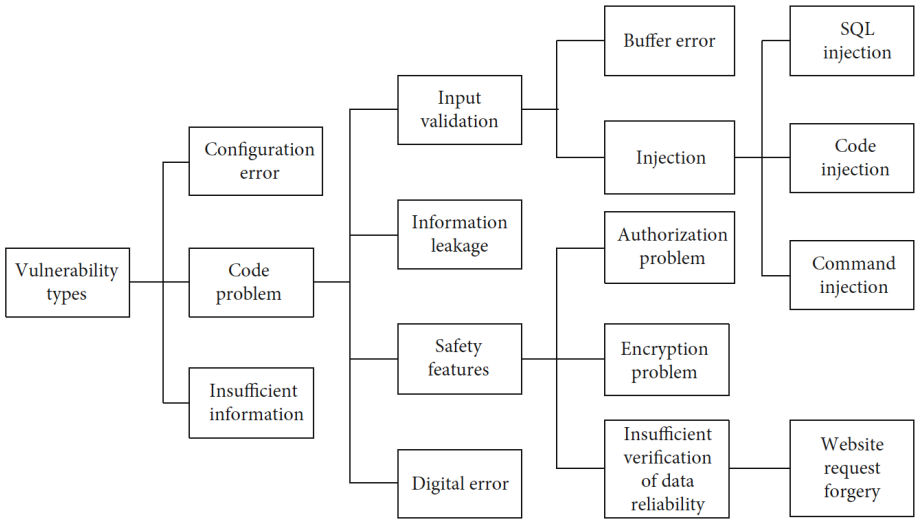


Figure 3.1: Classification of Vulnerabilities to Parent Pillars and Child categories

Various UML editing tools enable users to create UML Diagrams. We used StarUML [122] and Visual Paradigm [129] to create UML Class Diagrams and Sequence diagrams of the design patterns. These tools were selected due to the availability of free community editions that also supported exporting UML Diagrams to XMI format. The StarUML XMI plugin converts model files (.mdj) and model fragments (.mfj) of UML diagrams to XMI files. Visual Paradigm also has default features to export the diagram to XMI files. These XMI files serve as input to PatternScout. Since most UML Modeling tools provide the converted XMI files in a standard format according to the UML specifications, PatternScout is agnostic to the tool used to create XMI files.

3.2.2 Modeling Metrics

SDMetrics is an OO design quality measurement tool for UML [136]. SDMetrics analyzes the structure of UML models and works with all UML design tools that support XMI. Although the software is rich with features like comprehensive design measurements and automated design rule checks, the only functionality we use in this project is the Open Core library to

parse UML diagrams stored as XMI. It supports all currently used XMI versions.

3.2.3 Design Pattern

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. There are three main types of patterns: creational, structural, and behavioral [56]. Creational patterns provide the capability to create objects based on a required criterion in a controlled way. Structural patterns provide ways to organize different classes and objects to form larger structures and provide new functionality. Finally, behavioral patterns identify common communication patterns between objects.

Creational patterns are generally divided into two categories: class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns make effective use of delegation.

Structural patterns focus on class and object composition. Structural class-patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

Behavioral patterns are specifically concerned with communication between objects. Objects communicate and interact with each other in two ways: by calling (or invoking) each other's methods or by directly accessing their variables.

While we generated SPARQL queries for all the 23 patterns and their variants listed in Table 3.1, we evaluated PatternScout on representatives from each pattern types: Singleton, Factory Method and Prototype patterns from the Creation patterns category, Adapter pattern from Structural patterns category, and Strategy, State, Observer, Command and Template Method patterns from the Behavioral patterns category. These patterns were studied in detail and used for evaluations in [137] [87] [142] [125] [106] [29] and [20].

Pattern Type	**Design Pattern**	**Class Diagram**	**Sequence Diagram**	**# Variants**
Creational	Singleton (SGLT)	✓	×	12
	Abstract Factory (AF)	✓	×	6
	Builder (BLD)	✓	✓	2
	Factory Method (FM)	✓	✓[1]	9
	Prototype (PRTT)	✓	✓[1]	1
Structural	Adapter (ADPT)	✓	✓	3
	Bridge (BRGE)	✓	✓[1]	8
	Composite (CMPT)	✓	✓[1]	11
	Decorator (DCRT)	✓	✓[1]	6
	Façade (FCD)	✓	×	3
	Flyweight (FLWT)	✓	×	2
	Proxy (PRXY)	✓	✓[1]	9
Behavioral	Chain of Responsibility (COR)	✓	×	3
	Command (CMD)	✓	✓	4
	Interpreter (ITPT)	✓	×	2
	Iterator (ITRT)	✓	✓	9
	Mediator (MDT)	✓	×	3
	Memento (MMT)	✓	✓	1
	Observer (OBSV)	✓	✓	2
	State (STT)	✓	×	5
	Strategy (STTG)	✓	×	1
	Template Method (TPLT)	✓	×	2
	Visitor (VSTR)	✓	✓	3

Table 3.1: Design Patterns and Generated Queries

[1]Sequence diagram was constructed to represent method invocations that were represented as comments in GoF specification

3.2.4 Semantic Web

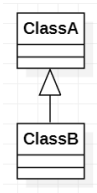
PatternScout is built on top of various Semantic Web technologies: RDF, CodeOntology, and SPARQL. An RDF graph is a finite set of RDF triples [89]. RDF triples contain facts, which are relationships between resources. Resources are represented as nodes, relationships are represented as edges. The vocabulary for RDF graphs is three disjoint sets: a set of URIs V_{uri}, a set of bnode identifiers V_{bnode}, and a set of well-formed literals V_{lit}. The union of these sets is called the set of RDF terms. An RDF triple is a tuple $(s, p, o) \in (V_{uri} \cup V_{bnode}) \cdot V_{uri} \cdot (V_{uri} \cup V_{bnode} \cup V_{lit})$ [119].

CodeOntology is a building block of the Web of Code, an attempt to represent code as a semantic graph [25] [24]. Its framework is composed of three main components: Ontology, Parser, and dataset.

The ontology component is designed to model OO programming languages. It is written in OWL 2 Web Ontology Language [67]. It is mainly focused on the Java programming language, but it can be replaced to support other OO languages. The modeling process underlying the creation of the ontology has been inspired by a re-engineering of the abstract syntax tree (AST).

The parser component analyzes Java code to serialize it into RDF triples. Internally, the RDF triple extraction is managed by Spoon [99] processor invoked for every package in the input project. The RDF serialization process is handled using Apache Jena. It can extract structural information common to all OO programming languages, like class hierarchies, methods, and constructors. Optionally, it can also serialize into RDF triples all statements and expressions, thereby providing a complete RDF-ization of source code. RDF serialization occurs as follows. First, the project is analyzed to download its dependencies and load them in the classpath. Then an AST of the source code and its dependencies is built and processed to extract a set of RDF triples.

We also use SPARQL queries. A building block for SPARQL queries is Basic Graph Patterns (BGP). A SPARQL BGP is a set of triple patterns. A triple pattern is an RDF




```
1 PREFIX woc: <http://rdf.webofcode.org/woc/>
2
3 SELECT ?ClassA ?ClassB
4 WHERE{
5 ClassA a woc:Class .
6 ClassB a woc:Class .
7 ClassB woc:extends ClassA .
8 }
```

Figure 3.2: Simple inheritance relationship SPARQL query

triple in which zero or more variables might appear. Variables are taken from the infinite set V_{var} which is disjoint from the above-mentioned sets [119].

SPARQL query has PREFIX, SELECT and WHERE sections (see Figure 3.2). PREFIX defines the database schema being queried from. SELECT defines the attributes being extracted. WHERE defines the various conditions that must be met to extract results. The WHERE clause can have one or more conditions including a FILTER.

Chapter 4

USE CASE

4.1 Approach 1

Although some organizations have established information description databases for information security vulnerabilities, the differences in their descriptions make it difficult to apply security precautions [115] [111]. The use case of KEVIS is to enable developers without the domain expertise of cybersecurity to make design decisions that mitigate security vulnerabilities. That is, for a system that is in the design stage (before development) if KEVIS predicts a product to potentially have a vulnerability, secure design patterns or architectural and design decisions can be made by foreseeing the vulnerability so that numerous developer hours adding security fixes post-production can be avoided. Similarly, for a system that is already developed, if a vulnerability is detected, the correlation analysis can be used to identify and proactively correct a vulnerability before it can be exploited by malicious actors.

4.2 Approach 2

The use case of PatternScout is to enable developers who do not have the knowledge of SPARQL queries to use it to detect design patterns in source code without having to navigate the learning curve associated with SPARQL. Further, PatternScout is also intended to overcome the limitations of existing design pattern detection tools like the requirement for pattern-specific pre-training, accounting for implementation variants, and to be able to support ad hoc patterns.

Chapter 5

METHOD

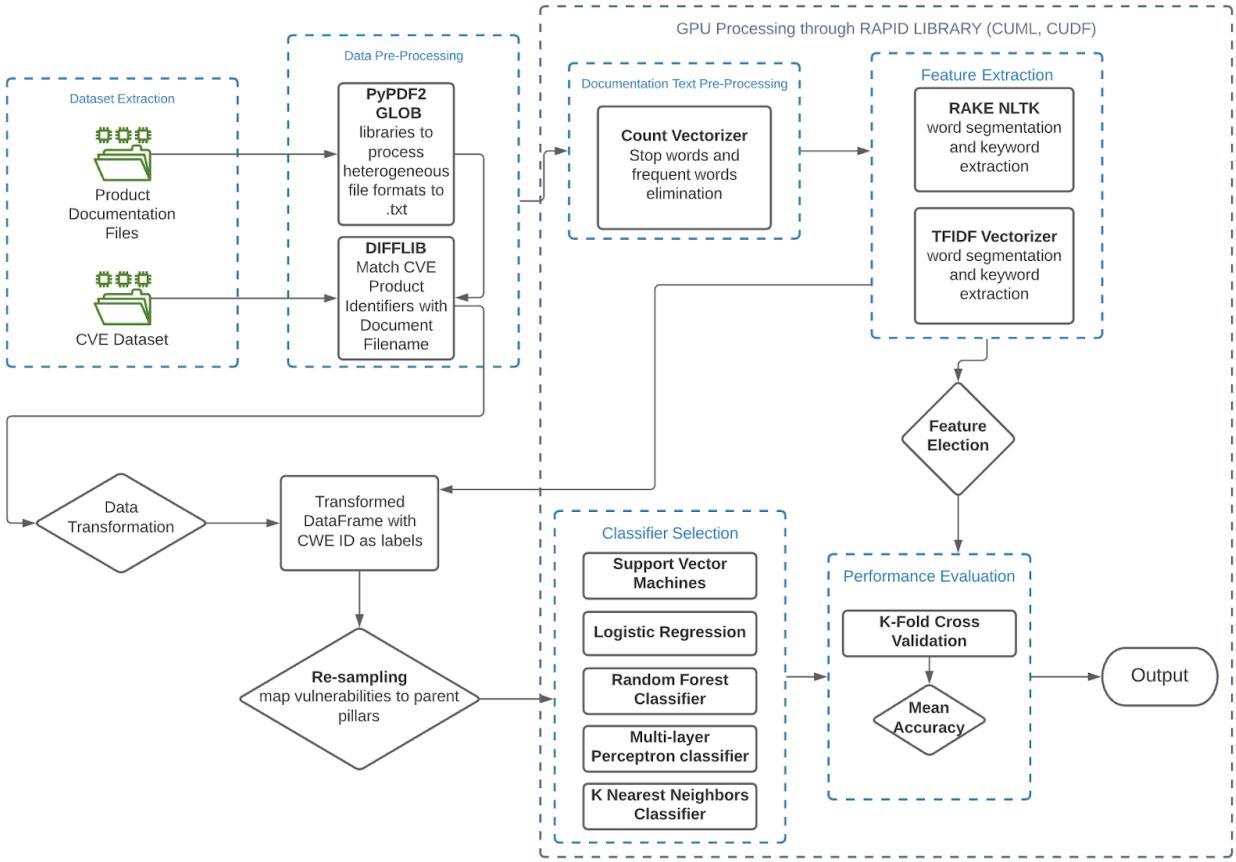


Figure 5.1: Flow Diagram for Keyword Extraction-based Vulnerability Identification

5.1 Approach 1

The architecture of our model is shown in Figure 5.1. The method involved data collection and dataset creation, data pre-processing, classifier and feature selection, and model

evaluation. Each of these steps is discussed in detail in the following sections.

5.1.1 Dataset Extraction

We use Natural Language processing techniques to automatically extract keywords and n-grams correlated with the occurrence of vulnerabilities from the technical or functional specifications document. For creating the pre-trained model, we extracted specification documents of 3602 products whose vulnerabilities are listed on CVE[5]. CVE is a list of publicly disclosed cybersecurity vulnerabilities that is free to search, use, and incorporate into products and services. [6] provides an easy-to-use web interface to CVE vulnerability data. It summarizes vulnerabilities reported per product by each vendor grouped by year. We developed a program that automatically downloads all the product names grouped by vendors which are then used to download vulnerabilities found in each of these products along with the year in which the vulnerability was reported as well as the CVE and CWE IDs associated with the vulnerability. The documentation files and CVE vulnerability reports were downloaded using the web scraping method (see Section 5.1.1).

Web Scraping

To download specification documents for the products mentioned in the CVE data, each company's website needed to be parsed. A web scraping program was created to apply the required security handshake protocols to fetch data (where user guides are in HTML or JSON format) from API/Web service endpoints or to download files in .txt or .pdf formats. A similar parser was used to download vulnerability reports from the CVE website.

Although the data parsed from CVE contained over 209000 vulnerability reports from 47345 products, due to difficulty in obtaining specification documents of the products, only 3602 products were considered for the analyses in this paper.

The project documentation or user manuals as well as the CSV files created are maintained in a distributed filesystem. For evaluations, the documentation of each product is imported and an automatic keyword extraction algorithm is applied. The top keywords or

	Vendor	Product	Documentation	79	20	119	200	264
0	Ffmpeg	Ffmpeg	b'ffmpeg Documentation Table of Contents 1 Syn...	0	1	1	0	0
1	Ffmpeg	Lavf Demuxer	b'Libavfilter Documentation Table of Contents ...	0	1	0	0	0
2	Ffmpeg	Libavcodec	b'Libavcodec Documentation Table of Contents 1...	0	0	0	0	0
3	Ffmpeg	Libswresample	b'Libswresample Documentation Table of Content...	0	0	0	0	0
4	Debian	APT	b'Translation(s): Deutsch - English - Espa\xc3...	0	1	1	1	0
...	...	...	...	...	...	...	...	...
3016	Symantec	Web Security	b"Assigning permissions to Partner portal user...	1	0	1	0	0
3017	Symantec	Web Security.cloud	b"Assigning permissions to Partner portal user...	0	0	1	0	0
3018	Symantec	Winfax Pro	b"About the Software Portal Software Portal Th...	0	0	1	0	0
3019	Symantec	Workspace Streaming	b'About Symantec\xe2\x84\xa2 Workspace Streami...	0	0	0	0	1
3020	Symantec	Workspace Virtualization	b'About Symantec\xe2\x84\xa2 Workspace Streami...	0	0	1	0	1

Figure 5.2: The dataset created for evaluating the model

n-grams selected by the product are mapped against the product name as shown in Figure 5.2. The keywords extracted from the documentation can be made as features to train a classifier model and the dataset can be split as Training set and Test set for evaluating performance. After training the model on the train set, the model is tested on the test set.

The problem of vulnerability prediction can then be simplified as a series of binary classification problems if we focus on predicting one vulnerability at a time. Hence, if the model performs a binary classification in each iteration for a single CWE-ID, it essentially becomes a sequential binary classification. In each iteration, the model will train on a single CWE ID label and predict 1 or 0 corresponding to that label for all entries. Formulating the task as a multi-class classification problem would be appropriate if each product was likely to have only one vulnerability. Since each product can belong to multiple classes (have multiple vulnerabilities) at the same time, binary classifiers were selected for evaluation.

Product Name Matching

Since the product name mentioned on CVE does not always match the file name, a direct text matching cannot be done to map the documentations to the CVE data. While some differences would be minimal like differences in the capitalization of letters, other differences could be the presence of special characters or abbreviations. To overcome this challenge, a text similarity scoring library: difflib [9] was used to identify the closest matching filenames for each product referred to in the CVE data. A cut-off threshold of 0.6 was set so that a documentation file is fetched only if there is a potential match. The product name as per CVE and corresponding matching filenames found in the website parsing output is shown below:

```
1
2  ----Libswresample-----
3  ['libswresample', 'libswscale', 'Audio resampler']
4  ----Git Changelog-----
5  ['git-changelog_', 'changelog-history_', 'gitlab-logo_']
6  ----Gearman-----
7  ['gearman-plugin_', 'variant_', 'vagrant_']
8  ----Gerrit Trigger-----
9  ['gerrit-trigger_', 'urltrigger_', 'ivytrigger_']
10 ----Mod Pagespeed-----
11 ['PageSpeed Module', 'Cloud Storage', 'Cost Management']
```

Even though more than 7000 files were downloaded, due to mismatch in the filenames or due to re-branding or changing of ownership due to company buyouts, the final dataset used for evaluations contained only 3602 products. When the cutoff applied was too low (0.3 - 0.5), the number of false positives was significantly high. When the cutoff applied was too high (>0.7), the number of false negatives was significantly high. At a cutoff of 0.6, we were able to achieve a significant number of True positives with very low false positives. By ensuring that only the subset of the CVE dataset corresponding to a specific vendor was compared with documentation files of the same vendor, the false positives were reduced to 0. To resolve false negatives, manual verification and mapping are required. For the preliminary

analysis, the missing samples (false negatives) were eliminated from the dataset.

5.1.2 Data Pre-Processing

The pre-processing required for this dataset involved the following.

1. **Homogenization of file formats to plain text** - The input formats of the word segmentation and binary classification algorithms require the data to be plain text or strings.

2. **Verification of the encoding and content of text data extracted from PDF and HTML file types** - Due to the difference in character encoding across filetypes, the extracted text needed to be manually verified to avoid keyword extraction algorithms from confusing wrongly decoded content as relevant features.

3. **Elimination of products that did not have vulnerability reports** - The rows without documentation cannot be transformed to feature set for the binary classifiers. A feature set is the set of attributes that act as input to a machine learning model. In this case, the set of keywords extracted from documentation files is the feature set.

4. **Transformation of the vulnerability reports to a (feature set, labels) pair format** - Binary classifiers require the dataset to be split as a feature set and labels. The feature set acts as the input. The output is compared with the labels to compute various performance metrics.

Each vulnerability report in the CVE data is given a CVE-ID. The number of unique CVE IDs in the data was large (86664 labels for 47345 products) as new CVE IDs are generated every year for the same vulnerability to maintain version details and granular descriptions. However, CWE IDs are not changed every year and are a more reliable label for the classification we intend to perform as part of this research. For the 3602 products in consideration, there were only 164 unique CWE IDs in the dataset. Each product receives

a 1 or 0 label corresponding to each CWE ID depending on the presence (represented by 1) or absence (represented by 0) of the vulnerability. Each product may contain multiple vulnerabilities.

5.1.3 Feature Extraction

We used the TF-IDF vectorizer [132] and RAKE [109] to tokenize the documentation files (break down the content to words/unigrams, bigrams, and trigrams). While TF-IDF vectorizer internally conducts a word embedding task to convert each text sample to a feature vector, a word embedding algorithm was used to convert keywords extracted using RAKE to a feature vector. Both TF-IDF Vectorizer and RAKE support specifying custom stopwords. RAKE helps in reducing the number of features extracted significantly due to the lemmatization step which a TF-IDF Vectorizer does not use. Lemmatization is the process of reducing different versions of the same word into the root form. For example, *write, written, writing* etc are counted as a single feature *write* instead of considering them separately. For the same size of the sample (1000 products), TF-IDF Vectorizer extracts 168531 features compared to 23203 features extracted by RAKE.

CountVectorizer to Extend Stopwords

While the default stopwords list includes frequent words in the English language, this specific dataset will involve frequent occurrences of certain technology-related terms like *technology, service, application, solution* etc, which do not assist with identifying vulnerabilities. To identify additional stopwords, Countvectorizer [132] was used to extract words that have high frequency across the documentation files in the dataset. This list was added to the default stopwords list so that keywords extracted do not involve these. This enables feature reduction as the vectorizers will otherwise associate significance to these terms in the feature set.

The top keywords extracted by TF-IDF Vectorizer and RAKE using different metrics are shown in Table 5.1. It was observed that the Degree_to_Frequency_Ratio metric of RAKE

performed the best in identifying keywords that are related to different functionalities and technologies used (for example, rfc0959 is an FTP protocol).

TF-IDF Vectorizer	RAKE		
	WORD FREQUENCY	DEGREE TO FREQUENCY RATIO	WORD DEGREE
fdk	ncompute	ncompute	headaches
mainframe	rfc0959	rfc0959	peap
latency	securex	securex	complexity
reviewing	rudimentary	rudimentary	x98tl
subsystems	envmon	envmon	1493
fpga	unhidden	unhidden	accompanies
fulfillment	polls	polls	dreams
pak	x98sub2	x98sub2	sequences
insights	whois	whois	006d
anyone	po_message_iterator	po_message_iterator	29436
nexus	nfamily	nfamily	370w
js	typeset	typeset	namespaces
getting	_ltlibraries	_ltlibraries	posix_trace_attr_get streamsize
started	shasum	shasum	x98none
maximo	dist_lisp_lisp	dist_lisp_lisp	orthis
sterling	memoir	memoir	calc
ibm	keyctl	keyctl	compiler_needs_object
netcool	nfailures	nfailures	parking
drupal	nsuccess	nsuccess	nbd

Table 5.1: Top Keywords extracted by each Feature Selection Method

Further, the computational efficiency and overall prediction accuracy when using these feature sets were also compared as summarized in Table 5.2. It was observed that keywords

Feature	TF-IDF Vectorizer				Rake			
	stopwords=en		stopwords=custom		stopwords = en		stopwords=custom	
Model Name	Mean	Max	Mean	Max	Mean	Max	Mean	Max
KNeighbors	0.768	0.795	0.766	0.815	0.809	0.835	0.768	0.835
LogisticRegression	0.841	0.845	0.841	0.845	0.76	0.815	0.76	0.815
MLP	0.841	0.845	0.841	0.845	0.736	0.82	0.753	0.82
RandomForest	0.836	0.845	0.83	0.845	0.841	0.845	0.844	0.845
SVC	0.8	0.835	0.8	0.835	0.812	0.835	0.812	0.835
Features	**168837**		**168531**		**26978**		**23203**	

Table 5.2: Feature Election By comparing the performance of TF-IDF Vectorizer and RAKE

extracted by RAKE enabled drastically reduce the number of features to be considered without compromising on accuracy. For example, the highest mean accuracy observed when using TF-IDF Vectorizer was 84.1% (with 168531 features) while the highest mean accuracy when using RAKE was 84.4% by using only 23203 features.

5.1.4 Class Imbalance Strategies

The task of vulnerability prediction, like any fraud detection or anomaly detection model, inherently faces the challenge of an imbalanced dataset. That is, the number of samples labeled as 1 (i.e., presence of the vulnerability) will be significantly smaller than the number of samples labeled as 0. This is shown in Figure 5.3. Out of 3602 products, even the more common vulnerabilities like CWE 707 and CWE 664 are found only in 1390 and 1361 products which account for less than 50% of the samples. When training models using imbalanced datasets, unless countermeasures are explicitly added, models tend to overestimate the likelihood of a test sample to be labeled as 0. That is, when samples are predominantly of class 0, predicting every test sample as 0 can give a high accuracy when other measures (precision, recall, f1-score, ROC AUC) are compromised. This phenomenon,

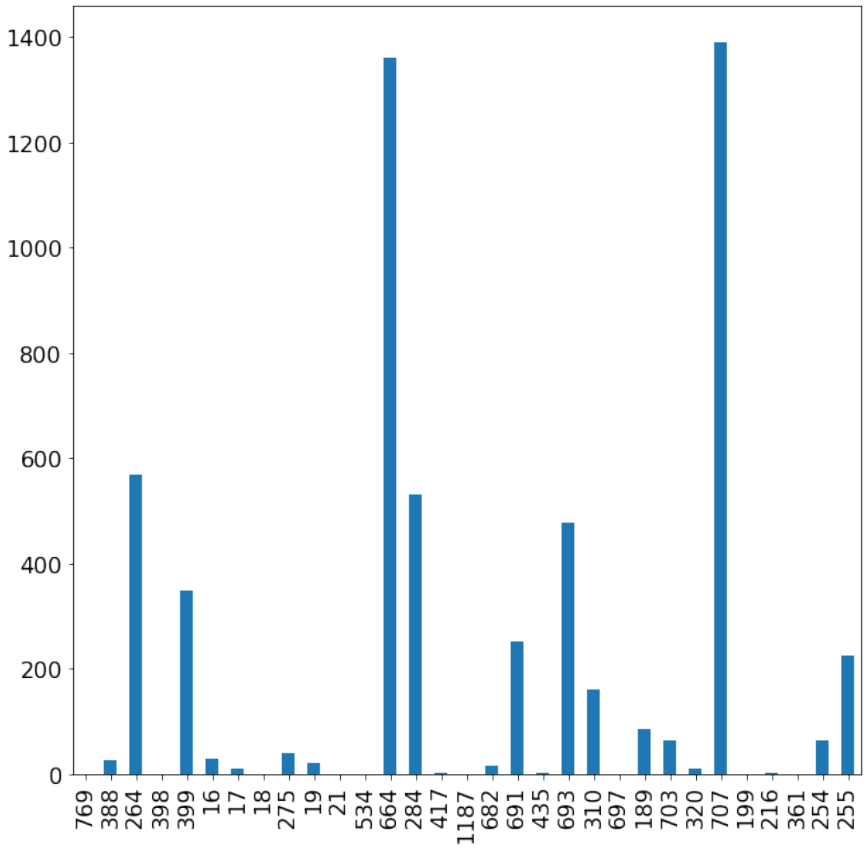


Figure 5.3: Analysing number of samples per CWE ID

called the accuracy paradox, is observed in this dataset as well, as shown in Table 5.3. For example, it can be seen that when using a K Nearest Neighbors model, even though the accuracy is high (80.9%), the precision and recall are drastically low at 3.87% and 14.71% respectively. The countermeasures for class imbalance are as follows.

Re-sampling and Under-sampling Majority Class

To correct the prior probability calculation by a classifier, the number of samples from each class should be balanced. By down-sampling, the majority class (reducing the number of samples) can help the model avoid overestimating the likelihood of the majority class[72]. This can be done by selecting all minority class items and randomly selecting the same

model name	recall	precision	f1	roc_auc	accuracy
KNN	0.0387	0.1471	0.0561	0.4247	0.809
LR	0.0129	0.0071	0.0092	0.4206	0.76
MLP	0.0516	0.0682	0.0477	0.3796	0.751
RF	0	0	0	0.3537	0.841
SVC	0.0129	0.02	0.0156	0.3769	0.812

Table 5.3: High accuracy observed when all other metrics are compromised due to imbalance in the dataset

number of samples from the majority class. The model can then be trained on the re-sampled balanced dataset. To check if the model has learned features correctly to predict both classes, the model can be tested on Stratified Data (data with the ratio of samples of each class matching the original dataset).

Alternate Metrics

In the past few years, several new metrics which measure the classification performance on majority and minority classes independently, hence taking into account the class imbalance, have been proposed[88]. When executing cross-validation to benchmark the performance of different classifiers, alternate metrics like recall, f1-score, or Matthews correlation coefficient can be used that will flag models that cannot counter the class imbalance effect. In this study, we analyzed precision, recall, f1-score, and ROC AUC score in addition to accuracy when benchmarking classifiers. We used the following formula to compute each of these metrics:

$$\text{Accuracy} = \frac{\text{True Positive} + \text{True Negative}}{\text{True Positive} + \text{True Negative} + \text{False Positive} + \text{False Negative}}$$

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive + False Negative}}$$

$$\text{f1-score} = \frac{\text{2.Precision.Recall}}{\text{Precision + Recall}}$$

ROC AUC score is computed as explained in [51].

As the metrics above are calculated for each class, the combined score calculated as weighted average and macro average for each metric as follows:

$\text{Score}_{weighted\_avg} = f_0 \ . \ Score_0 + f_1 \ . \ Score_1$

$\text{Score}_{macro\_avg} = 0.5 \ . \ Score_0 + 0.5 \ . \ Score_1$

where f_1 is the fraction of samples labelled 1 and f_0 represents the fraction of samples labelled 0 corresponding to a CWE ID in the dataset. Similarly, $Score_0$ represents the value of the metric in predicting samples of class labelled 0 and $Score_1$ represents the value of the metric in predicting samples of class labelled 1. For example, weighted average precision is calculated as follows:

$\text{Precision}_{weighted\_avg} = f_0 \ . \ Precision_0 + f_1 \ . \ Precision_1$

Penalized Algorithms

Some algorithms have penalties or regularization parameters to counter the class imbalance. For example, SVM and Random Forest classifiers support "balanced" class weight mode that automatically adjusts weights inversely proportional to class frequencies in the input data.

5.1.5 Classifier Selection

To identify the classifier model that works best in this context, a multi-metric grid search was conducted with 5 fold cross-validation. The grid search compared the precision, recall, f1-score, ROC AUC score (Area Under the Receiver Operating Characteristic Curve) for the

5 classifier models. The 5 fold cross-validation allows to check for any overfitting happening in the models as in each fold, different samples are used as training and test set. The metrics compared are the average of the metrics observed in each iteration of the cross-validation. The multi-metric grid search also helps to avoid any accuracy paradox scenarios due to an imbalance in sub-samples. Figure 5.4 and Figure 5.5 shows the summary of the cross-validation. The mean and variance of the performance metrics for the Random Forest classifier were considerably better than the other models.

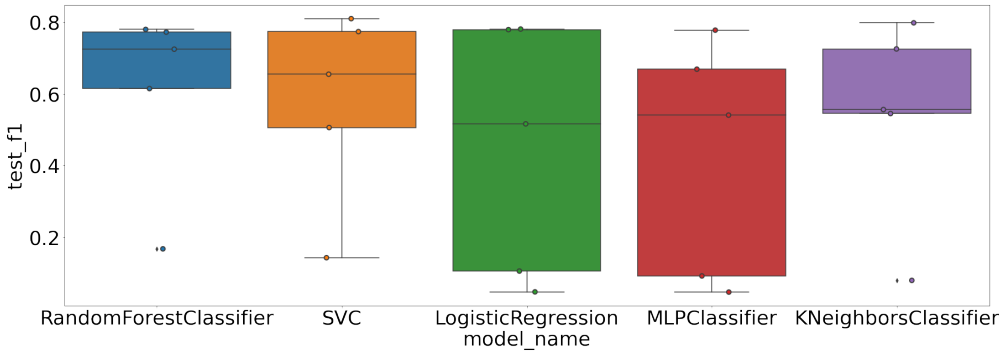


Figure 5.4: Classifier Selection by comparing f1-score

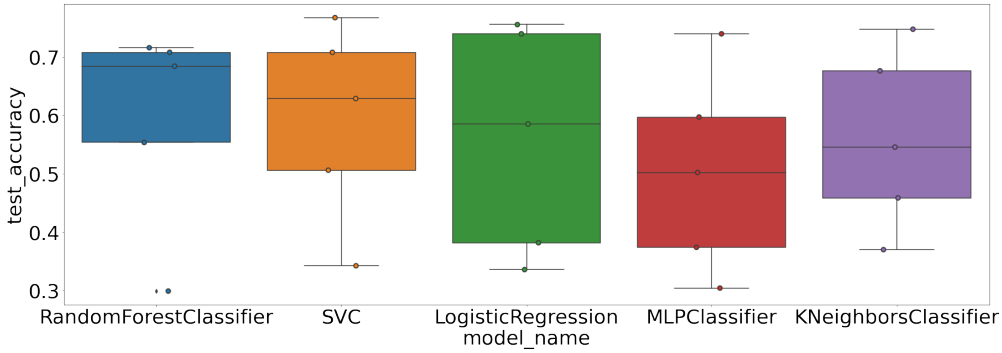


Figure 5.5: Classifier Selection by comparing accuracy

5.1.6 Dimensionality Reduction

Dimensionality reduction methods, such as PCA, have frequently been used in machine learning when a machine learning algorithm might be impeded by a correlation between features or an excessive number of features in the dataset. Dimensionality reduction can eliminate correlated attributes from a dataset, allowing for the machine learning algorithm to consider each uncorrelated characteristic of a software [123]. The large number of tokens extracted can also lead to overfitting [69]. While the preliminary evaluations conducted did not require dimensionality reduction, this step will need to be added if the dataset is to be extended.

5.2 Approach 2

5.2.1 Detecting Design Patterns

In this section, we discuss key concepts for automatically generating queries from UML Diagrams.

Identify Pattern Characteristics

Design pattern characteristics can be extracted from both UML Class and Sequence Diagrams. A Class diagram shows the objects within a design pattern and static relationships between those objects, while Sequence diagrams show interactions between objects. We discuss these diagrams below.

As a UML Class diagram contains the structure of a design pattern, including and relationships between objects, we can check these characteristics for any given pattern. Table 5.4 shows shows relationships between classes ("Class Relationships"), between classes and methods (e.g., hasMethod, hasConstructor in "Properties") and between methods and parameters (e.g., hasParameter, hasReturnType in "Properties"). For some design patterns, it is important to specify visibility or property (e.g., Singleton pattern). After resolving structural relationships, constraints about the data type of attributes, whether or not a method is

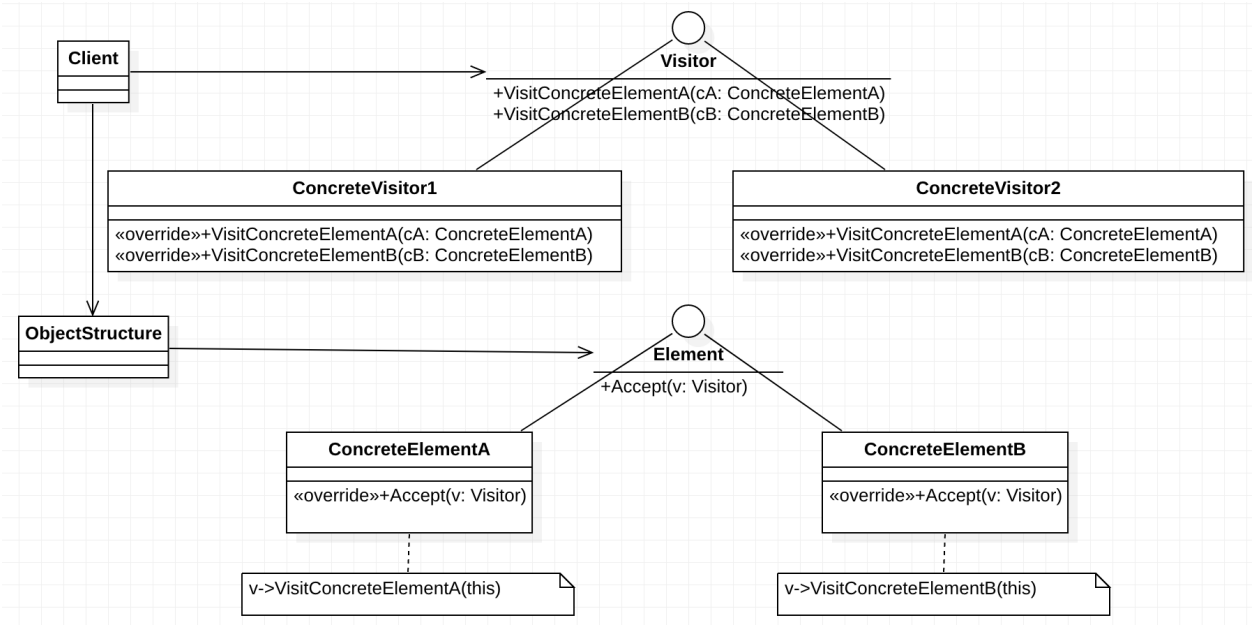


Figure 5.6: UML Representation of the Visitor Pattern

parameterized, the data type of the method parameter, the return type of the method are required to reduce false positives. For example, in a Visitor pattern, the accept method of a concrete element is the parameterized method whose parameter has Visitor datatype. In addition, by checking class attributes and method parameters, we can be more specific on the structure of the pattern.

We generate a SPARQL query by including relevant entities (i.e., object-oriented entities or "OO Entities" in Table 5.4) of the design pattern in the SELECT clause. We then add characteristics in the WHERE clause. Here's an example. A project that contains a visitor pattern may contain the following RDF triples in its RDF graph:

```
1 PREFIX woc: <http://rdf.webofcode.org/woc/>
2 woc:SoldierVisitor woc:implements woc:UnitVisitor .
3 woc:SoldierVisitor woc:hasMethod woc:SoldierVisitor-vistSoldier() .
4 woc:SoldierVisitor-visitComander woc:hasParameter
5 woc:visitComander(com.iluwater.visitor.Commander)-parameter-0 .
6 woc:SoldierVisitor-visitCommander(com.iluwater.visitor.Commander)-parameter-0 woc:hasType woc:Commander  .
```

These characteristics are captured in a UML Class diagram of a visitor pattern as shown in Figure 5.6. To generate a SPARQL query, we take all the entities in the class diagram, such as class names and method names, and add them to the SELECT clause, as shown below. Appended to these names are automatically generated identification, which is explained in Section 5.2.1.

```
1
2 PREFIX woc: <http://rdf.webofcode.org/woc/>
3 SELECT ?Visitor27 ?VisitConcreteElementA11 ?VisitConcreteElementB13 ?VisitConcreteElementA27
4 ?VisitConcreteElementB29 ?Accept13  ?Accept16 ?Accept20 ?VisitConcreteElementA24
5 ?VisitConcreteElementB26 ?ConcreteVisitor15 ?ConcreteVisitor211  ?Element14 ?ConcreteElementA18
6 ?ConcreteElementB22
```

Next, we add to the WHERE clause the structure of the pattern, such as the aforementioned relationships between entities. If included in the class diagram, characteristics related to visibility, property, data types, and return types are also generated. A partial list of characteristics for the above Class Diagram that would be included in a WHERE clause is below. Each line is a condition. The first line is a condition that states that Visitor27 is an Interface. All conditions in a WHERE clause must be satisfied for a design pattern match to occur. The more conditions, the more specific and restrictive. The fewer conditions, the more allowance for variation in implementation.

```
1
2 ?Visitor27  a woc:Interface .
3 ?ConcreteVisitor211  a woc:Class .
4 ?ConcreteVisitor211 woc:implements ?Visitor27 .
5 ?ConcreteVisitor211 woc:hasMethod ?VisitConcreteElementA11 .
6 ?VisitConcreteElementA11 woc:hasParameter ?cA10 .
7 ?cA10 woc:hasType ?ConcreteElementA18 .
```

For some design patterns, such as the Visitor pattern, we need behavioral information to accurately identify them. Behavioral specifications related to method invocation can be obtained from Sequence diagrams. Here is an example that illustrates the importance of behavioral information for identifying design patterns. The following snippet from [8] shows an RDF triple with a behavioral characteristic of Visitor design pattern pattern.

```
1
2 <http://rdf.webofcode.org/woc/com.iluwatar.visitor.Sergeant-
3 accept(com.iluwatar.visitor.UnitVisitor)>
4 <http://rdf.webofcode.org/woc/references>
5 <http://rdf.webofcode.org/woc/com.iluwatar.visitor.UnitVisitor-
6 visitSergeant(com.iluwatar.visitor.Sergeant)> .
```

A SPARQL query based only on the Class diagram of a Visitor pattern will include the following in the WHERE statement, which represents an association relationship:

```
1 ?ConcreteElementA18    woc:references   ?Visitor27   .
```

Since the RDF graph representation of the code lacks this triple, the query would have returned with a false negative result for Visitor pattern. However, by including information from the Sequence diagram shown in Figure 5.7, the WHERE statement would include a call to Accept method in VisitConcreteElement, as follows:

```
1 ?Accept16    woc:references  ?VisitConcreteElementA24  .
```

This query would return a true positive result for Visitor pattern.

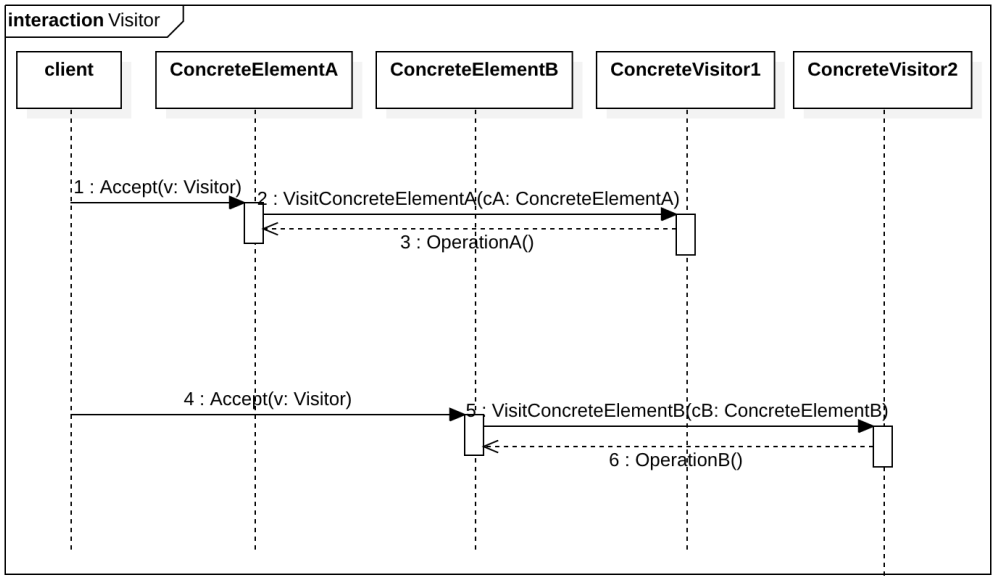


Figure 5.7: Sequence Diagram for Visitor Design Pattern

Use Stereotypes

Another way to improve the accuracy of design pattern identification is to use stereotypes. Although not part of the standard UML specification, numerous stereotypes have been used to differentiate or represent features like Constructors, Getters, Setters, and Overriding of methods (see Figure 5.8). When stereotypes are used, Constructors can be differentiated from other Methods as the SPARQL query can use woc:Constructor instead of $woc : Method$ entity and $woc : hasConstructor$ instead of $woc : hasMethod$ relationship. Similarly, methods of a child class that overrides methods of a parent class are differentiated with $woc : overrides$ stereotype. This was observed to significantly reduce false-positive results. This can be a powerful feature in improving the accuracy of detection of patterns like Proxy, Builder, and Singleton.

For example, in the Visitor pattern example, the only constraint on identifying the visit methods is that the parameter should have the data type of the ConcreteElement. Similarly, the parameter of the Accept method is a visitor. If there are other methods in these classes that satisfy this constraint, it can lead to a large number of false-positive results. However, by using the stereotype $<< override >>$ for override annotation the following condition can be included in the WHERE clause to reduce pattern matches to the methods that override a method from the superclass:

```
1 ?VisitConcreteElementA11 woc:hasAnnotation java.lang.Override.
```

Similarly, in patterns that specify behavior related to Constructors (e.g., Singleton, Builder), differentiating Constructor from other methods is only possible by using $<< constructor >>$ stereotype By including this stereotype, $woc : Constructor$ and $woc : hasConstructor$ relationship can be used instead of $woc : Method$ and $woc : hasMethod$ respectively. An example is given below:

```
1 ?proxyConstructor a woc:Constructor .
2 ?Proxy woc:hasConstructor ?proxyConstructor .
```

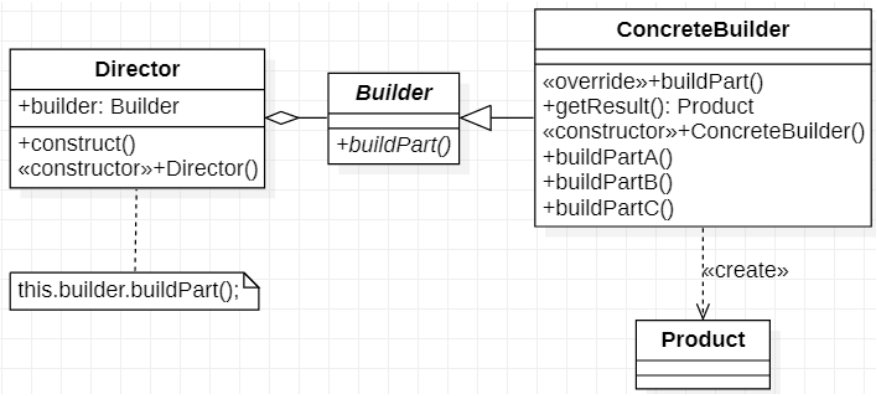


Figure 5.8: UML Diagram of Builder Pattern with stereotypes

OO Entities	**Contain Relationships**	**Visibility/ Property**	**Class Relationships**	**Stereotypes**	**Interactions**
Classes	hasMethod	Public	Association	Constructor	Method-
Methods	hasType	Private	Generalization	Override	Invocations
Constructors	hasReturnType	Protected	Aggregation		
Fields	hasModifiers	Static	Composition		
Method	hasField	Final	Interface-		
Parameters	hasParameter	Synchronized	Realization		
Interfaces	hasConstructor	Abstract	Dependency		

Table 5.4: Supported UML Concepts by PatternScout

Uniquely Identify OO Entities

When parsing a UML diagram to generate a query, the parser is required to make OO Entity names in the query unique. For example, in a Visitor DP, both the interface Visitor and each of its implementations will have visit methods of the same name. Although the names are the same, the Entities have independent existence and the RDF triples need to be distinguished. In the following snippet, both Visitor and ConcreteVisitor have a VisitElementA

method where ConcreteVisitor's VisitElementA method overrides Visitor's method during implementation. However, this snippet assumes both VisitElementA methods to be the same.

```
?Visitor woc:hasMethod ?VisitElementA .
?ConcreteVisitor woc:hasMethod ?VisitElementA .
```

The correct way of representing this scenario is

```
Visitor woc:hasMethod VisitElementA1 .
ConcreteVisitor woc:hasMethod VisitElementA2 .
VisitElementA2 woc:overrides VisitElementA1 .
```

An approach to make this possible would be to use uniquely generated identifiers for each component. However, when analyzing results, randomly generated identifiers would be difficult to interpret. Hence, a Running ID is appended to each Entity's name. Running ID is a sequence number that is auto-generated when the XMI is parsed using SDMetrics library [136].

For patterns that have multiple OO entities of the same type, the query might assume the same Entity to be suitable for both items in the SELECT statement. For example, the SPARQL query given below is intended for a class with two methods. The following triples, however, will also identify a class that has only one method by substituting the same method for both MethodA and MethodB.

```
ClassA woc:hasMethod MethodA .
ClassA woc:hasMethod MethodB .
```

There are two ways of avoiding this scenario:

1. Using SELECT DISTINCT keyword

```
SELECT DISTINCT ?ClassA ?MethodA ?MethodB
```

2. or Add FILTER criteria to WHERE clause

```
FILTER(MethodA != MethodB)
```

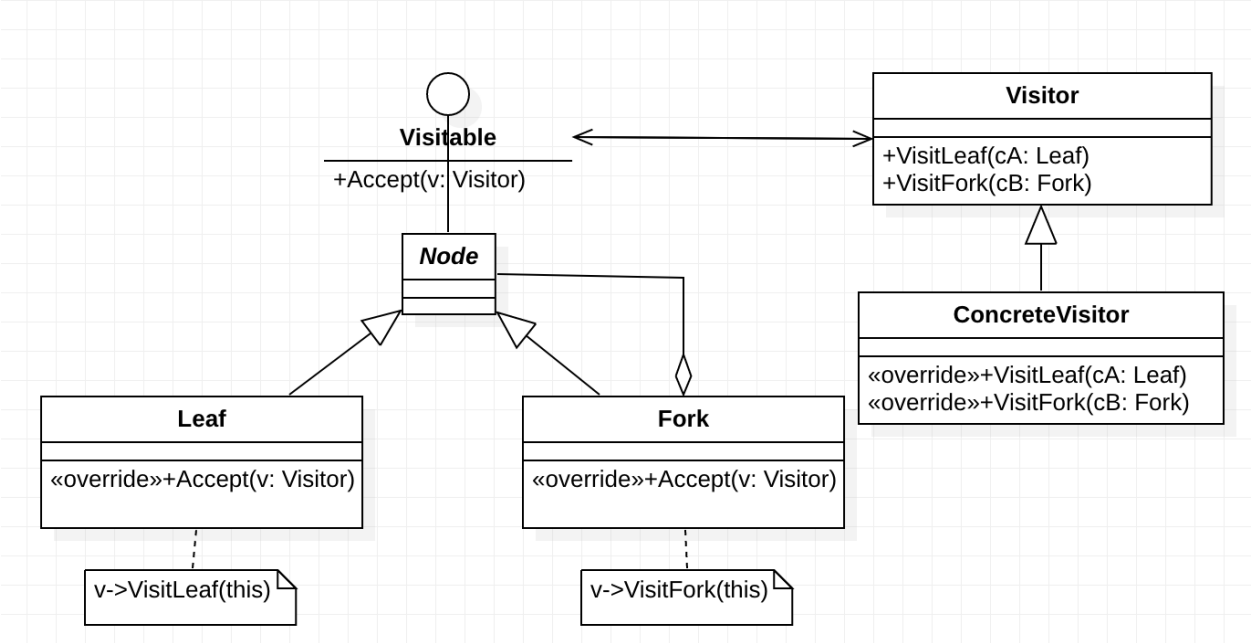


Figure 5.9: Visitor Combinator Pattern

Match Graph instead of Text

Most design pattern implementations in software projects follow naming conventions. That is, the Visit method in Visitor classes will have a name that starts with the word Visit, the accept method in the ConcreteElement classes usually have the name Accept.

However, inconsistencies in developer implementation make this check unreliable. Hence, using text-matching based on naming conventions was excluded in SPARQL queries generated by PatternScout. Instead, we rely on matching the graph pattern obtained from Class and Sequence diagrams.

Accommodate Design Pattern Variants

Design patterns not only have many implementation variations, but some variations are combinations of existing patterns. Figure 5.9 shows a variant of the Visitor pattern where the GoF specification of Visitor is combined with Composite pattern for object-oriented

tree traversal. Pattern Scout can accommodate these variations, as long as they can be represented as a Class diagram. A SPARQL query is automatically generated from the variant Class diagram and then added to a query map, which maps the variant to the base design pattern (see Section 5.2.2). Once the mapping is performed, we can run check all the variants of a design pattern on a given source code.

5.2.2 Tool Support

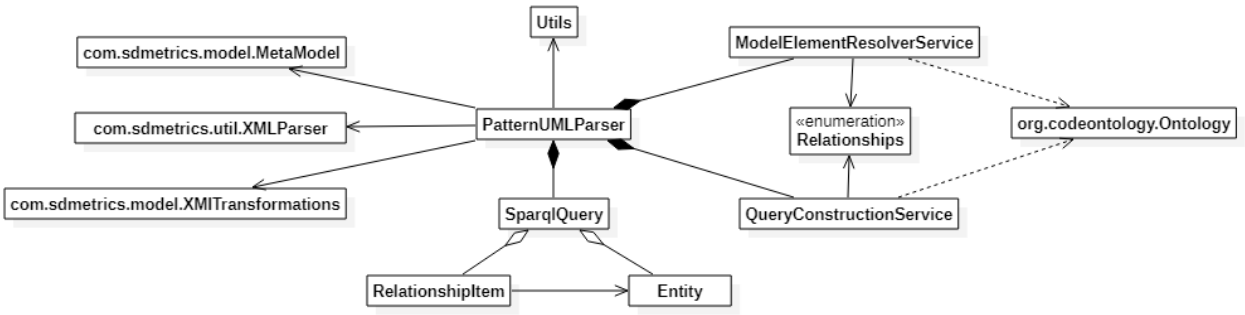


Figure 5.10: Object-oriented Design of PatternScout

This section provides details on generating SPARQL queries from UML diagrams (in XMI format). An OO design of PatternScout is as shown in Figure 5.10. The flow diagram summarizing the steps involved in the query generation process is shown in Figure 5.11. Objects contain query information and modules create our query. PatternScout can be extended to support more relationship and entity types with minimal changes.

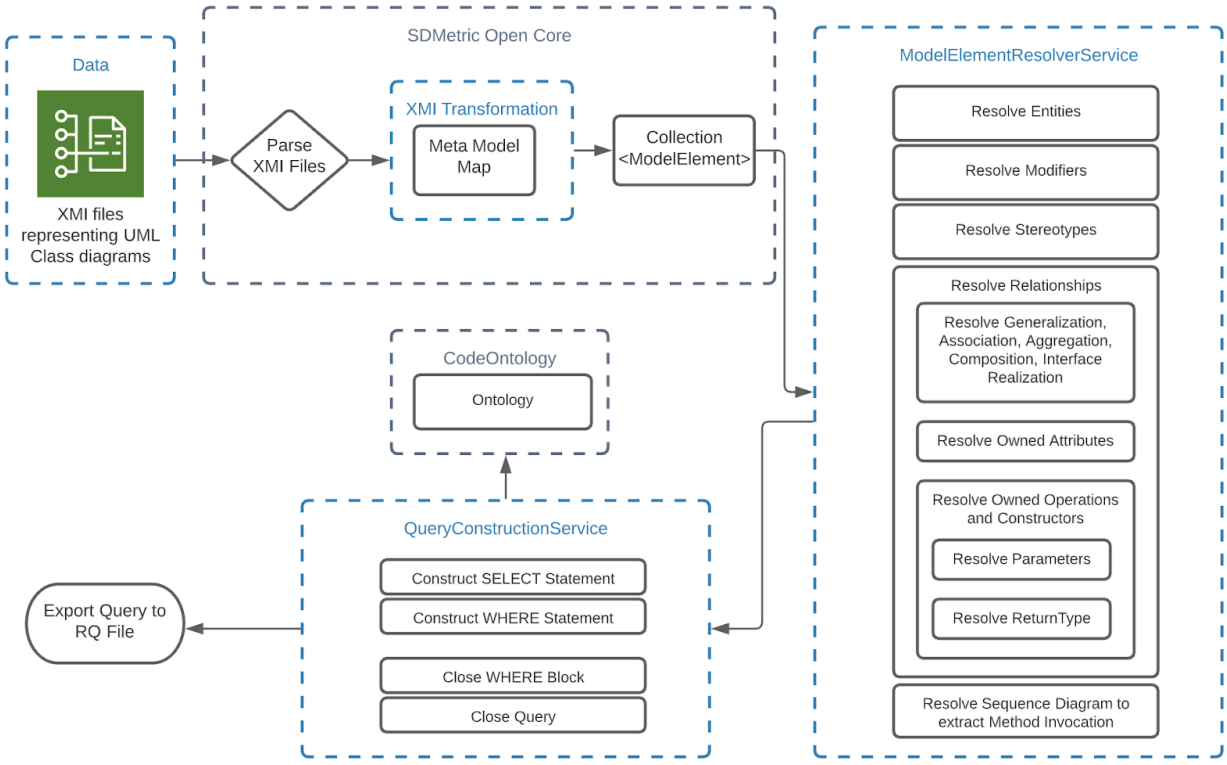


Figure 5.11: Diagram showing the Steps for Automatically Generating SPARQL Queries and the pipe and filter architecture

The architecture follows a modular design with a separation of concerns. For instance, ModelElementResolverService handles the task of identifying components and relationships. On the other hand, QueryConstructionService handles the task of constructing the query from identified Components and RelationshipItems. The two services are completely decoupled. We designed PatternScout by keeping the reusability and portability of the individual modules in mind.

The modularity also assists in enhancing testability. Test cases and Unit Test classes for automated testing for these features can be implemented independently. Similarly, the tasks of parsing XML and exporting output to a specified file format are also handled by dedicated methods. To avoid redundancy, all the features that can be reused from the open-

source libraries SDMetric Open Core and CodeOntology were utilized, extending only to accommodate additional features not supported by their native code.

The architecture can be considered similar to the pipe and filter style. Once the model is parsed, the tool processes the model sequentially through resolving components, constructing select statements, constructing were statements, and constructing filters. Within the construction of queries, there is an internal sequential check for whether to suppress visibility constraints and whether or not to incorporate stereotypes.

Objects

Entity: Entities are the nodes in a UML diagram between which relationships exist. Entities are OO Entities in Table 5.4.

RelationshipItem: A relationshipItem contains a $fromItem$, a $toItem$, and a $relationshipType$. Both $fromItem$ and $toItem$ are of Entity type. An RDF triple can be constructed as $(fromItem, relationshipType, toItem)$. Relationship types include relationships found between classes (Class Relationships in Table 5.4) and hierarchical relationships (Contain Relationships in Table 5.4). An example hierarchical relationship is between a method and its parameters (woc:hasReturnType, woc:hasParameter).

SparqlQuery: The SparqlQuery object represents the generated query. It contains a list of entities used to construct the SELECT statement, a list of relationship items used in the WHERE clause, and a query string.

Modules

PatternUMLParser: This is the main module that parses an XMI file and generates queries. A Java Object Model is created using the SDMetrics Open Core library [136] from an XMI file. Once we have the object model, we iteratively analyze each entity and relationship, converting them into Entity and RelationshipItem objects. Next, we create a query string with two parts: a SELECT statement and a WHERE clause. Relationships (both Contain

and Class Relationships in Table 5.4) as well as Visibility/Property and Stereotypes (see Table 5.4) are added to the WHERE clause.

ModelElementResolverService: This module is responsible for determining whether an element in the object model is relevant for constructing a query and the format for the query. If an element is relevant, then it is added to the list of Entity or RelationshipItems in the query object. For example, UML diagrams contain Roles to explain how an object participates in a relationship. However, the RDF triples created for the source code do not contain triples related to Roles. Hence ModelElementResolverService will skip any object that has type Role when parsing the diagram. Additionally, this module resolves all types of relationships.

QueryConstructionService: Once all elements are checked, this module constructs the query. Each entity is added to the SELECT statement. Each entity is also added to the WHERE clause defining its type. For example, if a Method is encountered, woc:Method type is added for that component in the WHERE clause:

```
1 SELECT ?ClassA ?OperationA
2 WHERE {
3 ?ClassA  a woc:Class .
4 ?OperationA  a woc:Method .
```

After adding entities to the query, we then add relationships to the WHERE clause. A triple pattern is appended to the query based on the relation of $fromItem$ to the $toItem$. In the following snippet, ClassA is the $fromItem$, OperationA is the $toItem$ and woc:hasMethod is the $relationshipType$. Once all relationships are checked, the WHERE clause is closed using a closing bracket "}".

```
1 SELECT ?ClassA ?OperationA
2 WHERE {
3 ?ClassA  a woc:Class .
4 ?OperationA  a woc:Method .
5 ?ClassA woc:hasMethod ?OperationA .
6 }
```

Runtime Settings

PatternScout exposes options whereby settings like suppressing visibility constraints and parsing stereotypes can be configured on a case-by-case basis. When visibility constraints are suppressed, constraint triples that state woc:hasModifier Public/Protected/Private are not generated. Similarly, the model looks for stereotypes only if the configuration is set to true.

Variant Query Map

We use a query map to efficiently connect variants with each pattern. The map is exposed so that users can extend it as needed, by adding the file to the map. Below is a snippet of a query map. The first line shows the path to the query. Each design pattern has its own section (e.g., Strategy in line 3). Below each design pattern is a list of variants, with the variant name and filename.

```
1  "stdir" : "/SPARQLS/queries/",
2  "queries" : {
3  "Strategy": {
4      "Strategy": "strategy.rq",
5      "Flexible Strategy Pattern":"strategy_flexible_form.rq"
6  },
7  "Template Method": {
8      "Template Method": "TemplateMethod.rq",
9      "TM { Factory Method Compound": "TM_factory_compound.rq",
10     "Enhanced Template Method": "EnhancedTemplateMethod.rq"
11 },
12 "Visitor": {
13     "Visitor GoF": "visitor.rq",
14     "Visitor Combinators": "visitor_combinators.rq",
15     "Extended Visitor Pattern": "extended_visitor.rq"
16 },
17 "Singleton" : {
18     "Singleton GoF" : "singleton.rq",
19     "Eager Instantiation": "singleton_Eager_Instantiation.rq",
20     "Lazy Instantiation (non-thread safe)": "singleton_LI_nts.rq",
1
```

5.2.3 Steps for Generating Queries

The algorithm we use to generate the SPARQL queries is summarized in Algorithm 1.

Algorithm 1 Algorithm used by PatternScout to generate SPARQL queries from UML Diagrams

```
1: Parse XMI file
2: Convert XMI to Collection of ModelElement using XMI_Transformation and
   MetaModel map
3: for ModelElement in Collection<ModelElement> do
4:    Resolve Entities
5:    for Entity in Collection<Entities> do
6:       Resolve Modifiers
7:       Resolve Stereotypes
8:    end for
9:    Resolve Relationships
10:   if ModelElement.type == Class OR ModelElement.type == Interface then
11:      Resolve Generalization and Interface Realization
12:      Resolve Association, Aggregation and Composition
13:      Resolve Owned Attributes
14:      Resolve Owned Operations
15:      for each operation do
16:         Check if Constructor
17:         Resolve Parameters
18:         Resolve ReturnType
```

```
19:     end for
20:
21:     Resolve Sequence Diagram to extract Method Invocation
22:     Construct Query
23:     Construct SELECT Statement
24:     for ModelElement in Collection<ModelElement> do
25:        query += ModelElement.name
26:     end for
27:     Construct WHERE Statement
28:     for rItem in Collection<RelationshipItem> do
29:        query += (rItem.from, rItem.relationship, rItem.to)
30:     end for
31:     Close WHERE Block
32:     Close QUERY
33:     Export query to RQ File
```

Chapter 6

RESULTS

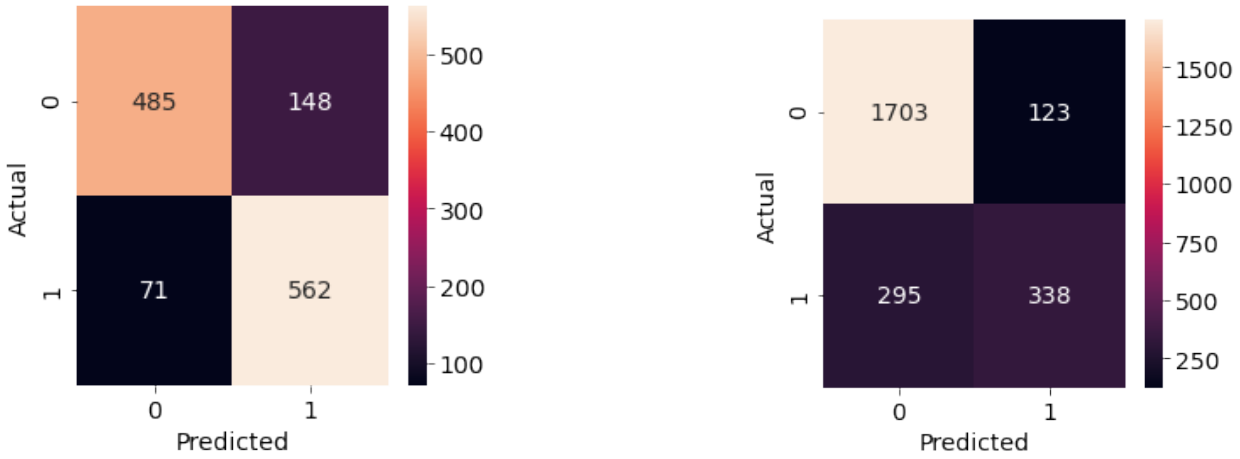
6.1 Approach 1

6.1.1 Model Evalutation

The selected classifier model was evaluated for performance on both a balanced dataset (with an equal number of samples with labels 1 and 0) and a stratified dataset (with the ratio of samples with labels 1 and 0 matching original data). The selected classifier, in this case, is the Random Forest classifier with hyperparameters as follows:

```
{
'bootstrap': True, 'class_weight': 'balanced_subsample', 'max_depth': 10,
'max_features': 2, 'min_samples_leaf': 3, 'min_samples_split': 4, 'n_estimators': 100
}
```

We were able to verify that the class imbalance strategies implemented were successful in helping the model to not label all samples to be of class 0 even though the original data was imbalanced. The confusion matrix and performance metrics of model evaluation on balanced data is summarized in Figure 6.1 and Table 6.1 respectively. The confusion matrix and performance metrics of model evaluation on stratified data is summarized in Figure 6.1b and Table 6.2. The average accuracy of model evaluation on a stratified dataset was 82% with a macro average precision of 79%, macro average recall of 73%, and macro average f1-score of 0.75. This is a considerable improvement compared to the performance before implementing class imbalance strategies (macro average precision of 79%, macro average recall of 59%, and macro average f1-score of 0.62).



(a) Balanced Dataset

(b) Stratified Dataset

Figure 6.1: Model Evaluation in predicting CWE-79

	precision	recall	f1-score	support
1	0.79	0.89	0.84	633
0	0.87	0.77	0.82	633
Accuracy			0.83	1266
Macro Average	0.83	0.83	0.83	1266
Weighted Average	0.83	0.83	0.83	1266

Table 6.1: Performance Metrics in predicting CWE-79 on a balanced dataset

	precision	recall	f1-score	support
0	0.85	0.93	0.89	1826
1	0.73	0.53	0.62	633
Accuracy			0.83	2459
Macro Average	0.79	0.73	0.75	2459
Weighted Average	0.82	0.83	0.82	2459

Table 6.2: Performance Metrics in predicting CWE-79 on a Stratified dataset

6.1.2 Correlation Analysis

A correlation analysis was conducted to compute the pairwise correlation of columns (CWE IDs) to analyze the co-occurrence of different vulnerabilities. The results obtained upon executing the analysis over 209000 rows are summarized in Table 6.3. Since we calculate the correlation using samples in the dataset where a vulnerability is reported (that is, labeled 1 corresponding to the CWE ID), the number of samples used to calculate the correlation between each vulnerability pair is different. Although the correlation calculated for the most commonly occurring vulnerability pairs have a statistical significance of $p < 0.001$, many less frequent vulnerability pairs have a statistical significance of $p < 0.01$ or $p < 0.1$.

Correlation analysis or PCA analysis can also help in dimensionality reduction. That is, if two CWE IDs are highly correlated, separate binary classification for the two labels need not be conducted. Moreover, the feature set (keywords extracted from the documents) can be merged to create a more fine-tuned classifier. In this evaluation, we were not able to identify any highly correlated vulnerability pairs where the co-occurrence was large enough for the dimensionality reduction to improve the performance of KEVIS. This analysis may however be helpful in case of resource constraints or for a multi-class classification model where subsetting labels are desirable.

The analysis implies that when there is a vulnerable component present in the system (for example, a software component, weak code snippets, vulnerable API endpoints, or dependencies on libraries with security issues) if a vulnerability is detected, it is worth looking at other possible weaknesses triggered due to the same component. The findings of our correlation analysis can be used to proactively add measures to mitigate such potential vulnerabilities. For example, if CWE-319 (Cleartext Transmission of Sensitive Information) is reported, security engineers can pro-actively add measures to counter CWE-307 (Improper Restriction of Excessive Authentication Attempts) as well, as it is very likely to be detected in the near future.

Vulnerability 1 (CWE ID)	**Description**	**Vulnerability 2 (CWE ID)**	**Description**	**Correlation**
415	Double Free	129	Improper Validation of Array Index	0.4511***
787	Out-of-bounds Write	125	Out-of-bounds Read	0.4529***
384	Session Fixation	326	Inadequate Encryption Strength	0.4705***
326	Inadequate Encryption Strength	319	Cleartext Transmission of Sensitive Information	0.4997***
476	NULL Pointer Dereference	125	Out-of-bounds Read	0.5151***
384	Session Fixation	319	Cleartext Transmission of Sensitive Information	0.5582***
129	Improper Validation of Array Index	120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	0.5741***
384	Session Fixation	307	Improper Restriction of Excessive Authentication Attempts	0.6158***
326	Inadequate Encryption Strength	307	Improper Restriction of Excessive Authentication Attempts	0.6379***
319	Cleartext Transmission of Sensitive Information	307	Improper Restriction of Excessive Authentication Attempts	0.6868***

Note: *** indicates p-value < 0.001

Table 6.3: Most Correlated Vulnerability Pairs

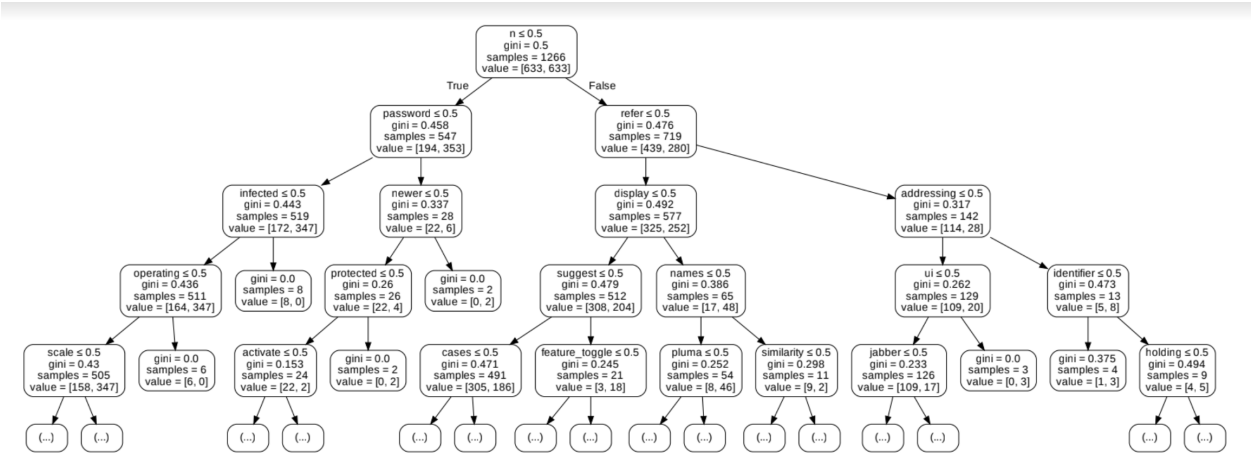


Figure 6.2: First five levels of the decision tree model

6.1.3 White-box Models

Most machine learning systems require the ability to explain to stakeholders why certain predictions are made. When choosing a suitable machine learning model, researchers face an accuracy vs. interpretability trade-off. For instance, Black-box models such as neural networks and complicated ensembles often provide great accuracy. However, they don't provide an estimate of the importance of each feature on the model predictions or any insight into how the different features interact. Simpler models such as linear regression and decision trees on the other hand provide less predictive capacity and are not always capable of modeling the inherent complexity of the dataset. They are however significantly easier to explain, interpret and visualize.

A Decision Tree classifier was used to gain insights into how certain features contribute to the prediction decisions. The first 5 levels of the decision tree model are shown in Figure 6.2. The maximum accuracy of the model was 82% which is less than the mean accuracy of the random forest classifier (84.4%) on the stratified dataset as the random forest classifier is an ensemble method that uses decision trees within it. Figure 6.3 shows the segment of the decision tree classifier that analyzes keywords extracted related to operating systems,

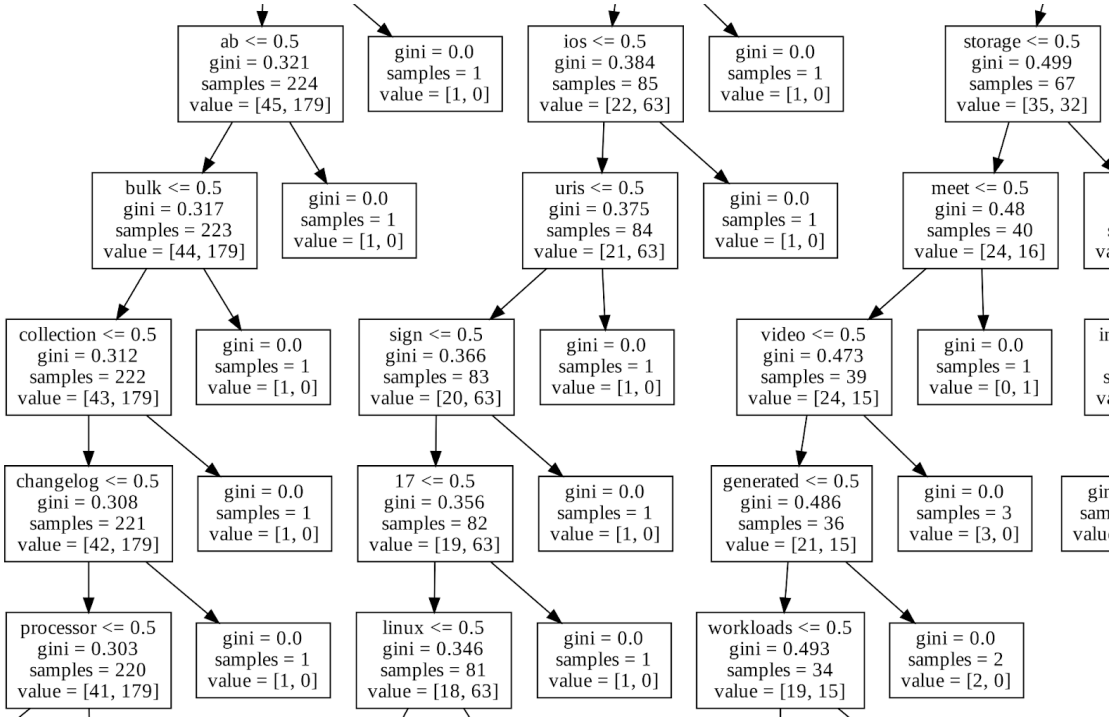


Figure 6.3: Partial view of the decision tree showing keywords related to operating systems, processing and storage in predicting CWE-79

storage, and processing units in how they contribute to predicting CWE-79.

According to our preliminary evaluation, black-box models seem to outperform white-box alternatives in the prediction. However, white-box models can be used to gain insights into which keywords are contributing to making key decisions about each vulnerability label. Hence, a combination of white-box and black-box models can be used so that not only are the findings more explainable to stakeholders but also the accuracy of the prediction is not compromised.

6.1.4 Semi-supervised learning

Semi-supervised learning is an approach to machine learning that combines a small amount of labeled data with a large amount of unlabeled data during training. Semi-supervised

learning falls between unsupervised learning (with no labeled training data) and supervised learning (with only labeled training data)[101]. We tried improving the performance of the vulnerability prediction using the semi-supervised approach by combining unsupervised clustering with supervised binary classification. That is, clustering was used to identify clusters of products using the keywords extracted. The label of the centroid of each cluster was used to train the model. During testing, the label predicted for the centroid of the cluster was assigned for all the products within the cluster. In the preliminary evaluation, it was observed that, for a stratified dataset with only 3602 products, this approach could only achieve maximum accuracy of 77% and average accuracy of 74.9%. Whereas, Random Forest Classifier without semi-supervised learning could achieve a mean accuracy of 84.4%. This is because using the centroid as the proxy for all products in the cluster reduces the resolution of the dataset. According to preliminary results, semi-supervised learning does not improve the accuracy of prediction. Further hyperparameter tuning and combination of different classifiers in the semi-supervised learning pipeline need to be explored to evaluate if semi-supervised learning can improve the predictive accuracy further.

6.1.5 Comparison with Related Tools

Since this is the first study that predicts vulnerabilities from specification document text, no direct comparison is possible for the precision, recall, accuracy, and f1-score measures discussed in the evaluations. Most other studies have looked at 3-5 products to try to predict vulnerabilities within them. For example, the performance of the deep learning-based vulnerability detection method VulDeePecker [84] that uses source code analysis was evaluated by checking if it can detect CWE 190 in Xen 4.6.0 [17], and CWE 119 in Seamonkey 2.31 [15] and Libav 10.2 [12]. To compare how our model performs on these products, the documentation of these 3 products were downloaded, text extracted, tokenized, and evaluated by training our model for both CWE 190 and CWE 119. We also verified to ensure that the products were not part of the training set. Our model was able to correctly predict the presence of CWE 119 in Seamonkey and Libav. The model labeled all three products as 0

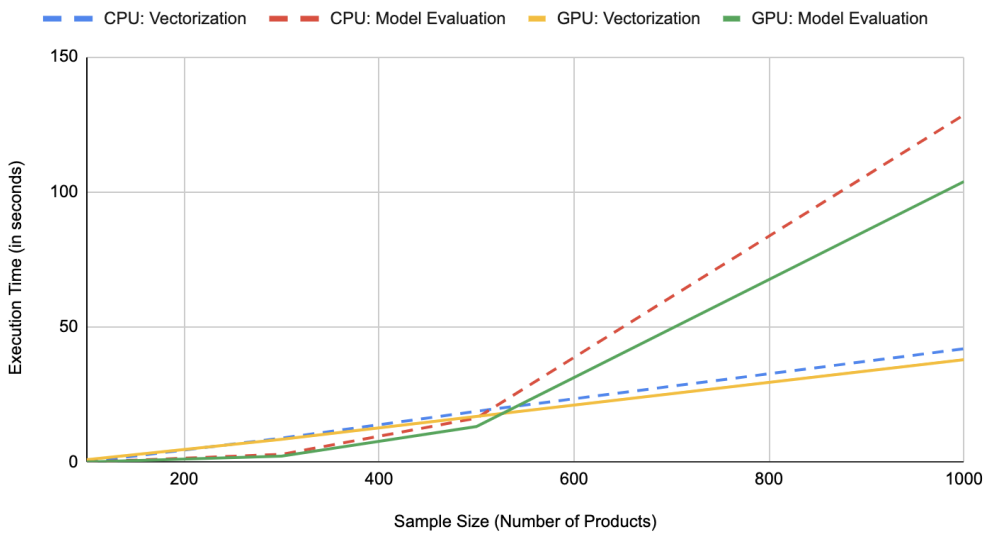


Figure 6.4: Comparison between the execution time observed when using a CPU vs GPU for executing Vectorization and Model Evaluation

for CWE 190 as the dataset (3602 entries) only contained 8 samples of CWE 190 reports. This shows that given a sufficient number of samples for the model to train on, the text classification of keywords extracted from specification documents can be used to reliably predict vulnerabilities that could potentially be present in the system.

6.1.6 Performance Measures

Due to the large size of input data and the large number of features involved in the computation, the computational complexity will increase drastically when the dataset is extended. This could create a bottleneck to train the model sufficiently well to create accurate predictions. To overcome this challenge, the model vectorization, benchmarking and model evaluation tasks were parallelized using RAPIDS CUML[4] library and CUDF [3]. The time taken to execute vectorization, benchmarking and model evaluation in sequential processing and parallel processing was compared using an Intel (R) Xeon (R) CPU @ 2.3 GHz with 26GB RAM processor mounted over a 105 GB file system and a GPU: Tesla P100, having

3584 CUDA cores, 16GB(16.28GB Usable) GDDR6 VRAM. The performance observed is summarized in Figure 6.4. The time taken by the parallel execution was consistently lower than that of sequential processing. While the sequential processing caused execution environment timeouts during benchmarking for larger sample sizes (more than 3000 products), the parallel execution was able to handle sample sizes with more than 3000 products by completing the execution in 3029.2657 seconds. The benchmarking execution time was not included in Figure 6.4 as it is a one-time task for classifier selection.

6.2 Approach 2

We conducted experiments to measure the accuracy and running time of generated queries. We selected nine design patterns to represent each category of design pattern (see Table 3.1). These patterns represent the different types of relationships obtained from a Class diagram. Class diagrams and Sequence diagrams were created for each variant, based on the literature [105]. Since some Sequence diagrams were not provided in the literature, we created these diagrams based on the description of the design pattern. SPARQL queries were automatically generated from these diagrams.

The only pre-processing or preparation required is generating a graph for each project using CodeOntology [25]. The preparation time taken for each project is shown in the Preparation Time column of Table 6.4. The number of RDF triples in the resultant RDF graph (in .nt format) is available in the RDF Triples column. Then, PatternScout was used to generate SPARQL queries for representatives of each type of object-oriented design pattern (see patterns in Table 3.1). Generated queries were run on the RDF graph of the projects.

6.2.1 Metrics

We used the following formula to measure the performance of our approach:

$$\text{Accuracy} = \frac{\text{True Positive} + \text{True Negative}}{\text{True Positive} + \text{True Negative} + \text{False Positive} + \text{False Negative}}$$

Open Source Project	**LOC**	**Java Classes**	**RDF triples**	**Preparation Time (ms)**
JHotDraw v5.1 (JHD)	8907	155	52824	2040
JRefactory v2.6.24 (JRF)	56187	569	70178	3023
JUnit v3.7 (JUN)	1347	33	9497	2001
QuickUML 2001 (QUM)	9250	156	59480	1756
MapperXML 1.9.7 (MPX)	14928	217	17147	6002
Dom4J v1.6.1 (DOM)	26350	328	29874	2059
Lizzy v1.1.1 (LZ)	12915	197	11617	1083

Table 6.4: Projects used for evaluation

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive + False Positive}}$$

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive + False Negative}}$$

$$\text{f1-score} = \frac{\text{2.Precision.Recall}}{\text{Precision + Recall}}$$

To evaluate if the SPARQL queries generated by PatternScout are accurate and sufficient for detecting design patterns, we conducted two analyses.

Analysis 1

A preliminary evaluation was conducted on how PatternScout can correctly label projects based on the presence/absence of patterns.

Gold Standard: Since results have been reported for presence or absence of design patterns in the projects we selected, we used these results as our gold standard [137] [87] [142] [125] [23] [20]. If a pattern is reported as present in a project (regardless of variation,

as previous studies did not specify variant used), we marked this as a check (or True) for the actual label of the project in Table 6.5. If it is reported as absent, we gave it an X-mark (or False).

Experiment: We used the query map (see section 5.2.2) to specify all the different variations of a design pattern to run on a project's RDF graph (in .nt format). If at least one of the variants of a pattern fetches results, the specific pattern is marked as detected (or true). If none of the variants fetched any results, the pattern is marked undetected (or false).

Calculation: True Positives (TP) is the number of patterns that were marked as detected in a project that was implemented in the project. False Positives (FP) is the number of patterns that were marked as detected which were not present in the projects. Similarly, False Negatives (FN) were the patterns that were present in the projects marked as undetected by PatternScout.

Result: Precision and Recall in labeling each project is summarized in the P and R columns of Table 6.5. PatternScout was able to label the 7 projects with an average precision of 0.91 and an average recall of 0.77.

	PRTT	SGLT	STT-STTG	FM	OBSV	CMD-ADPT	TPLT	P	R
JHD	✓| ✓	✓| ✓	✓| ✓	✓| ×	✓| ✓	✓| ✓	✓| ✓	1	0.86
JRF	×| ×	✓| ✓	✓| ×	✓| ✓	✓| ✓	×| ×	✓| ✓	1	0.8
QUM	✓| ✓	✓| ✓	✓| ✓	×| ×	✓| ✓	✓| ✓	✓| ✓	1	1
MPX	×| ×	✓| ✓	✓| ✓	✓| ×	✓| ✓	×| ✓	✓| ✓	0.8	0.8
DOM	✓| ×	✓| ✓	✓| ×	✓| ×	✓| ×	×| ×	✓| ✓	1	0.33
LZ	×| ✓	✓| ✓	✓| ✓	✓| ×	×| ✓	✓| ✓	✓| ✓	0.67	0.8
							Average	0.91	0.77

Table 6.5: Precision and Recall calculated for labeling projects based on presence of Design Patterns (Tuple represents Actual Label | Predicted Label)

Analysis 2

The secondary evaluation was conducted to analyze individual results retrieved by the SPARQL query. Table 6.6 shows the sum of the number of sub-patterns detected by all the variants of each pattern. The number is largely due to duplication of the same instance in the result due to permutations of entities that match the query as explained in Section 7.2.4.

	PRTT	**SGLT**	**STT-STTG**	**FM**	**OBSV**	**CMD-ADPT**	**TPLT**
JHD	126097	4	136826	0	14830	21256	346
JRF	0	32	0	36975	16383	0	983
JUN	0	0	0	0	0	0	0
QUM	4739	1	12705	0	20412	3372	47
MPX	0	2	1650	0	512	340	1
DOM	0	30	0	0	0	0	503
LZ	20800	2	22048	0	2880	1908830	563

Table 6.6: Number of sub-patterns recovered from each project

Experiment: The number of unique instances (summarized in Table 6.7) was obtained by filtering out the unique entities identified by the queries. For example, the number of unique Visitor pattern instances was obtained by counting the unique Visitor interfaces detected as the permutations caused by Concrete Visitor classes, Concrete Element classes and Visit methods are all part of the same pattern instance. The unique instances retrieved by all the variants are combined to arrive at the unique instances of the pattern.

		TP	FP	FN	P	R
PRTT	JHD	3	10	0	0.23	1
	JRF	-	-	-	-	-
	QUM	2	8	0	0.2	1
STT-STTG	JHD	14	0	1	1	0.93
	JRF	-	-	-	-	-
	QUM	2	7	0	0.22	1
FM	JHD	-	-	-	-	-
	JRF	8	1	0	0.89	1
	QUM	-	-	-	-	-
OBSV	JHD	8	1	0	0.89	1
	JRF	3	0	0	1	1
	QUM	3	5	0	0.38	1
ADPT-CMD	JHD	13	4	0	0.76	1
	JRF	0	0	0	-	-
	QUM	6	11	0	0.35	1
TPLT	JHD	2	7	0	0.22	1
	JRF	5	24	0	0.17	1
	QUM	1	4	0	0.2	1

Table 6.7: Analysis of unique design pattern instances retrieved by PatternScout ("-" means pattern does not apply)

Calculation: True Positives (TP) is the number of instances detected by the queries that were actual pattern implementation. False Positives (FP) is the number of instances detected that were not part of an actual pattern implementation. Similarly, False Negatives (FN) were the pattern instances that were present in the projects but undetected by PatternScout. The number of actual instances reported is inconsistent across different studies due to the differences in their methods [87] [29], with some using sub-patterns [143], sub-graphs [63], design motifs and micro-architectures [23]. The ground truth for this evaluation was

established using both manual validations and from the evaluations of [87] [125].

			PatternScout		**ePAD**[87]		**DPD**[125]		**RM**[106]		**DPF**[29]	
PRTT	**P**	**R**	0.22	1	1	1	0.57	0.75	1	0.67	0.58	0.58
	f1		0.36		1		0.65		0.8		0.58	
STT-STTG	**P**	**R**	0.61	0.97	0.59	0.67	0.1	0.51	0.38	0.26	0.17	0.69
	f1		0.75		0.63		0.17		0.31		0.27	
FM	**P**	**R**	0.89	1	0.88	0.6	0.25	0.08	0.33	0.17	0.14	0.52
	f1		0.94		0.71		0.12		0.22		0.22	
OBSV	**P**	**R**	0.75	1	0.93	0.96	0.67	0.23	1	0.29	0.78	0.26
	f1		0.86		0.94		0.34		0.45		0.39	
ADPT-CMD	**P**	**R**	0.56	1	0.86	1	1	0.62	1	0.62	0.34	0.4
	f1		0.72		0.92		0.77		0.77		0.37	
TPLT	**P**	**R**	0.30	1	1	0.75	0.15	0.5	0.15	0.5	0.19	1
	f1		0.46		0.86		0.23		0.23		0.32	

Table 6.8: Comparison with Tools that used JHD, JRF and QUM

Results: Precision, Recall, and F1-score were calculated on JHD, JRF, and QUM. Only these three systems were used for this evaluation to compare the average precision and recall with other related tools in Table 6.8. MPX was excluded as RM[106] and DPF[29] did not include it in their evaluations. The ePAD Web Appendix [40] was used to establish TP, FP, and FN values to calculate precision and recall of ePAD, DPD, RM, and DPF. The average precision and recall (0.56,1) of SPARQL queries generated with PatternScout are better than (DPD, RM, DPF) or comparable to (ePAD) that of the other tools. The accuracy is higher compared to DPD, RM, and DPF because they rely exclusively on structural analysis. ePAD uses a more sophisticated pipeline and performs similarly to the queries generated by PatternScout which also includes behavioral specification.

6.2.2 Performance Measures

We also performed experiments to measure the execution times of the generated queries of six design patterns, with their variations (see Variants in Table 3.1) on projects listed in Table 6.4). Queries were executed on a 2.3GHz Intel Core i5 processor with 8 GB of RAM and a Java Heap Size of 2 GB. We executed the queries twice and calculated the average (see Figure 6.5).

We see that execution time is related to the number of sub-patterns detected (see Table 6.6). The longest-running design patterns, State-Strategy, and Prototype in the JHotDraw project correspond with the highest number of sub-patterns detected in Table 6.6. This is likely how we keep the detected patterns in memory until we serialize them to a file at the end.

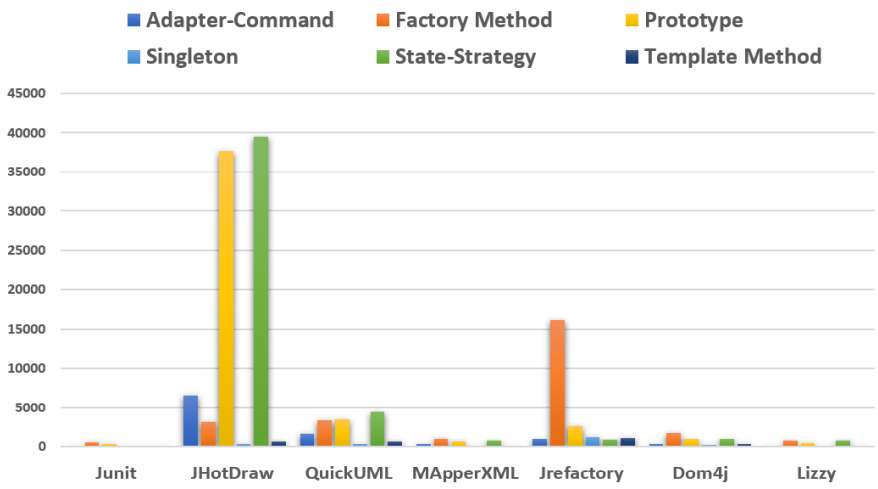


Figure 6.5: Execution times in ms

Chapter 7

DISCUSSION

This section covers threats to validity, the language-agnostic potential as well as current limitations of both approaches.

7.1 Approach 1

We now revisit the research questions we discussed earlier in the paper.

RQ1: Is there a significant correlation between keywords/n-grams extracted from software documentation (technical/functional/user guides) with security vulnerabilities reported in the system post-development?

Our current results suggest that there seems to be a significant correlation between keywords extracted from documentations with security vulnerability reports. The random forest classifier, for instance, was able to achieve an average accuracy of 82%. For some samples, the model was able to achieve accuracy up to 84.5% with 100% recall and 86% precision. This indicates that keywords extracted from specification documents are reliable to be used as features for predicting security vulnerabilities. The fact that keywords extracted using RAKE outperformed TF-IDF vectorization in feature election shows that using all words in documentation is not required. That is, selected keywords from a specification document are sufficient for the process of vulnerability prediction.

RQ2: Are different security vulnerabilities significantly correlated with each other such that detection of one vulnerability can be used as a warning for the other.

We were able to identify 7 vulnerability pairs (see Table 6.3) with a correlation coefficient of 0.5 or higher with $p < 0.001$. The actual correlation could be higher than the computed

value of the correlation coefficient as many vulnerabilities are under-reported. The highly correlated pairs can be used for reference when developers are creating security fixes for vulnerabilities. If a vulnerability is detected in a system, the findings of the correlation analysis can be used to identify and proactively correct a vulnerability before it can be exploited by malicious actors.

RQ3: How do different binary classification algorithms perform in terms of predicting security vulnerabilities by training on keywords and n-grams extracted from software documentation?

Our results show that the tree-based ensemble method Random Forest classifier was found to outperform logistic regression, multi-level perceptron, support vector machines, and k-nearest neighbor algorithms. The performance of Support Vector Machines and Random Forest classifiers improved drastically when class imbalance strategies (adding regularization parameters, penality, and class weighting) were implemented. Random Forest classifier showed higher resilience towards the problem of overfitting as the accuracy, precision, recall, f1-score, and ROC AUC score were consistently higher irrespective of the test-train split of the dataset.

RQ4: Does a perceptron model or multilayered neural network perform better in predicting security vulnerabilities using the same features?

As part of the preliminary evaluation, a multi-level perceptron model was evaluated for predicting CWE 79 in the 3602 products in the dataset. Hyperparameter tuning using multi-metric grid search was conducted to identify the best performing parameter combinations from the following options:

```
{
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
    'solver': ['lbfgs', 'sgd', 'adam'], 'max_iter': [200, 500,1000],
    'alpha': [0.001,0.01,1], 'learning_rate': ['constant', 'invscaling', 'adaptive'],
    'hidden_layer_sizes': [(100,),(5,2),(100,50,100),(50,100,50)]
}
```

MLP architecture with 2 hidden layers with 5 and 2 hidden units (neurons) each was found

to be the best performing combination. However, the model was found to overfit on samples as observed in cross-validation analysis. This was possibly due to a large number of features compared to the number of samples in the dataset. Feature reduction methods were not able to counter the problem without causing performance deterioration. This problem can however be overcome by incorporating more products into the dataset so that the number of samples is larger than the number of features. The MLP architecture without hidden layers performed similarly to a logistic regression model. Random Forest classifier was found to outperform MLP for all the different hyperparameter combinations we evaluated.

RQ5: Can the dataset created as part of this research be used to create a pre-trained model to predict security vulnerabilities for new software products before their development?

The model was able to correctly predict the presence of CWE 119 in Seamonkey 2.31 and Libav 10.2 similar to source code analysis-based techniques like [84]. This shows that KEVIS can be used to predict security vulnerabilities for new software products before their development by using just the specification documents with an accuracy comparable to existing deep learning-based source code analysis and static analysis techniques. The technique also shows promising flexibility in supporting products developed in different programming languages as the technique only depends on specification documents that are written in natural languages. The method is also generalizable over different software system domains (payment systems, e-commerce, operating systems, etc). While the technique was only validated over 3602 products for the preliminary evaluation, the dataset can be extended to incorporate the remaining 43000 products with vulnerability reports on CVE. Further evaluation is also needed to conclude if the model performance is dependent on product type, domain, and programming language used for its development or the type of vulnerability.

7.1.1 Challenges

A major challenge is the creation of the dataset as there are no existing datasets that can be re-used for this study. While we have managed to create programs that can automatically

download products to vulnerability mapping from the CVE website, downloading documentation files corresponding to the products in the CVE mapping is difficult. Some open-source websites like GNU[10] contain text-based documentation available for most of their products that can be automatically downloaded using a website parsing algorithm. However, for most other vendors and products, the documentation cannot be downloaded in bulk as the webpage for each product differs in structure. Moreover, many companies build both software and hardware products, and identifying the products that are part of the CVE dataset is time-consuming. Most products have online HTML-based documentation or PDF documents. If the documentation is in PDF format, this can cause encoding issues and invalid characters due to incorrect conversion of special characters and media files embedded in the document.

Another challenge is the computational complexity due to a large number of features. The size of specification documents can range from anywhere between 10-5000 pages. Even if we select only the top 1000 keywords from each document, this can add up to a significant number when keywords from all documents are combined to create the final feature set.

In addition to the class imbalance problem discussed in Section 5.1.4, we also faced the challenge that not all CWE labels had enough samples for classifiers to converge with reasonable accuracy. To tackle this problem, the entries were grouped together based on Vulnerability classifications. That is labels of child vulnerabilities were merged under parent vulnerability pillars. For example, all samples under *CWE-74: Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')* was merged under *CWE-707: Improper Neutralization*. When a child vulnerability has multiple parent pillars, samples were added to each parents' subset.

7.1.2 Limitations

Since KEVIS is inherently a static analysis method, it has the limitation that the output still needs to be verified. That is, the model can only be used to create checkpoints for the developers during the development or maintenance process. Moreover, as a static analy-

sis method, it can only approximate program behaviors. Like other static analysis methods, KEVIS also attempts to achieve goals despite this limitation. Compared to other static analysis methods, however, KEVIS has the added advantage of having a closer proxy to source code analysis as specification documents list functionalities, technologies and libraries used, the programming language used as well as architectural decisions involved. For instance, the mention of programming language in the specification document helps KEVIS identify language-specific vulnerabilities despite being a static analysis method.

7.1.3 Threats to Validity

Internal validity: To avoid selection bias, we used all known vulnerability reports for an application as per CVE. Reports were selected without regard to the severity or type of vulnerability reported. The criteria used to select applications for which documentation was downloaded were not random. To have a large set of product documentation and vulnerabilities to use for model building and prediction, we chose vendors with a large number of products each with a significant number of vulnerability reports. To avoid any manual bias in labeling products as vulnerable or not, the reports on CVE were used as the ground truth.

External validity: Unlike existing studies where results might be specific to the selected applications studied, our model was cross-validated and evaluated over a dataset of 3602 products. These products are of different types (Applications, Operating Systems, Hardware/Appliances), from different domains, and written in different programming/scripting languages. The dataset includes both commercial and open-source systems. Therefore, results will generalize over different programming languages and types of applications.

7.2 Approach 2

7.2.1 Threats to Validity

Internal Validity: Internal validity can be affected by the design patterns used to evaluate performance. As manually recovering all the pattern instances in each open source project

requires manual effort and are prone to human errors, we used the patterns that were common in evaluations of existing approaches [87] [125] [106] and [29]. These benchmarks are publicly available and maintained by researchers. They have been widely studied in the literature [20] [142], which reduces human error.

The design pattern instances in each project was further cross validated with [137] [23].

External Validity: Since the selection of the open source projects could pose a threat to external validity of the technique, the same projects that were studied in existing approaches were chosen to enable a thorough comparison. After carefully evaluating the evaluation context of existing approaches [108] [125] [142] [23] [39] [20] [73] as per surveys of different design pattern detection techniques [19] [46] [107] [106], the most commonly used open source projects were selected. The projects evaluated are listed in Table 6.4. While many approaches used projects like Swing, Apache Ant [66] [116] [39], AWT [44], and Log4J [140], those projects were excluded due to inconsistency in the versions of the projects used or the unavailability of the evaluated versions for download.

7.2.2 Limitation: UML Modeling

UML Ambiguities

The UML notation, which is semi-formal, may lead to ambiguities and inconsistencies [48]. The accuracy of our results can be improved if UML diagrams incorporate additional stereotypes, tagged values, constraints and meta-model elements [45] [43]. The Meta Models used for converting UML diagrams to XMI format that are embedded in software like StarUML and Visual Paradigm do not have a standardized practice for handling comments and stereotypes. While PatternScout has support for stereotypes like "constructor" and "override" annotation that are widely adopted in the industry, this feature is not robust as long as the handling of stereotypes is not standardized across UML Tools.

Sequence Diagram: Message Activation

In the XMI encoding of Sequence diagrams, most tools support Lifelines, Messages, Fragments, and Message Signature. However, some tools do not save the activation points of each message on the Lifeline. An RDF triple representing behavioral specifications like the sequence of method invocations or reference of a method in another method requires the $fromActivation$ attribute of one message to be matched against the *toActivation* attribute of another message. The support for storing message activation in XMI export of Sequence diagrams can significantly increase accuracy.

7.2.3 Limitation: Constraints

Differentiating hard and soft constraints

The current implementation of PatternScout does not differentiate between hard and soft constraints. For example, in the implementation of the Factory pattern, a UML Class Diagram shows two classes that implement the factory interface to represent cases of more than one implementation. However, the hard constraint is only to have at least one implementation. While PatternScout currently does not automatically identify this, intelligent ways can be incorporated to prioritize hard and soft constraints. For example, the WHERE clause of the query can be dedicated for hard constraints with the FILTER section handling all soft constraints. While hard constraints use "intersection" operation to filter results, soft constraints can use "union" operation. Results could further be ranked based on how many soft constraints are matched.

Negative Constraints

PatternScout relies on the granularity of the design pattern specification in the UML Class diagram and sequence diagram. While many design patterns can benefit from specifying negative constraints, UML diagrams inherently do not support this feature. For example, the constructor of a Singleton class should be private and should only be accessed through

the getInstance method that returns the instance which is instantiated only once during the lifecycle. While the UML diagram can support a specification that the constructor should be private, it cannot support a negative constraint that the constructor should not be public. Due to the dependency of PatternScout on the feature coverage of UML Diagrams, the limitations of UML diagrams in representing design patterns will also be limitations of PatternScout in detecting design patterns.

7.2.4 Lessons Learned

Tradeoff: Precision & Recall

The tradeoff in precision and recall depends upon how restrictive is the SPARQL query. For the same design pattern, the SPARQL query can be more restrictive if there are conditions specifying visibility, property, stereotypes, etc. The query will be less restrictive if these constraints are relaxed. While a more restrictive query can reduce false-positive results (increase precision). This, however, can fail to retrieve some instances that do not conform strictly to the structure of a pattern. If the SPARQL query is less restrictive, then it will include cases that exhibit the same structure as the pattern but are not meant to be instances of the pattern. This will likely result in false positives. For patterns like Singleton where access modifiers of the constructor and instance are important, a more restrictive query can be very precise. However, to avoid the false-negative results, a less restrictive query can be maintained as a variant. In addition, with our query map, users can be specific on which variants they wish to detect.

Nested Classes:

The RDF Triples created by CodeOntology for nested classes differ from triples created for non-nested classes. Hence if a design pattern can have a variant that uses nested classes in its implementation, a variant SPARQL query needs to be generated using a UML diagram that specifies this behavior to avoid a high false-negative rate.

Multiple Results for one pattern:

Query results show multiple sub-pattern entries corresponding to the same pattern instance (concept explained in [125]). For example, when executing the Visitor pattern query discussed in Section 5.2.1 on a system with visitor pattern implementation that has 2 Concrete Visitor classes and 2 Concrete Elements, the same pattern will appear as 4 result entries by substituting each Concrete Element class to the variables ?$ConcreteElementA18$ and ?$ConcreteElementB22$, and each Concrete Visitor class to the variables ?$ConcreteVisitor15$ and ?$ConcreteVisitor211$ in the SELECT statement. Similarly, an implementation with more than 2 Concrete Visitor classes will be split into multiple results by selecting 2 Concrete Visitors at a time.

A sample visitor pattern instance detected by the query is given below:

```
| Visitor27 | VisitConcreteElementA11 | VisitConcreteElementB13 |
VisitConcreteElementA27 | VisitConcreteElementB29 | Accept13 |
AcceptA16 | AcceptB20 | VisitConcreteElementA24 |
VisitConcreteElementB26 | ConcreteVisitor15 | ConcreteVisitor211 |
Element14 | ConcreteElementA18 | ConcreteElementB22 |
===========================
| <woc:me.zbl.visitor.UnitVisitor>|<woc:me.zbl.visitor.EngineerVisitor-
visitBoss(me.zbl.visitor.Boss)>|<woc:me.zbl.visitor.EngineerVisitor-
visitEngineer(me.zbl.visitor.Engineer)>|<woc:me.zbl.visitor.ManagerVisitor-
visitBoss(me.zbl.visitor.Boss)> | <woc:me.zbl.visitor.ManagerVisitor-
visitEngineer(me.zbl.visitor.Engineer)>  | <woc:me.zbl.visitor.Unit-
beVisited(me.zbl.visitor.UnitVisitor)> | <woc:me.zbl.visitor.Boss-
beVisited(me.zbl.visitor.UnitVisitor)> | <woc:me.zbl.visitor.Engineer-
beVisited(me.zbl.visitor.UnitVisitor)> | <woc:me.zbl.visitor.UnitVisitor-
visitBoss(me.zbl.visitor.Boss)> | <woc:me.zbl.visitor.UnitVisitor-visitEngineer
(me.zbl.visitor.Engineer)>|<woc:me.zbl.visitor.EngineerVisitor> |
<woc:me.zbl.visitor.ManagerVisitor> | <woc:me.zbl.visitor.Unit> |
<woc:me.zbl.visitor.Boss> | <woc:me.zbl.visitor.Engineer> |
```

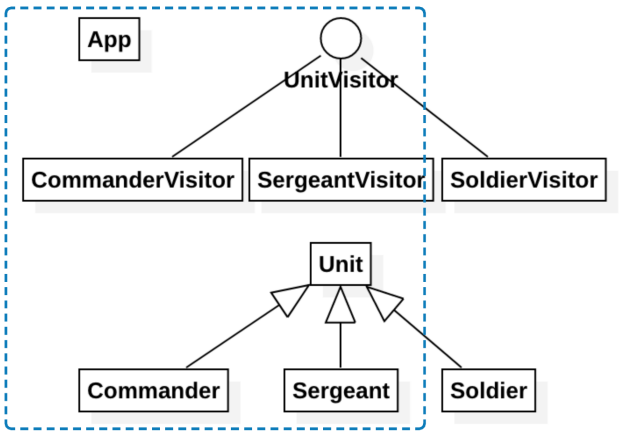


Figure 7.1: Visitor Sub-pattern detected by SPARQL query

This output is illustrated by Figure 7.1 which shows part of the pattern, in dashed box, that is detected by the query. The result selects 2 out of 3 concrete visitor classes and 2 out of 3 concrete element classes. Results can grow, as permutations of results also apply to methods and other entities, not just classes. To address this issue of multiple results, we can filter our results to only show unique instances of the pattern.

Chapter 8

CONCLUSION

As part of this study, we created an automated vulnerability prediction technique to identify vulnerabilities in a software system using specification documents. Further, we developed PatternScout, a novel technique that generates SPARQL queries from Class and Sequence diagrams to find design patterns in source code. For the vulnerabilities identified using KEVIS, security engineers can use PatternScout to verify if the mitigation measures are already incorporated in the codebase. Any missing mitigation measures (e.g. security design patterns) can be implemented before the vulnerability is exploited by adversaries.

While no predictive model can be expected to be 100% accurate, KEVIS is expected to be used to give checkpoints for software engineers. That is, before the development of a system, the specification document can be tested using this pre-trained model to get a list of potential vulnerabilities to look out for. This will enable the engineers to make a list of potential design patterns to be included in the development to tackle these vulnerabilities. The correlation analysis can also be used to see other correlated vulnerabilities that developers can keep in mind while making architectural/design decisions. The current implementation of KEVIS does meet the requirements discussed in Section 4. The method also has the added advantage of being agnostic to programming languages. Source code analysis-based methods and static analysis methods that use source code metrics as their features are very dependent on the source code in which the system is developed. On the other hand, using specification documents the input for the predictive system removes this dependency. The documentation of most of the systems is available in natural languages (mostly English language). They however mention the programming language and libraries that the system uses internally. This will help the keyword extraction-based method to be able to operate on products built

in a variety of programming languages while still being able to identify vulnerabilities related to specific programming languages and libraries.

The queries generated by PatternScout has the same granularity as the input diagrams in terms of capturing entities and relationships between those entities. Thus, it can identify more types of patterns than other techniques. While we primarily focused on object-oriented design patterns, security design patterns that can be expressed as Class or Sequence diagrams can also be detected.

We evaluated PatternScout using representative patterns from three types of design patterns: creational, structural, and behavioral. Precision and recall on the open-source projects that our technique is comparable to or better than related tools. The tool is lightweight as it does not require a large amount of data to train the model, pre-processing, or manual effort to identify each pattern. Performance measures show that overhead is noticeable for large projects (in tens of thousands of lines of code). Finally, we offer lessons learned from our experience with assessing our technique.

DATA AVAILABILITY

The dataset used for evaluations, the code for dataset pre-processing, benchmarking, data vectorization, and model evaluation are available at [11]. The website parsing program developed for the dataset creation is available at [1].

The XMI transformations to convert UML diagram to Java objects, input XMI files (UML Class and Sequence Diagrams), generated SPARQL queries used for evaluations, and the RDF graphs of the dataset are included as supplementary material. The repository of the SPARQL queries for 107 variants of the 23 design patterns, the source code and build of PatternScout tool to convert UML Class diagrams and Sequence diagrams to SPARQL queries, and the tool to execute all the SPARQL queries in batches on a given project's source code is available at [16].

FUTURE WORK

Future work includes the following. The performance of the automatic vulnerability detection model can be improved by extending the dataset to cover more products across domains, programming languages, and product types (Hardware/Appliances, Operating Systems, Applications). This will also enable granular analysis on how the model performance varies with these differences. The variance in performance related to vulnerability types also needs to be evaluated. Deep learning techniques or more complex neural network architectures can be used to evaluate if the model performance can be fine-tuned further.

The performance of PatternScout can be improved by adding a mechanism to distinguish hard and soft constraints for each design pattern. The automatically generated queries also need to be evaluated for programming languages other than Java by using the appropriate Ontology to serialize projects to RDF graphs.

Combining the results of the automated vulnerability detection and PatternScout, a recommendation system can be created to mitigate detected vulnerabilities. Mapping of security design patterns that mitigate each vulnerability can be maintained. Once the specification document is scanned and a potential vulnerability is detected, a query generated using pattern scout can be used to verify if the pattern is already present in the tool. Missing patterns can be incorporated to proactively mitigate vulnerabilities that might otherwise be exploited in the future.

BIBLIOGRAPHY

[1] Bulk download oss docs. https://github.com/jeffyjahfar/BulkDownloadOSSDocs. (accessed May 24,2021).

[2] China national vulnerability database of information security. http://www.cnnvd.org.cn. (accessed December 13,2020).

[3] cudf - gpu dataframes. https://github.com/rapidsai/cudf. (accessed May 24,2021).

[4] cuml - gpu machine learning algorithms. https://github.com/rapidsai/cuml. (accessed May 24,2021).

[5] Cve - common vulnerabilities and exposures. https://cve.mitre.org/. (accessed December 13,2020).

[6] Cve security vulnerability database. security vulnerabilities, exploits, references and more. https://www.cvedetails.com/. (accessed December 13,2020).

[7] Cwe - common weakness enumeration. https://cwe.mitre.org/. (accessed December 13,2020).

[8] Design patterns implemented in java. https://github.com/iluwatar/java-design-patterns/1. (accessed: 05.27.2020).

[9] difflib — helpers for computing deltas. https://docs.python.org/3.8/library/difflib.html. (accessed May 24,2021).

[10] Gnu operating system. https://www.gnu.org/manual/manual.html. (accessed May 24,2021).

[11] Gnu operating system. https://drive.google.com/drive/folders/1ef6XQv-AIyClJkgZ0iY33G6-hEl9lKHD?usp=sharing. (accessed May 24,2021).

[12] Libav documentation. https://libav.org/documentation/. (accessed May 24,2021).

[13] National vulnerability database. https://nvd.nist.gov/. (accessed December 13,2020).

[14] Natural language toolkit. `nltk.org`. (accessed May 24,2021).

[15] The seamonkey® project. `https://www.seamonkey-project.org/`. (accessed May 24,2021).

[16] Uml to sparql converter. `https://github.com/jeffyjahfar/uml-to-sparql-converter`. (accessed: 05.27.2020).

[17] Xen project 4.6 man pages. `https://wiki.xenproject.org/wiki/Xen_Project_4.6_Man_Pages`. (accessed May 24,2021).

[18] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of the sixth ACM conference on data and application security and privacy*, pages 183–194, 2016.

[19] Mohammed Ghazi Al-Obeidallah, Miltos Petridis, and Stelios Kapetanakis. A survey on design pattern detection approaches. *International Journal of Software Engineering (IJSE)*, 7(3):41–59, 2016.

[20] Awny Alnusair, Tian Zhao, and Gongjun Yan. Rule-based detection of design patterns in program code. *International Journal on Software Tools for Technology Transfer*, 16(3):315–334, 2014.

[21] Apostolos Ampatzoglou, Olia Michou, and Ioannis Stamelos. Building and mining a repository of design pattern instances: Practical and research benefits. *Entertainment Computing*, 4(2):131–142, 2013.

[22] Giuliano Antoniol, Roberto Fiutem, and Luca Cristoforetti. Design pattern recovery in object-oriented software. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242)*, pages 153–160. IEEE, 1998.

[23] Giuliano Antoniol and Yann-Gaël Guéhéneuc. Demima: A multilayered approach for design pattern identification. *IEEE Transactions on Software Engineering*, 34(5):667–684, 2008.

[24] Mattia Atzeni and Maurizio Atzori. Codeontology: Querying source code in a semantic framework. In *International Semantic Web Conference (Posters, Demos & Industry Tracks)*, 2017.

[25] Mattia Atzeni and Maurizio Atzori. Codeontology: Rdf-ization of source code. In *International Semantic Web Conference*, pages 20–28. Springer, 2017.

[26] Zsolt Balanyi and Rudolf Ferenc. Mining design patterns from c++ source code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 305–314. IEEE, 2003.

[27] Guntis Barzdins, Sergejs Rikacovs, and Martins Zviedris. Graphical Query Language as SPARQL Frontend. page 15.

[28] Federico Bergenti and Agostino Poggi. Improving uml designs using automatic design pattern detection. In *Handbook of Software Engineering and Knowledge Engineering: Volume II: Emerging Technologies*, pages 771–784. World Scientific, 2002.

[29] Mario Luca Bernardi, Marta Cimitile, and Giuseppe Di Lucca. Design pattern detection using a dsl-driven graph matching approach. *Journal of Software: Evolution and Process*, 26(12):1233–1266, 2014.

[30] Alex Blewitt, Alan Bundy, and Ian Stark. Automatic verification of design patterns in java. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 224–232, 2005.

[31] Grady Booch, James Rumbaugh, and Ivar Jacobson. Unified modeling language user guide,(the 2nd edition), 2005.

[32] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Neural Information Processing Systems (NIPS)*, pages 1–9, 2013.

[33] Michaela Bunke. *Security-Pattern Recognition and Validation*. PhD thesis, Universität Bremen, 2019.

[34] Xiang Chen, Yingquan Zhao, Zhanqi Cui, Guozhu Meng, Yang Liu, and Zan Wang. Large-scale empirical studies on effort-aware security vulnerability prediction methods. *IEEE Transactions on Reliability*, 69(1):70–87, 2019.

[35] Yi Chen, Luyi Xing, Yue Qin, Xiaojing Liao, XiaoFeng Wang, Kai Chen, and Wei Zou. Devils in the guidance: predicting logic vulnerabilities in payment syndication services through automated documentation analysis. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 747–764, 2019.

[36] Zhongqiang Chen, Mema Roussopoulos, Zhanyan Liang, Yuan Zhang, Zhongrong Chen, and Alex Delis. Malware characteristics and threats on the internet ecosystem. *Journal of Systems and Software*, 85(7):1650–1672, 2012.

[37] Koby Crammer and Yoram Singer. On the algorithmic implementation of multiclass kernel-based vector machines. *Journal of machine learning research*, 2(Dec):265–292, 2001.

[38] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. Large-scale malware classification using random projections and neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3422–3426. IEEE, 2013.

[39] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Design pattern recovery through visual language parsing and source code analysis. *Journal of Systems and Software*, 82(7):1177–1193, 2009.

[40] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Carmine gravino — web appendix - epad. `https://docenti.unisa.it/004724/risorse?categoria=335&risorsa=1126`, 2016.

[41] M. Dean and G. Schreiber. Owl web ontology language reference. `http://www.w3.org/TR/owl-ref/`.

[42] Beniamino Di Martino and Antonio Esposito. Automatic recognition of design patterns from uml-based software documentation. In *Proceedings of International Conference on Information Integration and Web-based Applications & Services*, pages 280–289, 2013.

[43] Jing Dong. Uml extensions for design pattern compositions. *Journal of object technology*, 1(5):151–163, 2002.

[44] Jing Dong, Dushyant S Lad, and Yajing Zhao. Dp-miner: Design pattern discovery using matrix. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, pages 371–380. IEEE, 2007.

[45] Jing Dong and Sheng Yang. Visualizing design patterns with a uml profile. In *IEEE Symposium on Human Centric Computing Languages and Environments, 2003. Proceedings. 2003*, pages 123–125. IEEE, 2003.

[46] Jing Dong, Yajing Zhao, and Tu Peng. A review of design pattern mining techniques. *International Journal of Software Engineering and Knowledge Engineering*, 19(06):823–855, 2009.

[47] Chad Dougherty, Kirk Sayre, Robert C Seacord, David Svoboda, and Kazuya Togashi. Secure design patterns. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2009.

[48] Ashish Kumar Dwivedi, Anand Tirkey, and Santanu Kumar Rath. An ontology based approach for formal modeling of structural design patterns. In *2016 Ninth International Conference on Contemporary Computing (IC3)*, pages 1–6. IEEE, 2016.

[49] Clément Elbaz, Louis Rilling, and Christine Morin. Automated keyword extraction from" one-day" vulnerabilities at disclosure. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2020.

[50] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[51] Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[52] Zhentan Feng, Shuguang Xiong, Deqiang Cao, Xiaolu Deng, Xin Wang, Yang Yang, Xiaobo Zhou, Yan Huang, and Guangzhu Wu. Hrs: A hybrid framework for malware detection. In *Proceedings of the 2015 ACM International Workshop on International Workshop on Security and Privacy Analytics*, pages 19–26, 2015.

[53] Rudolf Ferenc, Arpad Beszedes, Lajos Fulop, and Janos Lele. Design pattern mining enhanced by machine learning. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 295–304. IEEE, 2005.

[54] Roger Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2013.

[55] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[56] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[57] Qichuan Geng, Zhong Zhou, and Xiaochun Cao. Survey of recent progress in semantic image segmentation with cnns. *Science China Information Sciences*, 61(5):051101, 2018.

[58] Christian Giménez, Germán Braun, Laura Cecchi, and Pablo Rubén Fillottrani. Towards a Visual SPARQL-DL Query Builder. 2018.

[59] Jacob Goldberger, Geoffrey E Hinton, Sam Roweis, and Russ R Salakhutdinov. Neighbourhood components analysis. *Advances in neural information processing systems*, 17:513–520, 2004.

[60] Greg Goth. Functionality meets terminology to address network security vulnerabilities. *IEEE Distributed Systems Online*, 7(6):4–4, 2006.

[61] Mariano Graziano, Davide Canali, Leyla Bilge, Andrea Lanzi, Elaine Shi, Davide Balzarotti, Marten van Dijk, Michael Bailey, Srinivas Devadas, Mingyan Liu, et al. Needles in a haystack: Mining information from public dynamic analysis sandboxes for malware intelligence. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 1057–1072, 2015.

[62] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 3–14. IEEE, 2017.

[63] Manjari Gupta and Akshara Pande. Design patterns mining using subgraph isomorphism: Relational view. *International Journal of Software Engineering and Its Applications (IJSEIA)*, 270, 2011.

[64] Florian Haag, Steffen Lohmann, and Thomas Ertl. SparqlFilterFlow: SPARQL Query Composition for Everyone. In Valentina Presutti, Eva Blomqvist, Raphael Troncy, Harald Sack, Ioannis Papadakis, and Anna Tordai, editors, *The Semantic Web: ESWC 2014 Satellite Events*, Lecture Notes in Computer Science, pages 362–367, Cham, 2014. Springer International Publishing.

[65] Nadine Hanebutte and Paul W Oman. Software vulnerability mitigation as a proper subset of software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(6):379–400, 2005.

[66] Dirk Heuzeroth, Thomas Holl, Gustav Hogstrom, and Welf Lowe. Automatic design pattern detection. In *11th IEEE International Workshop on Program Comprehension, 2003.*, pages 94–103. IEEE, 2003.

[67] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F Patel-Schneider, Sebastian Rudolph, et al. Owl 2 web ontology language primer. *W3C recommendation*, 27(1):123, 2009.

[68] Aram Hovsepyan, Riccardo Scandariato, Wouter Joosen, and James Walden. Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th international workshop on Security measurements and metrics*, pages 7–10, 2012.

[69] Tom Howley, Michael G Madden, Marie-Louise O'Connell, and Alan G Ryder. The effect of principal component analysis on machine learning accuracy with high dimensional spectral data. In *International Conference on Innovative Techniques and Applications of Artificial Intelligence*, pages 209–222. Springer, 2005.

[70] Konrad Höffner, Sebastian Walter, Edgard Marx, Ricardo Usbeck, Jens Lehmann, and Axel-Cyrille Ngonga Ngomo. Survey on challenges of Question Answering in the Semantic Web. *Semantic Web*, 8(6):895–920, August 2017.

[71] IEEE. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[72] Nathalie Japkowicz. The class imbalance problem: Significance and strategies. In *Proc. of the Int'l Conf. on Artificial Intelligence*, volume 56. Citeseer, 2000.

[73] Olivier Kaczor, Y-G Guéhéneuc, and Sylvie Hamel. Efficient identification of design patterns with bit-vector algorithm. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 10–pp. IEEE, 2006.

[74] Naoto Kawaguchi and Kazumasa Omote. Malware function classification using apis in initial behavior. In *2015 10th Asia Joint Conference on Information Security*, pages 138–144. IEEE, 2015.

[75] Peyman Khodamoradi, Mahmood Fazlali, Farhad Mardukhi, and Masoud Nosrati. Heuristic metamorphic malware detection based on statistics of assembly instructions using classification algorithms. In *2015 18th CSI International Symposium on Computer Architecture and Digital Systems (CADS)*, pages 1–6. IEEE, 2015.

[76] Sascha Konrad, Betty HC Cheng, Laura A Campbell, and Ronald Wassermann. Using security patterns to model and analyze security requirements. *Requirements Engineering for High Assurance Systems (RHAS'03)*, 11, 2003.

[77] Bum Jun Kwon, Jayanta Mondal, Jiyong Jang, Leyla Bilge, and Tudor Dumitraş. The dropper effect: Insights into malware distribution with downloader graph analytics. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1118–1129, 2015.

[78] Charles LeDoux and Arun Lakhotia. Malware and machine learning. In *Intelligent Methods for Cyber Warfare*, pages 1–42. Springer, 2015.

[79] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 318–328. IEEE, 2017.

[80] Xiang Li, Jinfu Chen, Zhechao Lin, Lin Zhang, Zibin Wang, Minmin Zhou, and Wanggen Xie. A mining approach to obtain the software vulnerability characteristics. In *2017 Fifth International Conference on Advanced Cloud and Big Data (CBD)*, pages 296–301. IEEE, 2017.

[81] Zhen Li, Deqing Zou, Jing Tang, Zhihao Zhang, Mingqian Sun, and Hai Jin. A comparative study of deep learning-based vulnerability detection system. *IEEE Access*, 7:103184–103197, 2019.

[82] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 201–213, 2016.

[83] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[84] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[85] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, and Yang Xiang. Poster: Vulnerability discovery with function representation learning from unlabeled projects. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2539–2541, 2017.

[86] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. Software vulnerability discovery techniques: A survey. In *2012 fourth international conference on multimedia information networking and security*, pages 152–156. IEEE, 2012.

[87] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Detecting the behavior of design patterns through model checking and dynamic analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(4):1–41, 2018.

[88] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. Probabilistic information retrieval. *Introduction to Information Retrieval*, pages 220–235, 2009.

[89] Frank Manola, Eric Miller, Brian McBride, et al. Rdf primer. *W3C recommendation*, 10(1-107):6, 2004.

[90] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *arXiv preprint arXiv:1310.4546*, 2013.

[91] Aziz Mohaisen, Omar Alrawi, and Manar Mohaisen. Amal: high-fidelity, behavior-based automated malware analysis and classification. *computers & security*, 52:251–266, 2015.

[92] Stephan Neuhaus and Thomas Zimmermann. Security trend analysis with cve topic models. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 111–120. IEEE, 2010.

[93] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540, 2007.

[94] Jörg Niere, Matthias Meyer, and Lothar Wendehals. User-driven adaption in rule-based pattern recognition. 2004.

[95] Lisa O'Conner. 2016 ieee tenth international conference on semantic computing, icsc 2016: 3-5 february 2016, laguna hills, california: proceedings. In *10th IEEE International Conference on Semantic Computing*. IEEE Computer Society, 2016.

[96] Leon Osterweil. Integrating the testing, analysis and debugging of programs. In *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, pages 73–102, 1984.

[97] Yulei Pang, Xiaozhen Xue, and Akbar Siami Namin. Predicting vulnerable software components through n-gram analysis and statistical feature selection. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 543–548. IEEE, 2015.

[98] Attawat Panich and Wiwat Vatanawood. Detection of design patterns from class diagram and sequence diagrams using ontology. In *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, pages 1–6. IEEE, 2016.

[99] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, 46(9):1155–1179, 2016.

[100] Samad Paydar and Mohsen Kahani. A semantic web based approach for design pattern detection from source code. In *2012 2nd International eConference on Computer and Knowledge Engineering (ICCKE)*, pages 289–294. IEEE, 2012.

[101] Nitin Namdeo Pise and Parag Kulkarni. A survey of semi-supervised learning methods. In *2008 International conference on computational intelligence and security*, volume 2, pages 30–34. IEEE, 2008.

[102] Jedrzej Potoniec. Learning SPARQL Queries from Expected Results. *Computing and Informatics*, 38(3):679–700, 2019.

[103] Hamid Reza Pourghasemi and Omid Rahmati. Prediction of the landslide susceptibility: which algorithm, which precision? *Catena*, 162:177–192, 2018.

[104] Edward Raff and Charles Nicholas. An alternative to ncd for large sequences, lempel-ziv jaccard distance. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1007–1015, 2017.

[105] Ghulam Rasool and Hassan Akhtar. Towards a catalog of design patterns variants. In *2019 International Conference on Frontiers of Information Technology (FIT)*, pages 156–161. IEEE, 2019.

[106] Ghulam Rasool and Patrick Mäder. A customizable approach to design patterns recognition based on feature types. *Arabian Journal for Science and Engineering*, 39(12):8851–8873, 2014.

[107] Ghulam Rasool, Patrick Maeder, and Ilka Philippow. Evaluation of design pattern recovery tools. *Procedia Computer Science*, 3:813–819, 2011.

[108] Ghulam Rasool, Ilka Philippow, and Patrick Mäder. Design pattern recovery based on annotations. *Advances in Engineering Software*, 41(4):519–526, 2010.

[109] Stuart Rose, Dave Engel, Nick Cramer, and Wendy Cowley. Automatic keyword extraction from individual documents. *Text mining: applications and theory*, 1:1–20, 2010.

[110] Stuart J Rose, Vernon L Crow, Nicholas O Cramer, et al. Rapid automatic keyword extraction for information retrieval and analysis. Technical report, Pacific Northwest National Lab.(PNNL), Richland, WA (United States), 2012.

[111] Luis Alberto Benthin Sanguino and Rafael Uetz. Software vulnerability analysis using cpe and cve. *arXiv preprint arXiv:1705.05347*, 2017.

[112] Natan da Silva Severo and Ricardo de Sousa Job. An approach to detect false design patterns. In *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse*, pages 63–72, 2019.

[113] Joseph Sexton, Curtis Storlie, and Blake Anderson. Subroutine based detection of apt malware. *Journal of Computer Virology and Hacking Techniques*, 12(4):225–233, 2016.

[114] Asaf Shabtai, Robert Moskovitch, Yuval Elovici, and Chanan Glezer. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. *information security technical report*, 14(1):16–29, 2009.

[115] Kebin Shi, Yonghui Dai, and Jing Xu. Construction of a security vulnerability identification system based on machine learning. *Journal of Sensors*, 2020, 2020.

[116] Nija Shi and Ronald A Olsson. Reverse engineering of design patterns from java source code. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 123–134. IEEE, 2006.

[117] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 611–626, 2015.

[118] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE transactions on software engineering*, 37(6):772–787, 2010.

[119] Evren Sirin and Bijan Parsia. Sparql-dl: Sparql query for owl-dl. In *OWLED*, volume 258. Citeseer, 2007.

[120] Tommaso Soru, Edgard Marx, Diego Moussallem, Gustavo Publio, André Valdestilhas, Diego Esteves, and Ciro Baron Neto. SPARQL as a Foreign Language. *arXiv:1708.07624 [cs]*, May 2020. arXiv: 1708.07624.

[121] Alireza Souri and Rahil Hosseini. A state-of-the-art survey of malware detection approaches using data mining techniques. *Human-centric Computing and Information Sciences*, 8(1):1–22, 2018.

[122] UML StarUML. modeling tool. multilingual project. version 5.0. 2.1570, 2005.

[123] Jeffrey Stuckman, James Walden, and Riccardo Scandariato. The effect of dimensionality reduction on software vulnerability prediction models. *IEEE Transactions on Reliability*, 66(1):17–37, 2016.

[124] Qiang Tong, F. Zhang, and Jingwei Cheng. Construction of rdf(s) from uml class diagrams. *J. Comput. Inf. Technol.*, 22:237–250, 2014.

[125] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T Halkidis. Design pattern detection using similarity scoring. *IEEE transactions on software engineering*, 32(11):896–909, 2006.

[126] Satoru Uchiyama, Atsuto Kubo, Hironori Washizaki, Yoshiaki Fukazawa, et al. Detecting design patterns in object-oriented program source code by using metrics and machine learning. *Journal of Software Engineering and Applications*, 7(12):983, 2014.

[127] Michael Van Hilst and Eduardo B Fernandez. Reverse engineering and the verification of security patterns in code. 2007.

[128] Michael VanHilst and Eduardo B Fernandez. Reverse engineering to detect security patterns in code. In *Proc. of 1st International Workshop on Software Patterns and Quality. Information Processing Society of Japan (December 2007)*, 2007.

[129] Paradigm Visual. Visual paradigm. version 16.2, 2002.

[130] Jian Wang, Jiqing Sun, Hongfei Lin, Hualei Dong, and Shaowu Zhang. Convolutional neural networks for expert recommendation in community question answering. *Science China Information Sciences*, 60(11):1–9, 2017.

[131] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE, 2016.

[132] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 1113–1120, 2009.

[133] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98. IEEE, 2016.

[134] Terry Windeatt. Accuracy/diversity and ensemble mlp classifier design. *IEEE Transactions on Neural Networks*, 17(5):1194–1211, 2006.

[135] Xindong Wu, Vipin Kumar, J Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J McLachlan, Angus Ng, Bing Liu, S Yu Philip, et al. Top 10 algorithms in data mining. *Knowledge and information systems*, 14(1):1–37, 2008.

[136] J Wust. Sdmetrics: The software design metrics tool for uml, 2005.

[137] Renhao Xiong, David Lo, and Bixin Li. Distinguishing similar design pattern instances through temporal behavior analysis. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 296–307. IEEE, 2020.

[138] Zhou Xu, Shuai Li, Jun Xu, Jin Liu, Xiapu Luo, Yifeng Zhang, Tao Zhang, Jacky Keung, and Yutian Tang. Ldfr: Learning deep feature representation for software defect prediction. *Journal of Systems and Software*, 158:110402, 2019.

[139] Zhou Xu, Jin Liu, Xiapu Luo, Zijiang Yang, Yifeng Zhang, Peipei Yuan, Yutian Tang, and Tao Zhang. Software defect prediction based on kernel pca and weighted extreme learning machine. *Information and Software Technology*, 106:182–200, 2019.

[140] Shouzheng Yang, Ayesha Manzer, and Vassilios Tzerpos. Measuring the quality of design pattern detection results. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 53–62. IEEE, 2015.

[141] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26. IEEE, 2015.

[142] Dongjin Yu, Yanyan Zhang, and Zhenli Chen. A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures. *Journal of Systems and Software*, 103:1–16, 2015.

[143] Dongjin Yu, Yanyan Zhang, Jianlin Ge, and Wei Wu. From sub-patterns to patterns: an approach to the detection of structural design pattern instances by subgraph mining and merging. In *2013 IEEE 37th Annual Computer Software and Applications Conference*, pages 579–588. IEEE, 2013.

[144] MARCO Zanoni. *Data mining techniques for design pattern detection.* PhD thesis, Università degli Studi di Milano-Bicocca, 2012.

[145] Zhi-Xiang Zhang, Qing-Hua Li, and Ke-Rong Ben. A new method for design pattern mining. In *Proceedings of 2004 International Conference on Machine Learning and Cybernetics (IEEE Cat. No. 04EX826)*, volume 3, pages 1755–1759. IEEE, 2004.

[146] Wei Zheng, Jialiang Gao, Xiaoxue Wu, Fengyu Liu, Yuxing Xun, Guoliang Liu, and Xiang Chen. The impact factors on the performance of machine learning-based vulnerability detection: A comparative study. *Journal of Systems and Software*, 168:110659, 2020.

[147] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 421–428. IEEE, 2010.