Predicting True Vulnerabilities from Static Analyzer Warnings in Industry

An Attempt to Faster Releasing Software in Industry

S.P.D. Bisesser

Predicting True Vulnerabilities from Static Analyzer Warnings in Industry

An Attempt to Faster Releasing Software in Industry

by

S.P.D. Bisesser

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday December 14, 2020 at 11:00.

Student number:	1512250	
Project duration:	April 15, 2019 – December 14, 2020	
Thesis committee:	Dr. A. Panichella,	TU Delft, Supervisor
	Dr. ir. S. E. Verwer,	TU Delft, Committee Member
	Prof. dr. ir. R. L. Lagendijk,	TU Delft, Chair

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Abstract

An increasing digital world, comes with many benefits but unfortunately also many drawbacks. The increase of the digital world means an increase in data and software. Developing more software unfortunately also means a higher probability of vulnerabilities, which can be exploited by adversaries. Adversaries taking advantage of users and software vulnerabilities, by stealing data to cause harm, steal money, etc. This makes the digital world a dangerous environment.

To ensure software has a minimal number of vulnerabilities, companies invest in software tools and experts to check their software for vulnerabilities. One such company is ING, the largest bank of The Netherlands. At ING they use Fortify, a static analyzer. The problem with this tool is that it gives many false positives. Therefore, pentesters and developers have to manually check all the warnings given by Fortify, which takes a lot of time and slows down the whole software development process.

In this study, we propose to use supervised machine learning techniques to predict true vulnerabilities from static analyzer warnings. Using ING's data from Fortify, two highly imbalanced datasets with code metrics are created on class and method level. Various classifiers and sampling techniques are compared to determine which techniques perform the best. Next to that, we also compared the performance at different levels of granularity. Finally, we also investigate whether a dataset with different types of vulnerabilities performs better than a dataset consisting of only one vulnerability type.

From our study, it is clear that Bagging in combination with ClassBalancer gives the best f-measure (0.618) for the class-level dataset, which is slightly good. Random Forest with SMOTE gives the best f-measure (0.412) for the method-level dataset, which we consider weak. Depending on the type of vulnerability, the performance can benefit from a dataset per vulnerability type. Overall, the performance found in this study shows slightly promising results when using Fortify in combination with supervised machine learning, especially compared to only using Fortify.

Preface

With this report, not only does my pursuing for the Master degree in Computer Science ends, it is also the end of me being a student at TU Delft. The end of an era. I met many people, students and staff alike, which I have fond memories with. Friendships that will last forever, hopefully. We had a lot of fun times. Not only studying, but also watching movies, playing video games, watching and discussing Dragon Ball Super, playing foosball, going to concerts, eating out, etc. Unfortunately, some times were also tough. Failing some courses and retaking them, working all night to meet deadlines, etc.

Although most of the master program I consider a good experience, the Master thesis was definitely the hardest part. Not everything went as smooth as one might hope and on top of that we had to deal with the university closing due to the COVID-19 pandemic. Fortunately, for me, I had many people around me to support me and get me through this thesis. I am therefore very grateful to all my friends and family, especially my parents, for supporting me. There are, however, a few people I would like to thank especially, for they helped make this master thesis to what it is today.

So first of all I want to thank my supervisor, Dr. Annibale Panichella, for all the time he spend on making sure I will have a good thesis. Whenever times got tough, he would always know how we should continue. He thought me so much and made me have a better scientific mindset. I always liked our meetings, especially when we discussed pizza or movies and TV shows.

I also would like to thank Dr. ir. Sicco Verwer, who helped out whenever we got stuck from time to time. Especially in the beginning of the thesis, his point of view and that of Annibale could give us some clarity in how to progress.

Next, I would like to thank Hennie Huijgens, for giving me the opportunity to do a master thesis at ING. He helped us out by making sure we were treated well at ING and were able to do our thesis in the best environment possible.

Last, but not least, I would like to thank my ”partner in crime”, Ka-Wing Man. A fellow Computer Science student and a good friend, who did his master thesis at ING with me at the beginning. We spend a lot of time together, trying to figure out how to do this thesis the best way possible. In doing so, we had a lot of fun and stress. We also worked together on various courses throughout the master program, giving us fond memories and experiences.

Although this is definitely the end of an era, it is also a start of a new chapter in my life. Wherever it will take me, I will always cherish this time and I know now that I am fully prepared for it.

S.P.D. Bisesser
Delft, December 2020

Contents

1 Introduction

The digital world is becoming bigger and more important everyday. On the one hand, people are using more software everyday, sharing more (sensitive) data, while on the other hand more software is developed daily. This is also interesting for adversaries who want to hack into software to steal data or money from people, companies, and organizations or gain control over software.

Cyber attacks can have different types of risks. Business Insider reported in 2018, that billions of user accounts had been compromised in the top 21 biggest data breaches [38]. Adversaries can use these compromised data to sell it on the black market or blackmail people and companies. In July 2015, for instance, a group of adversaries stole the user data of the dating/affairs website Ashley Madison. Adversaries could use this data to blackmail or publicly shame users, for most users would have their affairs in secret [43]. In May 2017, a ransomware cryptoworm named WannaCry was released worldwide and targeted computers with Windows OS [48]. Ransomware infection is another type of risk. The idea behind ransomware is that computers infected with a malware, having all its files encrypted in such a way that the files cannot be accessed and decrypted by the users themselves. If the user paid the hacker within a given time limit, the files would be decrypted and thus releasing the computer as a hostage. If the time limit is expired without payment, the files are destroyed. Data, however, is not the only important digital asset. One of the most recent hacks with big monetary losses happened in 2018. During this hack, 12000 VISA cards were stolen from Cosmos Bank in India, which were used for 15000 transactions, totaling in a loss of $13.4 million [44]. The Carbanak group is one of the biggest hacks concerning banks. They stole almost $1 billion from over 100 banks around the world [36]. Another hack at a financial institute happened in February 2016, when hackers issued a fraudulent transfer of almost $1 billion from the Federal Reserve Bank of New York's account at the Central Bank of Bangladesh. Even though a total of $870 million was refused or halted, $81 million were successfully transferred[1], of which only $18 million has been recovered up till today[2].

To attack a system or application, an adversary needs to exploit a vulnerability in the software. There are many definitions of software vulnerability. Liu et al. [40] defined a software vulnerability by: *"An instance of a mistake in the specification, development, or configuration of software such that its execution can violate the explicit or implicit security policy"*.

Thus, cybersecurity is becoming more important everyday. Looking at figure 1.1, it is clear from the NIST (National Institute of Standards and Technology) study[3] that the number of vulnerabilities in IT has been increasing the last couple of years.

Developing more software unfortunately also means a higher probability of bugs in software. Some of these bugs can be vulnerabilities. More and more software is being developed

[1] https://nypost.com/2016/03/22/congresswoman-wants-probe-of-brazen-81m-theft-from-new-york-fed/
[2] https://newsinfo.inquirer.net/807690/ex-rcbc-branch-manager-free-on-bail
[3] https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time

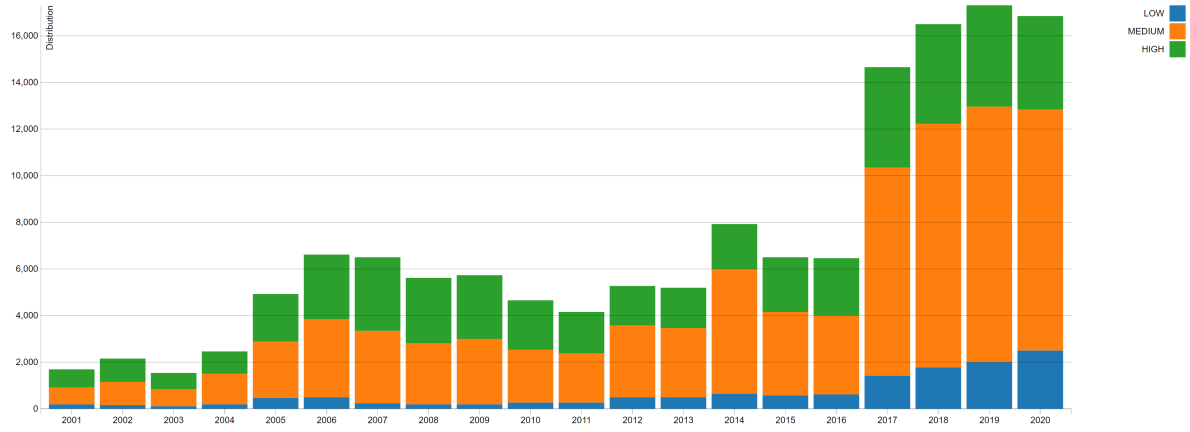

Figure 1.1: The Common Vulnerability Scoring System (CVSS) Severity Distribution Over Time[3]

and thus it is highly likely that more bugs and vulnerabilities in software exists.[4] These vulnerabilities can form risks, some more than others, when attackers try to exploit them. These risks can harm people and companies. Take, for instance, a log-in page on a website. A potential vulnerability on this page could be SQL injection. With SQL injection, an adversary can use the user input, in this case the login text fields, on a web page to execute an SQL statement. By sending an SQL statement through the login form, an adversary can temper with the database. Information about clients and the company could be stolen, deleted, edited, etc. In 2008, the Heartland Payment Systems had a data leak of 134 million credit cards being exposed due to a SQL injection attack. This cost the company $145 million in compensation for fraudulent payments[5]. Another example of SQL injection vulnerability being exploited was during the USA elections of 2016. The personal data of 200000 Illinois voters were breached[6]. The consequences of such vulnerabilities are thus severe.

To overcome these problems, vulnerabilities in software need to be tackled as soon as possible during the development of software, as bug fixes become more costly over time [56]. To ensure that most vulnerabilities are not present after release, developers have to test and check their code for vulnerabilities. This can be done by the developers themselves or by security experts. However, this process is time consuming [31] and can create delays for the development of the rest of the software.

There are various techniques for detecting software vulnerabilities [4]. One such technique is static analysis. This technique identifies weaknesses in the source code before the application is used in the user's environment. This is done by first assessing the code and then applying a rule set or algorithms. After that, a list of warnings (potential vulnerabilities)are generated, which are present in the code.

A major problem with static analysis is the high number of false positives [6]. This means that those warnings, more than 23% of all the warnings in Antunes and Vieira [6] research, are not actual vulnerabilities. Thus all the warnings need to be manually checked by developers and security experts to decide whether or not they need to fix a vulnerability.

Looking at current research, Park et al. [57] suggest to use dynamic information like snapshots of execution environments, logs of dynamically loaded files, and type-based modeling to eliminate false positives in JavaScript Web Applications. Whereas Junker et al. [33] suggests to use SMT (Simultaneous multithreading) to compute infeasible paths, by viewing static analysis as a model checking problem and adding observer automata to exclude such paths.

To overcome this problem, we propose to implement supervised machine learning for pre-

[4] https://www.enisa.europa.eu/publications/info-notes/is-software-more-vulnerable-today
[5] https://www.csoonline.com/article/2130877/the-biggest-data-breaches-of-the-21st-century.html
[6] https://www.econotimes.com/SQL-Injections-Continue-to-Embarrass-Big-Names-690349

dicting true vulnerabilities from static analyzer warnings. Supervised machine learning is a technique that predicts the outcome by using a dataset with a set of features and labels as the input [73]. In this case, the labels are the two classes: true or false vulnerable. By training such a supervised model, we can predict which warnings of a static analysis tool are true vulnerabilities.

Previous research has shown some promising results. Ruthruff et al. [69], for instance, used logistic regression models to accurately predict false positive warnings over 85% of the time on average. We will use different types of supervised machine learning techniques on a dataset from fintech (financial technology) industry in this research.

In this paper, we thus study how to predict true vulnerabilities from static analyzer warnings, given a pool of a large number of different software applications by industry standards.

1.1. Background

This research is done at ING, the largest bank of The Netherlands[7] and active worldwide. They also provided the penetration test data and the corresponding source code that was used in this research.

Within ING, there are many different teams, the so called ”squads”, which all develop their own applications. All of these ”squads” depend on the security officers for deployment. This, so software vulnerabilities are taken care of before deployment of the software. Various tools are used for penetration testing and code reviewing the software. Depending on the risks of the vulnerabilities, certain actions must be taken before the software can be deployed.

To ensure attackers can not exploit applications, software vulnerabilities need to be found and taken care of before applications are released. ING uses penetration testing teams to ensure that their applications do not contain software vulnerabilities.

The ING pentesting process works as follows: Automated static analysis tools such as CheckMarx[8] and Fortify SCA[9] are used to look for potential software vulnerabilities. The way static analysis tools work, is by looking through the source code for vulnerabilities. These vulnerabilities are not only related to the code itself, but can also be related to libraries, network, OS, etc. By comparing the source code with some rules, possible vulnerabilities could be detected. These rules check not only bugs, but also types, style, etc. [15]. An example of a rule would be to check the code for a call to the strcpy() function. The use of this function in C could result in a buffer overflow in case the second argument points to a very long string. The developer teams run these application security testing tools after every ready-to-deploy version of the application. The potential software vulnerabilities are ignored by the development team, if deemed to have low or no risk at all, or an alert is risen for the pentesting team to see if the application is actually vulnerable to being exploited.

Next to code reviewing, using the above two tools, the pentesting teams also use Burp Suite[10] for the actual penetration testing. This is done by the pentesting teams, when a developing team wants to deploy their application. If there are no high risks, they can deploy their software.

1.2. Problem Statement

The security officers at ING have to check all the code that is developed for software vulnerabilities, before they get deployed. This process is time consuming, which causes delays in the development of software, due to the fact that squads can not continue until their code is checked for any major vulnerability.

Besides, most of the validation is done manually, leading to a very laborious and intensive activity. Some static analysis tools are used for detecting software vulnerabilities, but the problem with most of these tools is that they have a high false positive rate [6].

[7] Based on total assets:
https://www.banken.nl/nieuws/20909/ranglijst-grootste-nederlandse-banken-2018
[8] https://www.checkmarx.com/
[9] https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview
[10] https://portswigger.net/burp

Static analysis tools do, however, give more true positives than pentesting tools [71] and tend to have a low number of false negatives [32]. Most of these tools use rule-based solutions. Machine learning could help speed up this process and deal with the manually checking of the high number of false positives.

The problem statement can thus be stated as:

Developing one or multiple models using machine learning that predict true vulnerabilities from static analyzer warnings to speed up the security check and software deployment within ING.

1.3. Research Questions

We investigate three research questions that steer our research.

First, different classifiers need to be compared to see how each of them perform. These different classifiers use supervised learning, meaning that the training data is labeled. This will also help understand which classifiers work better for finding true vulnerabilities in a dataset that contain static analyzer warnings of various types of vulnerabilities. The first research question is thus:

RQ1: *How accurate are supervised machine learning methods in detecting true vulnerabilities from static analyzer warnings using a dataset from ING?*

The dataset can be split into various granularity levels. Both class level and method level are available. These two datasets have different metrics and values. Given the various classifiers, it is interesting to know at which granularity level these classifiers are more accurate. Therefore the second research question will be:

RQ2: *How do classification methods perform at different levels of granularity on a dataset from ING?*

Various types of vulnerabilities exist within the dataset. Although they are mixed and used as binary (vulnerable or not), it is interesting to explore how this model compares to models built with only one vulnerability type. To explore if it is better to combine the vulnerabilities or to make a model for each vulnerability, the last research question states:

RQ3: *How accurate is a model built on a dataset with various types of vulnerabilities compared to models built on datasets per vulnerability type?*

1.4. Contributions

Looking into the current research, it is lacking real-world data usage with machine learning for predicting software vulnerabilities. Most research uses open source data, which is not very representative for big companies like ING. Next to that, software vulnerability predictions are usually done on method level instead of class level and the models are built per project [27]. This research will help overcome these problems and unknowns. Thus, the main contributions of this research are:

1. **Comparing different classifiers:** First, this research will compare the performance of various classifiers. This will help give an insight on the state-of-the-art supervised machine learning techniques, when applied to predict true vulnerabilities from static analyzer warnings, to see how well they perform and see if they are worth it to implement in a corporate setting.

2. **Comparing different granularity levels:** Next, this research will compare the performance of models built at class level with models built at method level. This will help in choosing the right level of granularity when performing true vulnerability from static analyzer warnings prediction.

3. **Comparing different models based on vulnerability type:** Different models will be created for various vulnerability types. The performance of these models will then be compared to a model with all vulnerabilities combined. This will give an insight into whether it is best to focus on finding certain vulnerability types or to find them all using just one model.

4. **Cross-project closed data performance of detecting true positives among different warning types using machine learning:** Finally, this research uses a dataset containing data from various projects from ING. These projects are anonymised and mixed together into one big dataset. Thus, instead of looking at just one project for one model, a model is built from various projects (cross-project). The performance of this model is determined in this research.

1.4.1. Fortify Overview

Figure 1.2 is an overview of the Fortify process combined with supervised machine learning. The source code is deployed to Fortify and checked for vulnerabilities. Pentesters will label these vulnerabilities as true or false. Both the source code as well as the vulnerability data from Fortify are extracted from Fortify. From the source code, code metrics are created. These code metrics are combined with the vulnerability data into one dataset. After preprocessing the dataset, it can be used by supervised machine learning algorithms. These algorithms will create models that detect true vulnerabilities from static analyzer warnings.

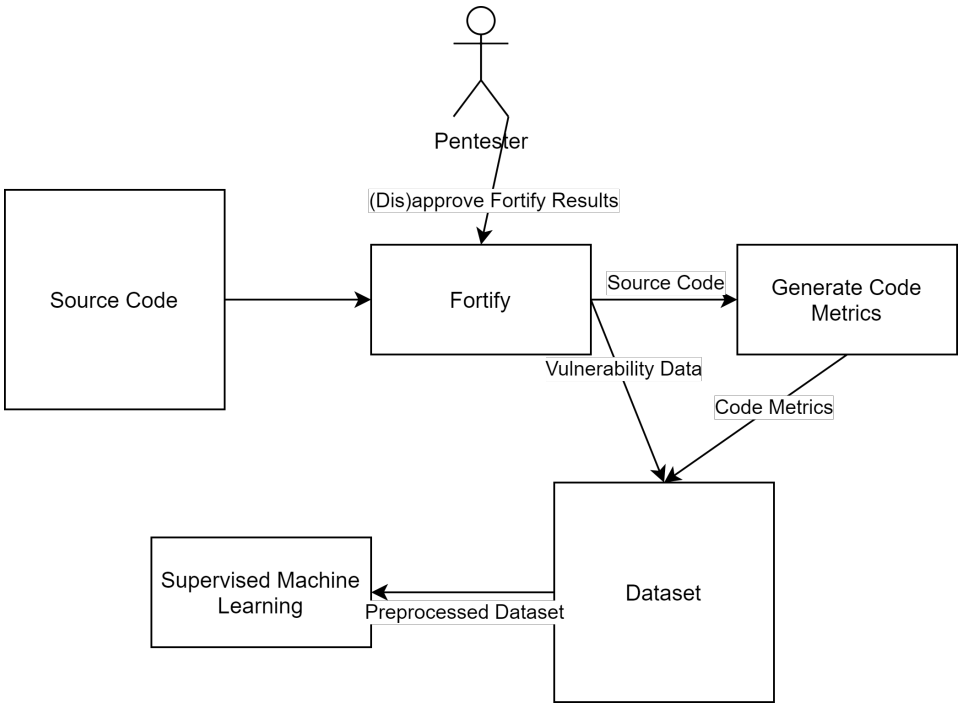

Figure 1.2: Schema of Fortify combined with supervised machine learning

1.5. Report Overview

The report consists of the following chapters.

First, we discuss the related work in chapter 2. In this chapter, we state the current most common software vulnerabilities and vulnerability detection methods. Next to that, the state of the art in vulnerability detection tools and the current state of research on the topic of software vulnerability prediction using software metrics are explained. Last, we discuss benchmark metrics and other approaches for decreasing software vulnerabilities. After that, we describe and visualise the dataset in chapter 3. How the data was retrieved and what it consists of are also explained in this chapter. In chapter 4, we illustrate the research design. This chapter will explain the experiment setup. The research results are shown in chapter 5. We discuss the results in chapter 6. This chapter will connect the results with the research questions and hypothesis. After that, we discuss the threats to validity in chapter 7. Both internal and external threats are mentioned and how they are handled. A conclusion is drawn in chapter 8. Finally, we discuss future work in section 8.1. This final section will look into how to improve this study.

2

Related Work

This chapter will look into the related work for predicting software vulnerabilities. First, the causes of software vulnerabilities are explained. Then, the most common software vulnerabilities are discussed. Next, the methods for detecting software vulnerabilities are explained and the state of the art is discussed for software vulnerability detection tools. After that, the current state of research on the topic of software vulnerability prediction using software metrics is summarised. Finally, benchmark metrics and other approaches for decreasing software vulnerabilities are discussed.

2.1. Software Vulnerabilities Causes

To understand which software vulnerabilities exists and how to detect them, it is important to understand what may cause software vulnerabilities.

According to Ping et al.[61] there are four causes which can create software vulnerabilities when building software.

Design
When designing software, designers may leave some defects in the software logical workflow because of negligence and inconsiderateness. These faults may cause crashes and become an exploitation point for attackers and are hard to fix. Because of the high risks, much attention is given to assure that these vulnerabilities do not exists. Thus the number of vulnerabilities by design are relatively small.

Encoding and Test
Because programs are becoming larger and larger, programmers only understand their part of the code. The program flow of their part may conflict with predefined security policies. When integrating all the code into one large system, vulnerabilities may then occur.

Operating Environment
Software is usually tested in different environments. However, when software is released, it will be run in multiple different software and hardware environments, which cannot be all tested. Software vulnerabilities will then inevitably appear in time.

Software Patches
To fix software vulnerabilities, patches will be released which will effect the whole system. These patches may thus cause new software vulnerabilities in the system.

2.2. Most Common Software Vulnerabilities

The Open Web Application Security Project (**OWASP**)[1] is an open community, dedicated to enabling organizations to conceive, develop, acquire, operate, and maintain applications that can be trusted. According to OWASP, attackers can use different paths to attack businesses. Each of these paths are potential risks, with different outcomes from zero consequences to putting a business out of business. Figure 2.1[2] shows such a path.

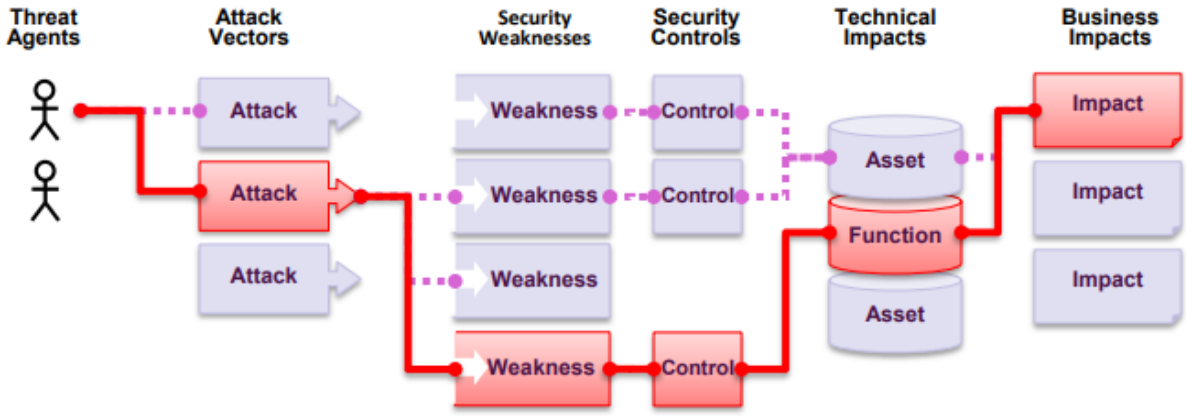

Figure 2.1: Application Security Risks Path

OWASP has studied vulnerabilities from hundreds of organizations and over 100000 real-world applications and APIs. From that study, the software vulnerabilities in table 2.7 are considered the top 10 most common software vulnerabilities as of 2017 for web applications.[3]

The column exploitable shows how exploitable the vulnerability is. Security weakness shows how prevalence and detectable the vulnerability is. Last, technical impacts show the level of impact and thus the consequences an attack on that vulnerability has. The meaning of each value and color is described in table 2.8.

For each of the risk aspect levels, three is considered high, two medium and one low.

Common Weakness Enumeration (**CWE**) created a list in 2011 of the top 25 most common vulnerabilities[4]. Comparing their top 10 with the OWASP top 10, we see that they include Classic Buffer Overflow, Unrestricted Upload of File with Dangerous Type, and Reliance on Untrusted Inputs in a Security Decision.

Atashzar et al. [7] examined common software vulnerabilities and described countermeasures for them. They also discuss common hacking tools and tools for security improvement. The common software vulnerabilities are based on OWASP 2010. The differences can be seen in table 2.1.

[1] https://www.owasp.org/index.php/About_The_Open_Web_Application_Security_Project
[2] https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf
[3] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_2017_Project
[4] https://cwe.mitre.org/top25/

OWAPS TOP 10 - 2007	**OWAPS TOP 10 - 2010**
A1 - Cross Site Scripting (XSS)	A1 - Injection
A2 - Injection Flaws	A2 - Cross Site Scripting (XSS)
A3 - Malicious File Execution	A3 - Broken Authentication and Session Management
A4 - Insecure Direct Object Reference	A4 - Insecure Direct Object References
A5 - Cross Site Request Forgery (CSRF)	A5 - Cross Site Request Forgery (CSRF)
A6 - Information Leakage and Improper Error Handling	A6 - Security Misconfiguration (NEW)
A7 - Broken Authentication and Session Management	A7 - Insecure Cryptographic Storage
A8 - Insecure Cryptographic Storage	A8 - Failure to Restrict URL Access
A9 - Insecure Communications	A9 - Insufficient Transport Layer Protection
A10 - Failure to Restrict URL Access	A10 - Invalidated Redirects and Forwards (NEW)
OWAPS TOP 10 - 2013	**OWAPS TOP 10 - 2017**
A1 - Injection	A1 - Injection
A2 - Broken Authentication and Session Management	A2 - Broken Authentication and Session Management
A3 - Cross-Site Scripting (XSS)	A3 - Sensitive Data Exposure
A4 - Insecure Direct Object References	A4 - XML External Entity (XXE)
A5 - Security Misconfiguration	A5 - Broken Access Control
A6 - Sensitive Data Exposure	A6 - Security Misconfiguration
A7 - Missing Function Level Access Control	A7 - Cross-Site Scripting (XSS)
A8 - Cross-Site Request Forgery (CSRF)	A8 - Insecure Deserialization (NEW)
A9 - Using Components with Known Vulnerabilities	A9 - Using Components with Known Vulnerabilities
A10 - Invalidated Redirects and Forwards	A10 - Insufficient logging and monitoring (NEW)

Table 2.1: OWASP top 10 2007-2017[5]

Tables 3.2 and 3.3 show the top 10 vulnerabilities found in the class-level and method-level dataset. These are the vulnerabilities that occur most often in the dataset. Table 2.2 describes all these vulnerabilities according to the documentation of Fortify[6].

Vulnerability Type	**Description**
System Information Leak: Internal	Disclosing the IP addressing scheme of the internal network can allow attackers to discover internal systems and expand the attack surface.
J2EE Bad Practices: Threads	Thread management in a web application is forbidden in some circumstances and is always highly error prone.
Log Forging	Writing unvalidated user input to log files can allow an attacker to forge log entries or inject malicious content into the logs.
Password Management: Hardcoded Password	Hardcoded passwords can compromise system security in a way that is not easy to remedy.
Path Manipulation	Allowing user input to control paths used in file system operations could enable an attacker to access or modify otherwise protected system resources.
Rare Condition: Singleton Member Field	Servlet member fields might allow one user to see another user's data.
Unreleased Resource: Streams	The program can potentially fail to release a system resource.
Dynamic Code Evaluation: Unsafe Deserialization	Deserializing user-controlled object streams at runtime can allow attackers to execute arbitrary code on the server, abuse application logic, and/or lead to denial of service.
Missing XML Validation	Failure to enable validation when parsing XML gives an attacker the opportunity to supply malicious input.
System Information Leak	Revealing system data or debugging information helps an adversary learn about the system and form a plan of attack.
Insecure Randomness	Standard pseudorandom number generators cannot withstand cryptographic attacks.
Header Manipulation	Including unvalidated data in an HTTP response header can enable cache-poisoning, cross-site scripting, cross-user defacement, page hijacking, cookie manipulation or open redirect.
SQL Injection	Constructing a dynamic SQL statement with input that comes from an untrusted source could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.

Table 2.2: Top vulnerabilities in the dataset[6]

[5] https://www.incibe-cert.es/en/blog/owasp-publishes-top-10-2017-web-application-security-risks
[6] https://vulncat.fortify.com/en/weakness

2.3. Vulnerability Detection Methods

There are quite a few methods to detect several different vulnerabilities in software. This can be done statically (before the code is compiled) or dynamically (after the code is compiled and up and running). Table 2.3 shows the differences between static and dynamic testing.

Characteristic	*Dynamic detection*	*Static detection*
Determine the position of overflow vulnerabilities	No	Yes
Proceed in early period	No	Yes
Need input	Yes	No
Need source code	Partly need	Yes
Have relation to programming language	No	Yes
Precision of detection	High	General
Amount of misinformation	Low	High

Table 2.3: Comparison of static and dynamic detection[61]

Amankwah et al.[4] states five different vulnerability detection methods:

2.3.1. Fuzzing

Fuzzing [4] [39] [42] uses a random or invalid input to test an application and outputs the unexpected behavior, errors, and possible vulnerabilities in the program. For example, if you have a program where the user can choose between three options, 0, 1, and 2. By transmitting a different option, like 3, the program may crash in case the default switch case in not implemented securely. This could lead to buffer overflow, ddos, etc.[7] A fuzzer is able to find such a bug automatically.

Black-box fuzzers generate input data without knowledge of the application details (like source code, structure, etc.), whereas White-box fuzzers have full knowledge of the application details. Random fuzzing is the most simplest way of fuzzing. It streams a random input to the program under test. Mutation-based fuzzing uses previous data to generate new data, making it a more efficient fuzzer than random fuzzing. Generation-based fuzzing has higher coverage than random fuzzing by creating input based on some specifications. For instance, having valid SQL statements as input instead of just random text. The control flow of the program is used as a direction for the fuzzing in Direction-based fuzzing.

2.3.2. Web Application Scanners

Web Application Scanners [4] scan applications on the web for security vulnerabilities. With white box testing, the source code of the application is analysed, while black box testing uses fuzzing to detect vulnerabilities. By going through the web pages with malicious input, the web application scanners check which response the application gives and analyses that. By using web application scanners in the testing phase, it must identify vulnerabilities, report how to carry out the vulnerabilities, and have a low false positive ratio.

2.3.3. Static Analysis Techniques

Static Analysis Techniques [4] identify weaknesses in the source code before the application is used in the user's environment. This is done by first assessing the code. Then applying a rule set or algorithms, which is also called inference. Last, a list of warnings (potential vulnerabilities) are generated, which are present in the code.

There are many different categories of static analysis tools. Tools can focus on type checking, style checking, program understanding, program verification, property checking, bug finding, security review, etc. Most static analysis tools focus on various of these categories. Static analysis tools that target security mostly, are often a hybrid of property checkers and bug finders. Fortify is such a static analysis tool, that focus on security mainly. Due to the

[7] https://owasp.org/www-community/Fuzzing

high false positives, static analysis that focus on security requires human review of the static analyzer warnings for best results [15].

2.3.4. Brick

Brick [4] stands for Binary Run-time Integer Based Vulnerability Checker and checks integer based vulnerabilities at runtime by converting the binary code to an intermediate representation VEX on Valgrind [52], intercepting integer related statements at runtime and recording the necessary information and detecting and locating vulnerabilities with a set checking scheme.

2.3.5. CRED

CRED C Range Error Detector [4] finds buffer overruns attacks. Where Dynamic Buffer Overrun Detector lacks the power to protect against all buffer overrun attacks, break existing code, and produce too high overhead, CRED proved to be the only tool to guard against 20 buffer overflow attacks [84].

2.3.6. Manual Testing

Manual Testing is observing the state and output of a program, after manually constructing an input for the program [83]. Depending on the skills, knowledge and experience of the analyst, manual testing can have a high accuracy.

Liu et al. [40] state the advantages and disadvantages of several techniques. They also included a technique called Vulnerability Discovery Models (VDM), a way to predict vulnerabilities using previous discovery event data. Table 2.4 shows these advantages and disadvantages.

Technique	Advantages	Disadvantages
Static analysis	No requirements of executing target programs; Sound to describe properties of programs; easily integrated into the whole software development circle; Able to find most of implement bugs before the release of software.	Not precise enough to describe program properties; High false positive; Need human to verify the results and can not be entirely automatic; Most of such methods depend on source code; Unable to detect design bugs; Unable to detect vulnerabilities caused by configurations or environment.
Fuzzing	Simple idea; Easy to be understood; No false positive; High automation degree.	High randomness; High false negative; Low degree of generalization and long construction circle of Fuzzing tools.
Penetration Testing	No false positive and vulnerability discovery equal to vulnerability exploit; Based on practical user environments; Able to expose vulnerabilities hard to be detected by other tools; Take social engineering factors into consideration.	Heavily depend on human and the results depend to a great extent on testers' abilities, skills and experience; May do harm to the tested system.
VDMs	A new method to make use of the discovered vulnerabilities; In theory, able to predict the rate of vulnerability discovery and the total amount of vulnerabilities in a single software. Help to assess threats.	Some assumptions some VDMs base on need to be validated; Only apply to a single software; Lack general valid VDMs.

Table 2.4: Advantages and disadvantages of vulnerability detection techniques [40]

Austin and Williams [8] found that systematic manual penetration testing is more effective in finding vulnerabilities (and design flaws) than exploratory manual penetration testing. Static analysis found different types of vulnerabilities and automated penetration testing is the most efficient way of finding vulnerabilities. However, one cannot rely on static analysis and automated penetration testing alone, for that would leave many vulnerabilities undiscovered.

Ghaffarian and Shahriari [27] conducted a survey on machine learning and data mining techniques and algorithms for predicting software vulnerabilities. Conventional approaches consists of the following categories: Static, Dynamic and Hybrid Analysis. Software Penetration Testing, Fuzz-Testing and Static Data-Flow Analysis and more established approaches in the industry. There are four categories, in which previous work in the field of software vulnerability analysis and discovery, can be categorized: Vulnerability Prediction Models based on Software Metrics, Anomaly Detection Approaches, Vulnerable Code Pattern Recognition, Miscellaneous Approaches.

As is clear from the related research, there are many different methods for detecting software vulnerabilities. All of these techniques have advantages and disadvantages, thus there is no holy grail for detecting software vulnerabilities. Using static analysis, like Fortify, is therefore not a bad idea, despite the high number of false positives. By focusing on lowering that number using machine learning, as our research does, we might make static analysis as a tool for detecting software vulnerabilities more useful.

2.4. State-of-the-Art Tools for Vulnerability Detection

At ING pentesters manually check the code for software vulnerabilities. They do however, use some tools.

For penetration testing, the pentesters at ING use Burp Suite. When developers want to deploy their software, the pentesters first use Burp Suite to pentest the code. If there are no high risks, the code is approved and can be deployed.

For code reviewing, both the developers and the pentesters use Checkmarx and Fortify. Developers first use the tools for code reviewing. The results are then shared with the pentesters, who then check the results. If there are any software vulnerabilities they are reported back to the developers. Together they try to fix any vulnerability.

The various tools used by ING are not the only tools available. Table 2.5 lists the current state-of-the-art tools for software vulnerability detection.

Source Code Analysis Tools[8] [9]	Vulnerability Scanning/Dynamic Analysis Tools[10] [11]	Fuzzing Tools[12]
heckmarx	Burp Suite	American Fuzzy Lop
Fortify	Netsparker	Peach
SonarQube	Acunetix	Radamsa
RipsTech	Probely	
PVS-Studio	Indusface WAS	
Kiuwan	Zed Attack Proxy	
Reshift	Sqlmap	
Veracode	Canvas	
Parasoft	Social Engineering Toolkit	
Coverity	BeEF	
Cast		
Codesonar		
Microfocus		

Table 2.5: State-of-the-art tools

Much research has been done in finding tools to detect software vulnerabilities. Given the fact that the tools at ING do not perform optimal, for their results have a high amount of false positives, we will look into some of that research to determine how well state-of-the-art tools perform in different environments.

Awang and Manaf [9] uses a combination of black-box automatic detection and manual penetration testing to filter out false positives. IBM Rational AppScan is used as a black box. After checking the detected vulnerabilities, there were no false positives found.

Medeiros et al. [45] creates an approach that uses data mining to predict false positive vulnerabilities after an initial step that uses taint analysis to flag candidate vulnerabilities. The tool also corrects the code by inserting fixes, currently sanitization and validation functions.

Bau et al. [10] studied 8 black-box web application scanners for vulnerabilities. They looked at which class of vulnerability they detect, how effective they are against these vulnerabilities, and the relevance of target vulnerabilities to vulnerabilities found in the wild. They found that Cross-Site Scripting, SQL Injection, other forms of Cross-Channel Scripting, and Information Disclosure are the most prevalent classes of vulnerabilities. As good as they

[8] https://www.softwaretestinghelp.com/tools/top-40-static-code-analysis-tools/

[9] https://www.owasp.org/index.php/Source_Code_Analysis_Tools

[10] https://www.softwaretestinghelp.com/penetration-testing-tools/

[11] https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools

[12] https://www.owasp.org/index.php/Fuzzing#Fuzzing_tools

are for finding historical vulnerabilities and textbook cases of Cross-Site Scripting and SQL Injection, there is room for improvement in other classes of vulnerabilities, such as advanced and second-order forms of XSS and SQLI, other forms of Cross-Channel Scripting, Cross-Site Request Forgery, and Malware Presence.

Nunes et al. [55] [53] [54] used four different scenarios to test combining different Automated Static Analysis Tools (ASAT) to improve the overall detection of SQLi and XSS vulnerabilities. Five different ASATs (RIPS, Pixy, phpSAFE, WAP, WeVerca) were tested under two datasets (Wordpress plugins and PHP synthetic test cases). They found that combining different ASATs doesn't necessarily increase the overall detection over one ASAT and that it depends on the scenario which tools are most useful to combine.

Algaith [2] uses five different SATs (phpSAFE, RIPS, WAP, Pixy, and WeVerca) to see which combination of SATs gives the best results in detecting SQli and XSS vulnerabilities. Combining phpSAFE with WAP gave the best improvement, with higher detection and lower false positives for a setup where an alarm is raised when any SAT finds something. With a setup where the alarm is raised only when all SAT find the vulnerability, both TP and FP increase. In a setup with the majority finding a vulnerability, TP improves but FP deteriorates.

Curphey and Arawo [18] introduced a framework to decide which tools to use for detecting vulnerabilities in web application and websites. The security framework covers the following aspects: Configuration management, Authentication, Authorization, Data protection, User and session management, Data validation, Error handling and exception management and Auditing and event logging. The following types of tools are listed: Source-code analyzers, Web application (black-box) scanners, Database scanners, Binary analysis tools, Runtime analysis tools, Configuration analysis tools, Proxies, and Miscellaneous tools (mix of white and black box tools). Before deciding on a tool, the tool needs to be tested on an application or website for which the tester knows the vulnerabilities.

Shahriar and Zulkernine [72] suggests four ways to mitigate vulnerabilities in software: Program security testing, Static analysis, Monitoring and Hybrid analysis.

Yan et al. [85] created ExploitMeter, a fuzzing framework to quantify software exploitability. ExploitMeter is a Bayesian reasoning engine, which first uses machine learning to predict exploits in software, using static features. Then it uses various dynamic fuzzers to update its findings[70].

From the previous work, it is clear that many different tools and methods exist for detecting software vulnerabilities in source code. Depending on the dataset, tools might perform good or bad. Most research is done on open source data. This research will look into different supervised learning models to find true vulnerability warnings from the static analysis tool Fortify. Comparing these machine learning models and using industrial data, will add to the research of detecting software vulnerabilities.

2.5. Software Vulnerability Prediction using Machine Learning

2.5.1. Software Metrics

Suresh et al. [79] looked into the effectiveness of software metrics for object-oriented systems. They found that Cyclomatic complexity is best used for assessing the complexity of the system and estimate the number of test cases needed to achieve maximum code coverage. Chidamber and Kemerer metrics are best used for indicating the fault sensitivity of the system. R. C. Martin's metrics is used to compare the systems to the ideal models of abstraction and dependency. In our study Chidamber and Kemerer metrics are thus best suited, compared to the other two software metric suites. This because software vulnerabilities are types of software faults that can be exploited.

Moshtari et al. [50] used complexity metrics on both within-project (1 project with different releases) and cross-project (5 different projects) open source data to predict vulnerabilities. Using 8 different classifiers, they found that 92% of vulnerable files were detected on the within-project dataset, whereas 70% were detected on cross-project dataset. The within-project had a false positive rate of 0.12% and the cross-project dataset had a false positive rate of 26%, thus showing that complexity metrics are good predictors for vulnerabilities between different projects and different releases of the same project.

Shin and Williams [74] used nine code complexity metrics and binary logistic regression to predict vulnerable functions. This was done on six versions of Mozilla JSE. Nesting complexity was the best distinguishing factor among the nine complexity metrics in JSE. Prediction of vulnerabilities from source code using complexity metrics is a feasible approach with fewer false positives, however, still misses many vulnerabilities.

Catal and Diri [12] found that most research is done on method level. However, class level shows above acceptable levels and should be used more in research. Mostly public datasets are used.

Harer et al. [30] uses lexicon based features. They then compare methods applied directly to the source code with methods applied to artifacts extracted from the build process. This resulted in source-based models performing better. They also compare more traditional models, such as random forests, with the application of deep neural network models. Combining the features learned by tree-based models with deep models turns out to result in the best performance. Their highest performing model has an area under the ROC curve of 0.87 and an area under the precision-recall curve of 0.49. This research will use random forest. Deep learning is out of the scope of this research, however.

In Alves et al. [3], a large dataset to evaluate several state-of-the-art vulnerability prediction techniques (Log reg, Bayesian Network, Decision Tree, Random Forest, Naive-Bayes) was used. The dataset contains information of 2186 vulnerabilities from five open source projects. The results show that the dataset can be used to determine the best techniques and that some of the techniques can predict nearly all vulnerabilities present in the dataset, although with very low precision. Recall, precision, and accuracy are not the most effective to characterize the effectiveness of these tools. Although other studies show that these evaluation metrics are effective.

In Walden et al. [82], a public dataset was provided, containing 223 vulnerabilities from three web applications. This dataset was used to compare vulnerability prediction models based on text mining with models using software metrics as predictors, using Random Forest and cross validation. They found that for all three applications text mining models had a higher recall than software metrics based models. Although this research focus on software metrics, text metrics could be added in future research.

Ghaffarian and Shahriari [27] surveyed recent work on vulnerability prediction models based on software metrics. Table 2.6 summaries their findings. The table shows for each research the type of metrics they used, the granularity of the code, whether one or multiple projects were used, and how the vulnerabilities were found for labeling. They consider the following topics lacking in current work and suggest these to be important for future work:

- Imbalance class data.
- A semi-automated framework for vulnerability detection, gives higher precision and recall than gathering information available via public advisories and vulnerability databases.
- Cross-project studies in the field of vulnerability prediction models are few.
- Most studies in the field of vulnerability prediction based on software metrics report poor results. One possible conclusion is that traditional software metrics are not suitable indicators for software vulnerabilities. Henceforth, defining security-specific metrics, such as the Security Resources Indicator (SRI) proposed by Doyle and Walden (2011) [21] is another area for future studies. Due to time constraints, security metrics were not included in this study. The unique nature of this study (cross-project, closed source, imbalanced dataset, etc.) makes it interesting to see how well code metrics based datasets perform. Especially the fact that this study is done at ING. Using an industry standard dataset based on code metrics is interesting enough to study.
- An uncharted area in this field, is using deep learning methods for vulnerability prediction. Deep learning is however not used in this study, for it is expensive to use. It needs much power to perform and a larger set of data points. This study will thus use more related machine learning techniques, however deep learning should be considered for future research.

Paper	Metrics	Granularity	Within/ Cross Project	Vulnerability info
(Zimmerman et al. 2010)[89]	Code-chrun, complexity, coverage, dependency, organizational	Binary modules	Within-project	Public advisories
(Meneely and Williams 2010)[46]	Developer-activity	Source file	Within-project	Public advisories
(Doyle and Walden 2011)[21]	Code complexity, Security Resources Indicator	Source file	Within-project	Tool-based detection
(Shin and Williams 2013)[76]	Complexity, code-churn, fault-history	Source file	Within-project	Public advisories
(Shin and Williams 2011)[75]	Code complexity, dependency network complexity, execution complexity	Source file	Within-project	Public advisories
(Shin et al 2011)[77]	Complexity, code-churn, developer-activity	Source file	Within-project	Public advisories
(Moshtari et al. 2013)[50]	Unit complexity, coupling	Source file	both	Self-developed detection
(Meneely et al. 2013)[47]	Code-churn, developer-activity	Code commits	Within-project	Public advisories
(Bosu et al. 2014)[11]	Developer-activity	Code commits	Within-project	Public advisories
(Perl et al. 2015)[60]	Code-churn, developer-activity, GitHub meta-data	Code commits	Cross-project	Public advisories
(Walden et al. 2014)[82]	Code complexity	Source file	both	Public advisories
(Morrison et al. 2015)[49]	Code-chrun, complexity, coverage, dependency, organizational	Binary modules, source file	Within-project	Public advisories
(Younis et al. 2016)[87]	Code complexity, Information Flow, Functions, Invocations	Functions	Within-project	Public advisories

Table 2.6: Summary of Recent Works on Vulnerability Prediction Models Based on Software Metrics [27]

As suggested by Ghaffarian and Shahriari [27], this research will use an industrial imbalanced dataset to predict true vulnerabilities from Fortify warnings. The dataset consists of code metrics from various different software projects, thus we will perform a cross-project study. Next to that, we will look and compare both on class and method level.

2.5.2. Feature Selection

According to both Chandrashekar and Sahin [13] and Khalid et al. [34] it is hard to determine the best methods or algorithms for feature selection. This because it really depends on the dataset and the classifier that is going to be used. Khalid et al. [34] found that feature selection methods that handle elimination of both redundant and irrelevant features at once are much more robust and beneficial for the learning process. mRMR (Minimal Redundancy and Maximal Relevance) [59] is a method that eliminates both redundant and irrelevant features. Yun and Yang [88] compared 9 different feature selection methods and found that mRMR was the most powerful and had the most stable performance overall.

2.5.3. Class Imbalance

There are three common sampling techniques for the imbalanced class problem[37]:

- **Random Over-Sampling (ROS):** Duplicate minority class instances randomly.
- **Random Under-Sampling (RUS):** Remove majority class instances randomly.
- **Synthetic Minority Over-Sampling Technique (SMOTE)** [14]**:** Creates new minority class instances by interpolating between several minority class instances that lie relatively close to each other.

Fernández et al. [23] compared the performance of these three techniques using Big Data with MapReduce framework. Both RUS and ROS have better classification results than SMOTE, with ROS performing better than RUS.

Rodriguez et al. [67] used software metrics for their datasets to predict software defects. Comparing ROS, RUS, and SMOTE, ROS performed the worst out of the three. On average, RUS and SMOTE performed rather equally. Ensemble methods, like SMOTEBoost and RUSBoost, perform best. These methods, however, do not provide inside information on the decision making, which can help identify important metrics. Next to that, ensemble methods are computationally costly.

García et al. [26] found that on average, oversampling techniques outperform undersampling techniques, when the data is highly imbalance. Using 17 real datasets and 8 different classifiers, they also found that the classifier has a poor influence on the effect of sampling.

Elrahman and Abraham [22] gives an overview of research on techniques for the imbalance class problem. From the various research, it is clear that different techniques work best in different circumstances.

Given the previous research, we see that oversampling techniques perform better than undersampling techniques and SMOTE performs better than ROS. Thus, we will use SMOTE in combination with the various classifiers to deal with the imbalance class problem of the datasets. Next to SMOTE, we will also use Weka's ClassBalancer[13].

[13] https://weka.sourceforge.io/doc.dev/weka/filters/supervised/instance/ClassBalancer.html

2.5.4. Machine learning techniques

Chug and Dhall [17] found that Random Forest performs best using various code metrics. It scored the highest accuracy, Recall, F-Measure and ROC, and had the lowest Root Mean Square error. J48 and Naive Bayes are the second and third, respectively. Next to these performance evaluators, they also used Precision, Mean absolute error (MAE), and Mean absolute error (MAE).

Muhammad and Yan [51] state that Neural Networks and Support Vector Machine (SVM) perform better when dealing with continuous and multidimensional features. These, however, require a large dataset, whereas a smaller dataset suffices with Logic-based systems. These systems also tend to perform better when dealing with discrete/categorical features.

Aleem et al. [1] compared various classifiers using code metrics, in order to predict bugs in code. They found that SVM, Multi-layer Perceptron (MLP), and bagging performed best. For performance comparison they used accuracy, Precision, Recall and F-Measure.

Prasad et al. [63] looked at different supervised learning techniques and how they are used in research for predicting software defects using code metrics. Bayesian Network, Ensemble Method, Random Forests, SVM and Decision Tree were all used in different studies. They found that each of these techniques can perform best, depending on different datasets.

Thus, the following supervised classifiers will be used: J48, Random Forest, Naive Bayes, SVM, MLP and Bagging.

2.6. Other Approaches

Next to predicting and detecting software vulnerabilities, there has also been research in other approaches for decreasing software vulnerabilities. Tevis and Hamilton [80] suggests that a shift in the programming paradigm from imperative programming to functional programming might solve any existence of software vulnerabilities. They list some tools for vulnerability detection and some common types of vulnerabilities.

Yang et al. [86] created an approach called *Priv* to improve static application security testing (SAST) techniques by decreasing false positives. Priv helps prioritize developers' quality-assurance efforts, identifies actionable warnings by locating relevant warnings for database- and attribute related warnings, and improves the current remediation pages in a commercial SAST product with customized remediation which includes the customized fix suggestion for each detected vulnerability warning. Priv focuses mainly on visualising the warnings in groups and providing automatically-generated fix suggestions. This is different from this research, which focuses on reducing the false positives and workload of pentesters and developers by using various supervised machine learning techniques. Priv also only focuses on database-related and attribute-related vulnerability warnings, whereas this research looks at multiple vulnerability types.

Vulnerability	Description	Exploitable	Security Weakness Prevalence	Security Weakness Detectability	Technical Impacts
Injection	Attacker uses user input to execute code.	3	2	3	3
Broken Authentication	Attacker can pretend to be someone else and access their data.	3	2	2	3
Sensitive Data Exposure	Attackers can access sensitive data and read the data.	2	3	2	3
XML External Entities (XXE)	Attackers can upload or exploit XML.	2	2	3	3
Broken Access Control	Attackers can manipulate sessions or urls to gain unauthorised access.	2	2	2	3
Security Misconfiguration	Attackers can exploit unpatched flaws, default pages, etc.	3	3	3	2
Cross-Site Scripting (XSS)	Attackers can run scripts on website visitors browser.	3	3	3	2
Insecure Deserialization	Attackers can use untrusted data to abuse an application.	1	2	2	3
Using Components with Known Vulnerabilities	Attackers can exploit known vulnerabilities that are used in components of the application.	2	3	2	2
Insufficient Logging & Monitoring	Attackers can attack indefinitely without being detected.	2	3	1	2

Table 2.7: Top 10 most common vulnerabilities

Value	Color	Exploitable	Security Weakness Prevalence	Security Weakness Detectability	Technical Impacts
1	Yellow	Difficult	Uncommon	Difficult	Minor
2	Orange	Average	Common	Average	Moderate
3	Red	Easy	Widespread	Easy	Severe

Table 2.8: Legend for table 2.7[14]

[14] https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_Application_Security_Risks

3

The Datasets

To speed up the overall software deployment process within ING, we first need to create datasets. These datasets are going to be used by the machine learning models for training and testing.

Figure 3.1 shows the pipeline for the prediction of true vulnerabilities. It shows that we first gather the data from Fortify and extract the code metrics from that data. This data consists of the source code, vulnerability type, line of vulnerability, and the vulnerable label. After that, we normalise the data and select some features for further classification. Last, we apply sampling to overcome class imbalance and train the model to classify the data.

This chapter describes and visualises the dataset. How the dataset is built and what it consists of are both explained. We discuss all the steps from figure 3.1 in this chapter, except for the classification part.

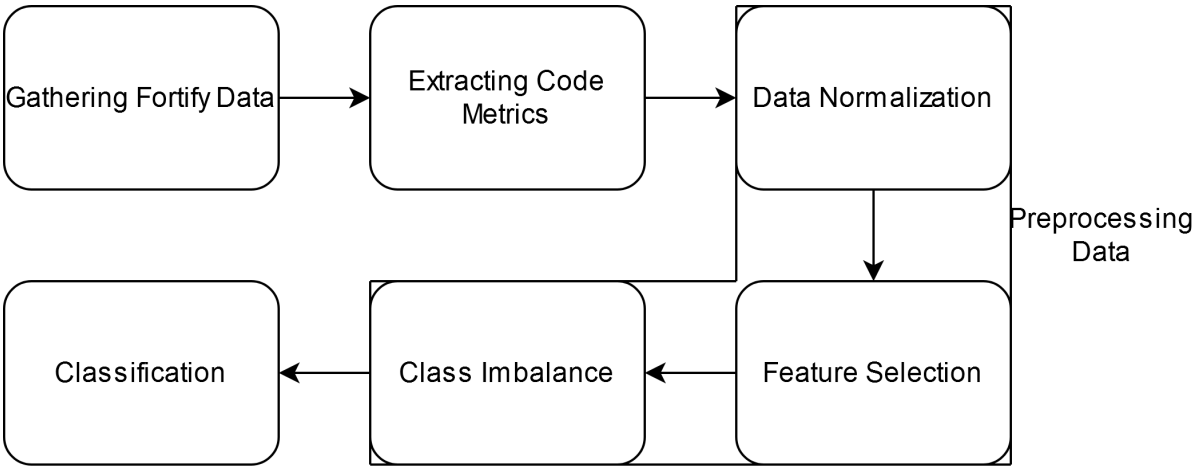

Figure 3.1: Pipeline of true vulnerabilities prediction

3.1. Data Origin

Various teams develop all different kinds of software within ING. When they deem their software ready for deployment, the software needs to be checked for security issues. Within ING, there are several ways software is checked for vulnerabilities. Next to pentesting, code review is an important part of this process. Software developing teams upload their Java code to Fortify SCA[1] for code checks. Fortify SCA is a static analyzer for mainly Java code, which checks the source code for vulnerabilities.

The process is as follows. When the teams upload their projects to Fortify SCA, Fortify SCA identifies classes with potential software vulnerabilities. For each vulnerability, it shows

[1] https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview

the full code of the class and highlights the line with the potential vulnerability. It also states the type of vulnerability and how severe the vulnerability is according to Fortify SCA. The teams go through these results. They write comments on whether or not they agree with the results and why. The pentesters check these comments. If they agree that there is a false positive, they will label the finding as ”approved”. Thus the warning from Fortify is deemed false positive. If they find the finding of Fortify SCA to be a true vulnerability, they label it as ”not approved”. Discussions might happen between the developers and pentesters in case of disagreement.

3.2. Gathering Data

To build the datasets, data from Fortify SCA needed to be extracted. To this aim, we used web scraping. We built a script in Python[2], which uses the Selenium WebDriver[3] to scrape the data from the Fortify SCA server. The data consisted of:

1. **Project Name:** Name of the project the class belongs to.
2. **Class location:** The path of the class within the project.
3. **Vulnerability Type:** The type of the vulnerability.
4. **Class name:** The name of the class.
5. **Vulnerability Location:** The line number of the vulnerability.

We built two datasets with these properties, one with the ”approved” vulnerabilities and one with the ”not approved” vulnerabilities. Next to these properties, we also extracted the source code of each.

The extraction of the dataset was done together with Ka-Wing Man, who used the same dataset for his thesis that focused on predicting true vulnerabilities using unsupervised learning [41]. We extracted the ”not approved” dataset first, so we could already look into projects that had true vulnerabilities and approach the developers to get access to their full project. The ”not approved” dataset was larger, so we split up the work of extracting that dataset. Finally, all the data was combined into one large dataset with true and false vulnerabilities.

3.3. Building the datasets

Now that the source code and vulnerability information are extracted from Fortify SCA, code complexity metrics can be extracted from the source code of each class. These metrics will be used for the research. Using a tool called CK [5], which includes the famous Chidamber and Kemerer metrics for object-oriented code [16], two datasets were built. One on class level and one on method level. Next to the code metrics, the vulnerability metrics from Fortify SCA were included in the datasets. That is the vulnerability type and the vulnerability location.

Because some classes may have multiple vulnerabilities, duplicates need to be removed. Some classes have both true and false vulnerabilities. In these cases, the class will be labeled as being truly vulnerable in the dataset. For the method-level dataset, only the method with the true vulnerability will be labeled as true.

Last, we came up with an additional metric. This metric is related to the vulnerabilities:

1. **Nested:** The level of nesting of the line of code of the potential vulnerability in the code.

This metric describes what level of nesting the potential vulnerability is in the class or in the method. A higher level of nesting might indicate more complexity and thus a higher chance of a potential vulnerability being a true vulnerability. Listing 3.1 shows an example code with a potential vulnerability on line 6. Counting the left curly brackets up to line 6 gives a nesting level of 3. If the potential vulnerability would have been on line 5, then the nesting level would have been 2.

[2] https://www.python.org/
[3] https://selenium.dev/

```
1 class Example{
2     private int parameter;
3
4     public void method(){
5         if(parameter == 0){
6             System.out.println("Potential vulnerability here");
7         }
8     }
9 }
```

Listing 3.1: Nested metric example

3.4. Analysing the datasets

Now that it is clear how the datasets were built, initial analysis can be performed on the datasets. This section will visualise the datasets and discuss the different metrics of the datasets.

3.4.1. Datasets visualised

Table 3.1 shows the number of vulnerable and not vulnerable instances in the class dataset and method dataset, respectively. It is clear from table 3.1 that in the dataset 12% of the classes are actual vulnerabilities, whereas 98% of the methods do not contain any vulnerability. The precision of Fortify for the class dataset is thus 0.122 and 0.018 for the method dataset. Recall can not be computed, for the negatives are missing in the Fortify dataset.

Dataset	False Vulnerabilities	True Vulnerabilities	Total	Precision
Class	10593 (87.8%)	1474 (12.2%)	12067	0.122
Method	97672 (98.2%)	1741 (1.8%)	99413	0.018

Table 3.1: Amount of vulnerabilities

Table 3.2 shows the top 10 vulnerability types found in the class dataset.

Vulnerability Type	False Vulnerabilities	True Vulnerabilities	Total
System Information Leak: Internal	4563	64	4627
J2EE Bad Practices: Threads	1511	104	1615
Log Forging	582	45	627
Password Management: Hardcoded Password	478	2	480
Path Manipulation	299	36	335
Rare Condition: Singleton Member Field	305	5	310
Unreleased Resource: Streams	217	48	582
Dynamic Code Evaluation: Unsafe Deserialization	110	82	192
Missing XML Validation	130	55	185
System Information Leak	79	106	185

Table 3.2: Top 10 vulnerability types in class dataset

The following table 3.3 shows the top 10 vulnerability types found in the method dataset.

Vulnerability Type	False Vulnerabilities	True Vulnerabilities	Total
System Information Leak: Internal	34821	73	34894
J2EE Bad Practices: Threads	15202	115	15317
Password Management: Hardcoded Password	6836	1	6837
Log Forging	6668	68	6736
Path Manipulation	3595	46	3641
Rare Condition: Singleton Member Field	2065	3	2068
Insecure Randomness	1503	5	1511
Header Manipulation	1481	3	1484
SQL Injection	1188	251	1439
System Information Leak	1238	144	1382

Table 3.3: Top 10 vulnerability types in method dataset

The top 5 vulnerability types from tables 3.2 and 3.3 will be used to create individual models for RQ3.

System Information Leak: Internal reveals system data or debug information by sending that to console, screen or local file. This can occur by logging or printing that data or information. An attacker might get some useful information of the system to exploit. For instance an SQL error message indicates that the system is vulnerable for SQL injection. In listing 3.2 the printed exception could leak information about the program, OS, and other applications.[4]

```
1 try {
2     //some code
3 } catch (Exception e) {
4     e.printStackTrace();
5 }
```

Listing 3.2: System Information Leak: Internal example

J2EE Bad Practices: Threads occurs when thread management is detected in the code. By the J2EE standard, thread management is forbidden in web applications in some circumstances. It is highly error prone, for it is difficult and leads to bugs that are hard to diagnose and detect. For instance race conditions, deadlock, and other synchronization errors[5]. In listing 3.3 the doGet() method creates and invokes a new thread. Because it is a Java servlet, whenever doGet() is called, a thread is already created[6].

[4] https://vulncat.fortify.com/en/detail?id=desc.dataflow.abap.system_information_leak_internal#Java%2fJSP

[5] https://vulncat.fortify.com/en/detail?id=desc.semantic.java.j2ee_badpractices_threads#Java%2fJSP

[6] https://cwe.mitre.org/data/definitions/383.html

```
1  public void doGet(HttpServletRequest request, HttpServletResponse response)
2  throws ServletException, IOException {
3
4      // Perform servlet tasks.
5      ...
6
7      // Create a new thread to handle background processing.
8      Runnable r = new Runnable() {
9          public void run() {
10
11             // Process and store request statistics.
12             ...
13         }
14     };
15
16     new Thread(r).start();
17 }
```

Listing 3.3: J2EE Bad Practices: Threads example

Password Management: Hardcoded Password occurs when passwords are visible in the code. People who have access to the code will be able to see the password. It is also hard to repair, for the code must be patched after production. Listing 3.4 shows a line of code that shows the username and password for connecting to a database. When an attacker has access to the bytecode of the program, that can get access to the disassembled code. As can be seen from listing 3.5, both the username and password are visible from the disassembled code.[7]

```
1 DriverManager.getConnection(url, "scott", "tiger");
```

Listing 3.4: Password Management: Hardcoded Password example

```
1 javap -c ConnMngr.class
2
3 22: ldc   #36; //String jdbc:mysql://ixne.com/rxsql
4 24: ldc   #38; //String scott
5 26: ldc   #17; //String tiger
```

Listing 3.5: Disassembled code with password

Log Forging occurs when an application gets data from an untrusted source and writes that data to a system or application log file. An attacker could then temper with the log file or corrupt it so it is no longer available. This way the attacker can cover its traces or put traces of someone else being the adversary. Depending on the log processing utility, an attacker might also be able to inject code or commands into the log file. Listing 3.6 logs an error when the code fails to parse an integer. Line 1 in listing 3.7 shows that error, when an attacker submits "twenty-one". If an attacker would submit "twenty-one%0a%0aINFO:+User+logged+out%3dbadguy", it will show line 3 of listing 3.7. The log file would then say that user "badguy" has logged out, while this is not the case.[8]

[7] https://vulncat.fortify.com/en/detail?id=desc.semantic.abap.password_management_hardcoded_password#Java%2fJSP

[8] https://vulncat.fortify.com/en/detail?id=desc.dataflow.abap.log_forging#Java%2fJSP

```
1 String val = request.getParameter("val");
2 try {
3         int value = Integer.parseInt(val);
4 }
5 catch (NumberFormatException nfe) {
6         log.info("Failed to parse val = " + val);
7 }
```

Listing 3.6: Log Forging code example

```
1 INFO: Failed to parse val=twenty-one
2
3 INFO: User logged out=badguy
```

Listing 3.7: Log Forging log example

Path Manipulation occurs when an attacker can specify a path used in an operation on the file system and by specifying the resource so that the attacker gains a non permitted capability. In listing 3.8 a file name is created from a HTTP request. By using a file name such as "../../tomcat/conf/server.xml", an attacker could overwrite or delete a configuration file.[9]

```
1 String rName = request.getParameter("reportName");
2 File rFile = new File("/usr/local/apfr/reports/" + rName);
3 ...
4 rFile.delete();
```

Listing 3.8: Path Manipulation example

3.4.2. Metrics

Next to the metrics and labels discussed in the previous two sections (vulnerability type, vulnerability location, and nested), we need to get metrics related to the source code for the dataset. As stated before, the CK [5] tool was used to gather these code metrics. This tool can extract both class level and method level related metrics and has important complexity-related metrics like Chidamber and Kemerer metrics. To better understand some of the more complex metrics, they will be explained after listing all the used metrics. The list of class-level metrics with their definitions are reported in Table 3.4.

[9] https://vulncat.fortify.com/en/detail?id=desc.dataflow.abap.path_manipulation#Java%2fJSP

Metric	Description
Type	The type of the instance.
CBO (Coupling Between Objects)	Counts the number of dependencies a class has, ignoring dependencies to Java itself.
WMC (Weight Method Class)	Counts the number of branch instructions in a class.
DIT (Depth Inheritance Tree)	Counts the number of ”fathers” a class has.
RFC (Response For Class)	Counts the number of unique method invocations in the class.
LCOM (Lack of Cohesion of Methods)	Number of method pairs who are not similar.
totalMethods	Total amount of methods in the class.
staticMethods	Total amount of static methods in the class.
publicMethods	Total amount of public methods in the class.
privateMethods	Total amount of private methods in the class.
protectedMethods	Total amount of protected methods in the class.
defaultMethods	Total amount of default methods in the class.
abstractMethods	Total amount of abstract methods in the class.
finalMethods	Total amount of final methods in the class.
synchronizedMethods	Total amount of synchronized methods in the class.
totalFields	Total amount of fields in the class.
staticFields	Total amount of static fields in the class.
publicFields	Total amount of public fields in the class.
privateFields	Total amount of private fields in the class.
protectedFields	Total amount of protected fields in the class.
defaultFields	Total amount of default fields in the class.
finalFields	Total amount of final fields in the class.
synchronizedFields	Total amount of synchronized fields in the class.
NOSI (Number of Static Invocations)	Counts the number of invocations to static methods.
LOC (Lines Of Code)	Counts the lines of actual code in the class.
returnQty	Counts the number of return instructions.
loopQty	Counts the number of loops.
comparisonsQty	Counts the number of comparisons.
tryCatchQty	Counts the number of try/catches.
parenthesizedExpsQty	Counts the number of expressions inside parenthesis.
stringLiteralsQty	Counts the number of string literals.
numbersQty	Counts the amount of numbers literals.
assignmentsQty	Counts the amount of assignment statements.
mathOperationsQty	Counts the number of math operations.
variablesQty	Counts the number of declared variables.
maxNestedBlocks	Highest number of blocks nested together.
anonymousClassesQty	Counts the number of anonymous classes.
subClassesQty	Counts the number of subclasses.
lambdasQty	Counts the number of lambda expressions.
uniqueWordsQty	Counts the number of unique words in the source code.
modifiers	Counts the number of modifiers in the class.

Table 3.4: Metrics for class dataset

For the method-level dataset, Table 3.5 list all method-level metrics with their definitions.

Metric	Description
method	The name of the method.
constructor	Boolean if the method is a constructor.
line	Line number of the method.
CBO (Coupling Between Objects)	Counts the number of dependencies a method has, ignoring dependencies to Java itself.
WMC (Weight Method Class)	Counts the number of branch instructions in a method.
RFC (Response For Class)	Counts the number of unique method invocations in the method.
LOC (Lines Of Code)	Counts the lines of actual code in the method.
returns	Counts the number of return instructions.
variables	Counts the number of declared variables.
parameters	Counts the number of parameters.
startLine	The line number where the method starts, which is the same as the line metric.
loopQty	Counts the number of loops.
comparisonsQty	Counts the number of comparisons.
tryCatchQty	Counts the number of try/catches.
parenthesizedExpsQty	Counts the number of expressions inside parenthesis.
stringLiteralsQty	Counts the number of string literals.
numbersQty	Counts the amount of numbers literals.
assignmentsQty	Counts the amount of assignment statements.
mathOperationsQty	Counts the number of math operations.
maxNestedBlocks	Highest number of blocks nested together.
anonymousClassesQty	Counts the number of anonymous classes.
subClassesQty	Counts the number of subclasses.
lambdasQty	Counts the number of lambda expressions.
uniqueWordsQty	Counts the number of unique words in the source code.
modifiers	Counts the number of modifiers in the class.

Table 3.5: Metrics for method dataset

CBO is the dependency between classes. From figure 3.2 it is clear that even tough class D does not depend on any other class, the CBO is still 1, for class C depends on class D.

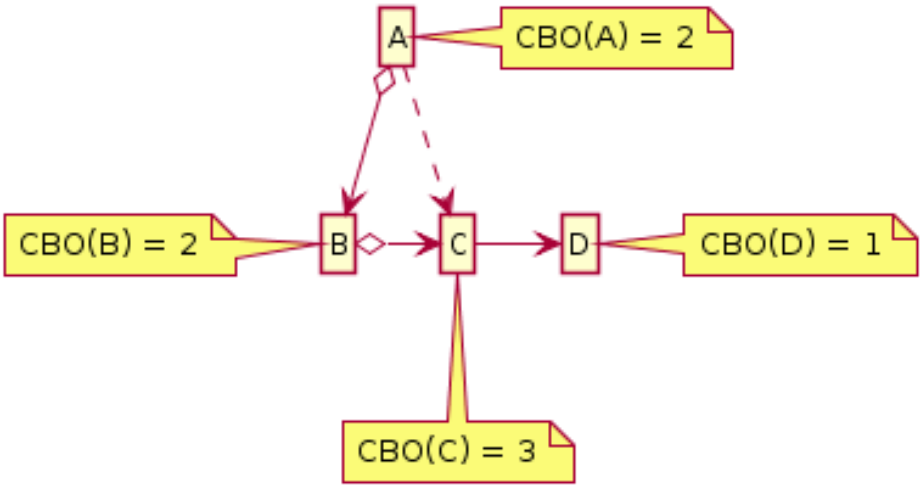

Figure 3.2: Example of CBO[10]

WMC is the sum of methods in a class or the sum of complexities of those methods [68]. Figure 3.3 shows three classes. Because the Clothing class has only one method, the WMC

[10] https://stackoverflow.com/questions/27515541/cbo-coupling-between-object

is 1. Appliances has three methods and thus the WMC is 3.

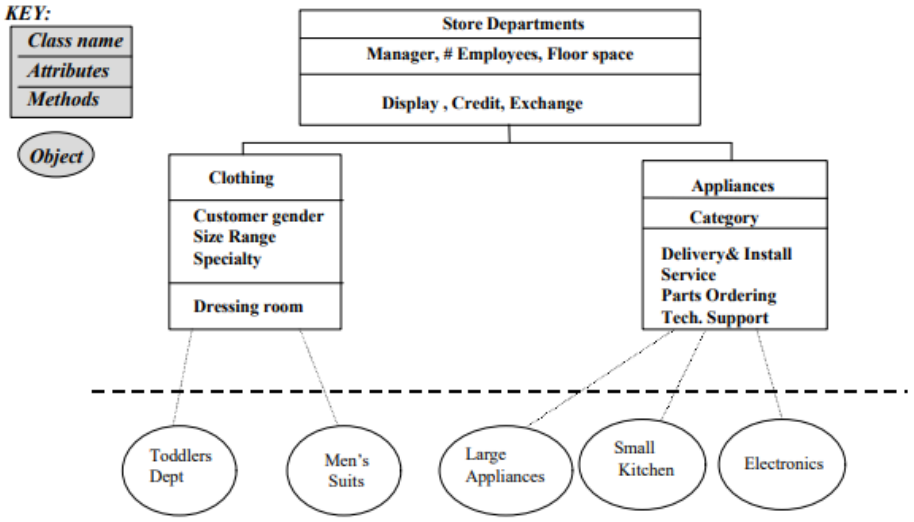

Figure 4: Object Oriented Application Example

Figure 3.3: Example of WMC [68]

DIT is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes within the inheritance hierarchy [68]. Looking at figure 3.3, Store Departments is the root and thus has a DIT of 0, whereas both Clothing and Appliances have DIT of 1. Those two classes inherent attributes and methods from Store Department.

RFC is the count of the set of all methods that can be invoked by some method in the class or in response to a message to an object of the class [68]. So from figure 3.3 the class Store Departments has 3 methods of its own, 1 method from the Clothing class and 4 methods from the Appliances class it can message. Thus the RFC of the Store Department class is 8.

LCOM is the dissimilarity of methods in a class by instance variable or attributes [68]. Figure 3.4 shows that two objects, Auto Parts and Cosmetics, depend on the class Store Department. Both objects however do not need all the methods of Store Department and thus have few methods in common. In this case LCOM is therefore high.

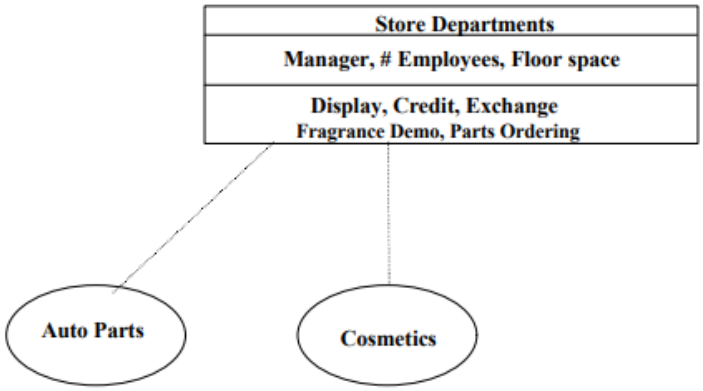

Figure 3.4: Example of LCOM [68]

Table A.1 summarizes all the metrics used for the class-level dataset, whereas table A.2 summarizes the metrics for the method-level dataset.

3.5. Preprocessing the datasets

Before the data can be used by the classifiers, the data needs to be preprocessed. The preprocessing of the data consists of three steps: **Normalization**, **Feature Selection** and **Class Imbalance**. This can also be seen in figure 3.1.

3.5.1. Normalization

First the data needs to be normalized. This because various metrics have different ranges, which can effect the way certain metrics are prioritized by certain classifiers, feature selection, and sampling algorithms. Normalization is not necessary for all the classification algorithms, for instance, Random Forest does not need normalization of the data, because it is tree based. Tree based algorithms require partitioning, whereas distance-based algorithms, which require Euclidean Distance, do need scaling/normalization. SVM, for instance, does need data normalization.[11]

There are two ways of normalization:

- **Standardization:** Rescaling the metrics to have a mean around 0 and a standard deviation of 1. The formula is given by equation 3.1, with v as the numeric value of the feature, v' as the standardised value, μ as the mean of the feature values, and σ as the standard deviation of the feature values.

$$v' = \frac{v - \mu}{\sigma} \tag{3.1}$$

- **Normalization:** Changing the metrics values to be in a specific range, for instance, scaling the metrics to be in a range between 0 and 1. This way the difference in range is kept, but the values between metrics will not differ too much. For example, metrics age and income. The age metric will have lower values than income. Classifiers might then become bias towards income in this case, when no normalization is applied.

Depending on the classifier, it can be more beneficial to use standardization instead of normalization.[12] Because the distribution of the metrics is unknown and no classifier is used which assumes a Gaussian distribution, normalization is used.

We use Weka's normalization function[13] (Min-Max normalization), to scale all the numeric values to be in the range between 0 and 1. The Min-Max normalization also scales each feature independently of all other features. This is done by checking the minimum and maximum value for each attribute. Then the function takes the difference between the numeric value and the minimum value and divides that with the difference between the maximum value and the minimum value. Thus, we have the following equations, with v as the numeric value of the feature, v' as the normalised value, and min and max as the minimum and maximum of the feature respectively:

$$v' = \frac{v - min}{max - min} \tag{3.2}$$

3.5.2. Feature Selection

With the two datasets now normalized, we need to remove some metrics that are not useful for this research. The project name, class location, class name, and method name are irrelevant for this research and they may also introduce biases towards classes and methods with certain names. Therefore, we removed them from the dataset.

Next, we need to select the right features. Using Pearson correlation[14], two heatmaps are created. Figure 3.5 shows a heatmap of the features in the class-level dataset, whereas figure 3.6 shows a heatmap for the features in the method-level dataset. Green blocks indicate a strong correlation and red blocks show no correlation. For instance, loc is correlated with many of the quantity related features, 9 out of 14. This because more lines of

[11] https://towardsdatascience.com/understand-data-normalization-in-machine-learning-8ff3062101f0
[12] https://medium.com/@swethalakshmanan14/how-when-and-why-should-you-normalize-standardize-rescale-your-da
[13] https://weka.sourceforge.io/doc.dev/weka/filters/unsupervised/attribute/Normalize.html
[14] https://www.statstutor.ac.uk/resources/uploaded/pearsons.pdf

code means more variables, more strings, etc. Or totalmethods being strongly correlated with publicmethods, which indicates that there are more public methods than other types of methods. Looking at the vulnerable label, we see that no other feature is correlated with it. Thus, it is not trivial to predict static analyzer warnings with this dataset.

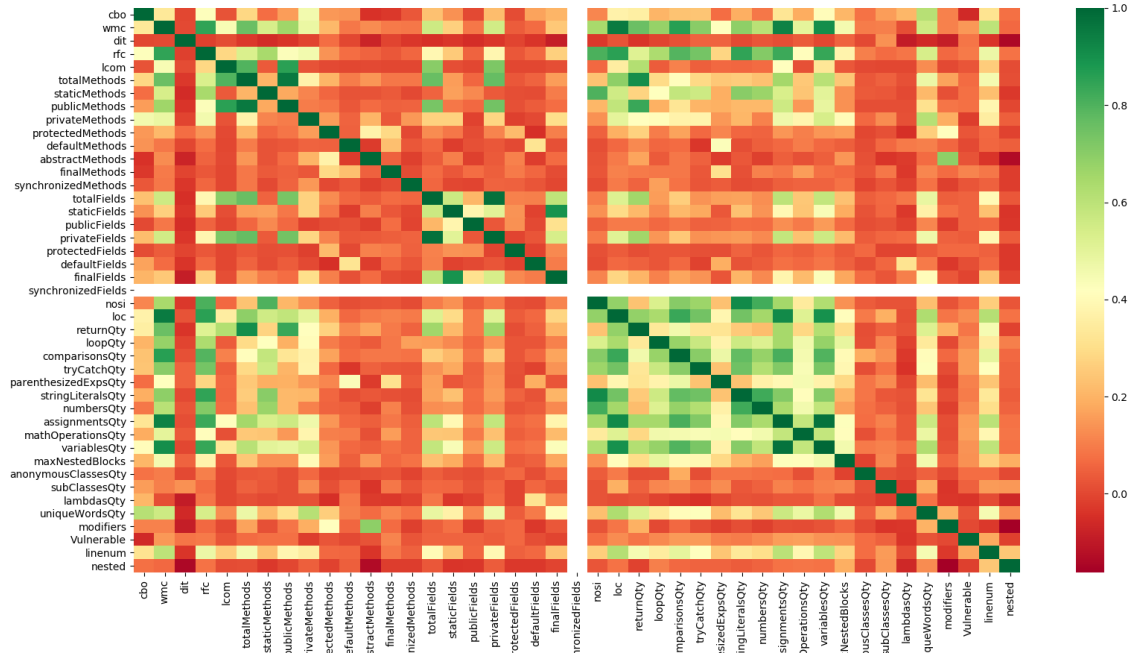

Figure 3.5: Heatmap of correlation between class-level features

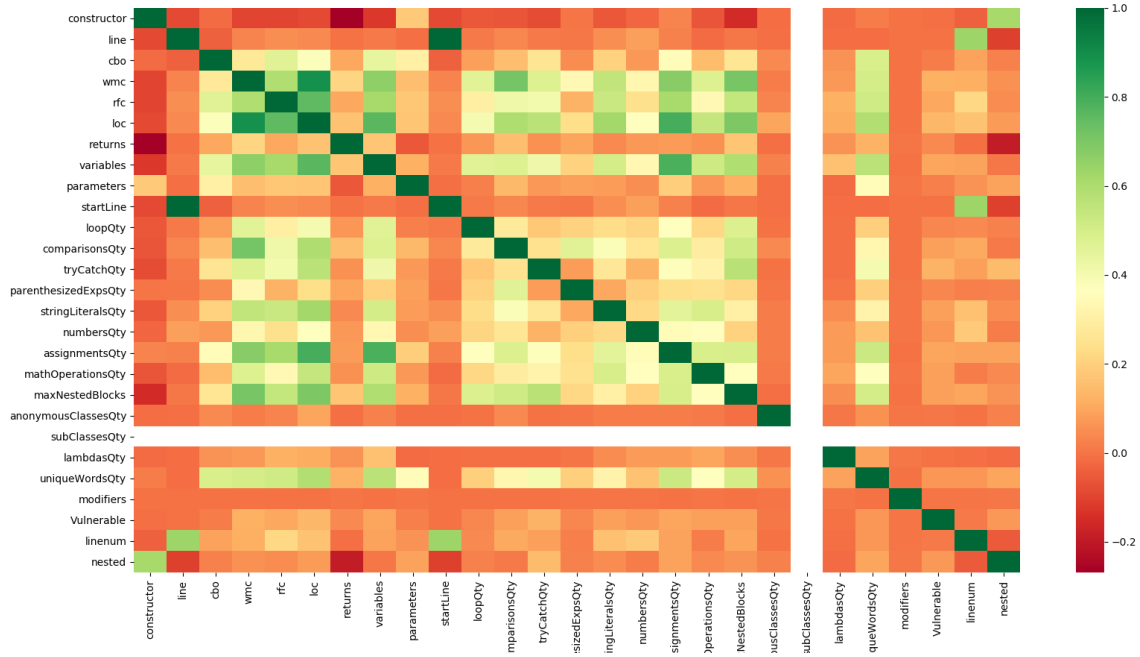

Figure 3.6: Heatmap of correlation between method-level features

It is clear from both figures that quite many of the features are correlated and feature selection can reduce the dimension of features. From the related work, it is clear that mRMR was the most powerful method for feature selection and had the most stable performance overall. Using this method[15], the following features are ranked highest for the class dataset (score above 0.04):

Ranking	Feature	Score
1	**loc**	0.128
2	**wmc**	0.117
3	**assignmentsQty**	0.109
4	**totalFields**	0.098
5	**rfc**	0.095
6	**totalMethods**	0.086
7	**variablesQty**	0.079
8	**uniqueWordsQty**	0.067
9	**privateMethods**	0.056
10	**stringLiteralsQty**	0.050
11	**returnQty**	0.046
12	**privateFields**	0.044

Table 3.6: Ranked features for class-level dataset

A threshold of 0.04 was chosen to get a decent number of features. With a threshold of 0.05, we would have only had 5 features for the method-level dataset. A threshold of lower than 0.04 would give to many features for the class-level dataset. For the method dataset, mRMR ranked the following features highest (score above 0.04):

Ranking	Feature	Score
1	**loc**	0.110
2	**maxNestedBlocks**	0.087
3	**variables**	0.077
4	**rfc**	0.058
5	**wmc**	0.050
6	**startLine**	0.045
7	**assignmentsQty**	0.040

Table 3.7: Ranked features for method-level dataset

3.5.3. Class Imbalance

As is clear from table 3.1 in section 3.4.1, the datasets are imbalanced. One of the classes (not vulnerable) is over represented in the datasets. To overcome this problem, sampling must be applied to the datasets.

Given previous research, we saw that oversampling techniques perform better than undersampling techniques and SMOTE performs better than ROS. Thus, we will use SMOTE in combination with the various classifiers to deal with the imbalance class problem of the datasets. Next to SMOTE, we will also use Weka's ClassBalancer[16]. The ClassBalancer algorithm reweights the instances, so that both classes have the same total weight. Thus, we will compare SMOTE and ClassBalancer to see which sampling technique gives a better performance with the different classifiers.

[15] http://home.penglab.com/proj/mRMR/
[16] https://weka.sourceforge.io/doc.dev/weka/filters/supervised/instance/ClassBalancer.html

4

Research Design

Now that the datasets are created and ready, the actual research can be executed. The research consists of multiple experiments, according to the three research questions:

RQ1: *How accurate are supervised machine learning methods in detecting true vulnerabilities from static analyzer warnings using a dataset from ING?*

RQ2: *How do classification methods perform at different levels of granularity on a dataset from ING?*

RQ3: *How accurate is a model built on a dataset with various types of vulnerabilities compared to models built on datasets per vulnerability type?*

This chapter will explain the setup of these experiments. For each research question, the methodology is explained to understand the approach for answering these questions.

4.1. RQ1

RQ1 is about comparing different supervised classifiers.

To answer RQ1, the following supervised classifiers will be used. Each of these classifiers represents one of the various categories of classifiers and are applicable for binary classification, as is our case. The default settings in Weka are used for these algorithms. The results were hardly effected by changing these settings. A summary of the algorithms can be found in table 4.1.

4.1.1. J48

J48 is a Java implementation of the C4.5 decision tree algorithm. It is therefore part of the decision tree or trees type machine learning algorithms[1] [24] [65]. Figure 4.1 shows a simple example of a decision tree.

[1] https://machinelearningmastery.com/a-tour-of-machine-learning-algorithms/

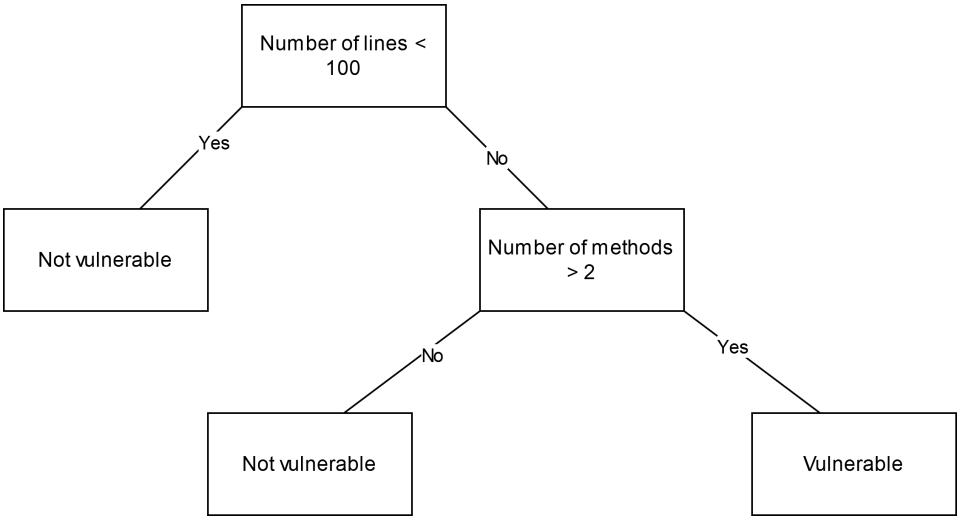

Figure 4.1: A simple example of a decision tree for classifying code

J48 creates decision trees by first checking for three base cases. If all the samples belong to the same class, it creates a leaf node which corresponds to that same class. If none of the features provide any information gain or the algorithm encounters a previously unseen class, then a decision node is created higher in the tree using the expected value of the class. After checking these base cases, the algorithm finds the normalised information gain for each attribute in the dataset. This information gain is the difference in entropy. A larger information gain means smaller entropy. The best attribute is then chosen by taking the attribute with the highest normalised information gain. A decision node is created, which splits at the best attribute. The algorithm then recurses on the sublists obtained by splitting and those nodes are added as children of the created decision node. Once the tree is created, J48 goes through the tree to remove branches that do not help and replace them by leaf nodes. This is the pruning the tree procedure and helps avoid overfitting.[2][3]

In Weka, the confidence factor used for pruning is set to 0.25 and the minimum number of instances per leaf is set to 2.

4.1.2. Random Forest

Random Forest is both part of the ensemble machine learning algorithms and decision trees algorithms[1] [25].

The way Random Forest works is by first creating multiple decision trees. To find the best split in the trees, each decision tree only considers a random number of features to select from instead of all the features. This way, diversity is created between the trees, so that they are less correlated[4]. The decisions of the trees are then aggregated and the majority vote is chosen as the prediction. Figure 4.2 shows a diagram of Random Forest.

[2] https://towardsdatascience.com/what-is-the-c4-5-algorithm-and-how-does-it-work-2b971a9e7db0
[3] https://en.wikipedia.org/wiki/C4.5_algorithm
[4] https://towardsdatascience.com/understanding-random-forest-58381e0602d2

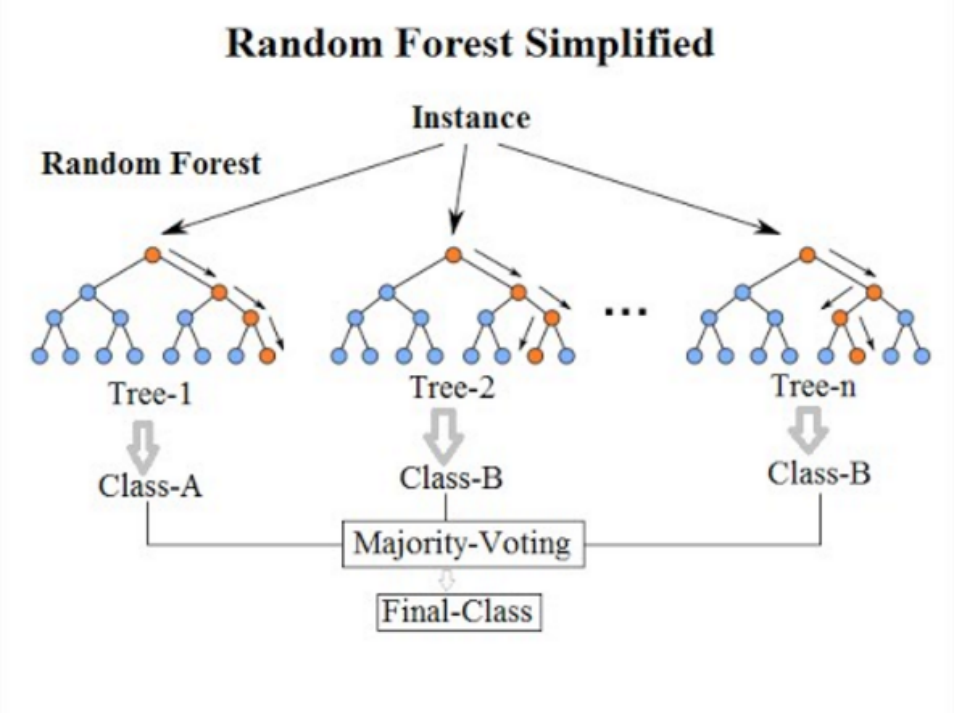

Figure 4.2: Random Forest[5]

The parameters used in Weka are set to 100 for the bagSizePercent and batchSize. The number of execution slots, the minimum number of instances, and the seed are set to 1. The minimum variance for splitting is set to 0.001 and the number of randomly chosen attributes is set to 0. These are the default parameters in Weka.

4.1.3. Naive Bayes

Naive Bayes is a Bayesian machine learning algorithm[1] [25]. It is based on Bayes theorem:

$$\Pr(A|B) = \frac{\Pr(B|A)\Pr(A)}{\Pr(B)} \tag{4.1}$$

In the equation above, A represents the class to be predicted and $B = (b_1, \dots, b_n)$ represents the set of features. Thus, given the chain rule, we get the new equation:

$$\Pr(A|b_1, \dots, b_n) = \frac{\Pr(b_1|A) \dots \Pr(b_n|A)\Pr(A)}{\Pr(b_1) \dots \Pr(b_n)} \tag{4.2}$$

Using the dataset, the above equation can be calculated to predict the class.[6]

The default parameters used in Weka are 100 for the batchSize and 2 for the numDecimalPlaces.

4.1.4. Support-vector machine (SVM)

SVM is a regression machine learning algortihm[1] [25]. It is a non-probabilistic binary linear classifier[7].

By creating a hyperplane, SVM divides the dataset in two different classes. The data point closes to the hyperplane on each side, is called the support vector. To get the best possible hyperplane, and thus division between the classes, the distance between the support vectors and the hyperplane, called margins, needs to be as large as possible. This way, there is a higher chance of new datapoints being classified correctly[8]. Figure 4.3 shows an example of

[5] https://en.wikipedia.org/wiki/Random_forest
[6] https://towardsdatascience.com/naive-bayes-classifier-81d512f50a7c
[7] https://en.wikipedia.org/wiki/Support-vector_machine
[8] https://towardsdatascience.com/support-vector-machine-simply-explained-fee28eba5496

the hyperplane in $\mathbb{R}^2$ and the support vectors. The hyperplane in this case is a line. When dealing with more than 2 features, like in this study, the hyperplane is an actual plane. This can be seen in figure 4.4.

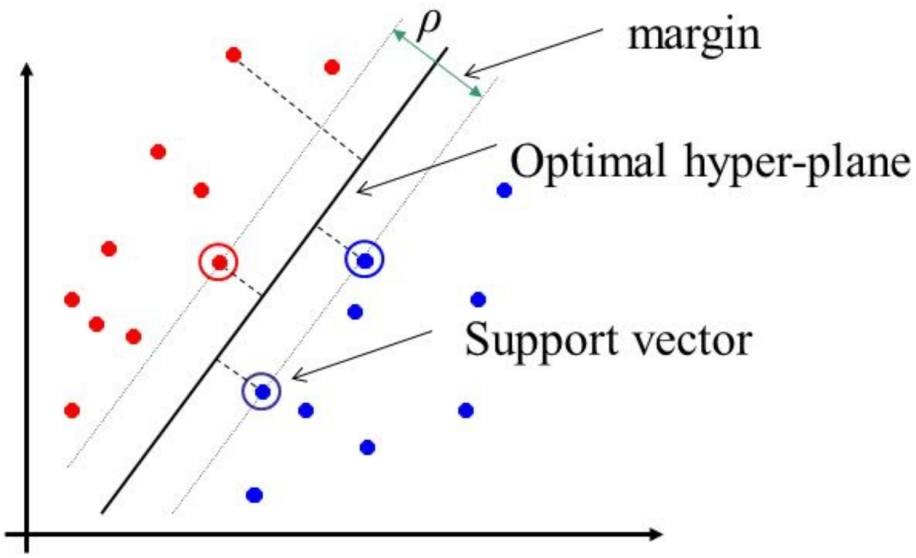

Figure 4.3: Example of SVM in $\mathbb{R}^2$[9]

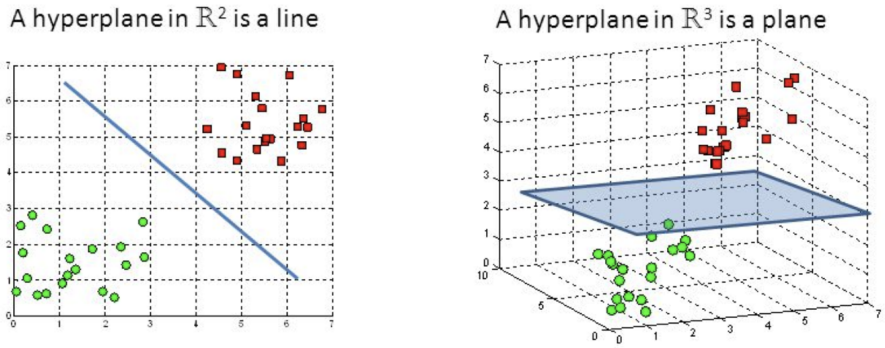

Figure 4.4: Hyperplanes in 2D and 3D feature space[10]

The default following parameters were used in Weka for SVM. batchSize = 100, cache-size = 40, coef0 = 0, cost = 1, degree = 3, eps = 0.001, gamma = 0, loss = 0.1, nu = 0.5, numDecimalPlaces = 2 and seed = 1.

4.1.5. Multilayer perceptron (MLP)

MLP is an artificial neural network algorithm[1] [64].

There are three layers in MLP, input, hidden, and output, with the possibility of multiple hidden layers between the input and output layer, as can be seen in figure 4.5. The input layer passes the data from the dataset to the hidden layer. The hidden and output layers consist of neurons that uses an activation function. Each neuron has weights for the inputs and a weight to deal with the bias of the inputs. The activation function sums the weights of the neuron and checks it against threshold to decide what the output will be. In Weka,

[9] https://www.mitosistech.com/support-vector-machine/

[10] https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a44

ApproximateSigmoid is used for the activation function, which approximates the logistic sigmoid activation function[11]:

$$f(x) = \frac{1}{1 + e^{(-x)}} \tag{4.3}$$

Before training the MLP, the data must be prepared. MLP requires data to be numerical and scaled. During training, each row of the dataset is exposed to the network at a time as the input. After processing the input, an output is generated. This output is then compared to the expected class of the input and an error is calculated. This error is then passed back through the network, layer by layer, to update the weights. This process is called backpropagation and is the learning part of MLP, where depending on the influence a weight has on a error, it is updated accordingly.[12]

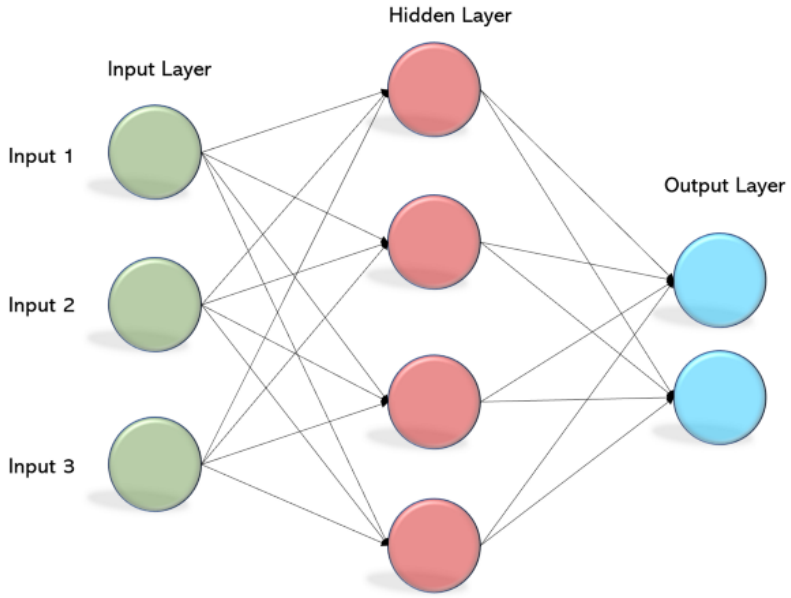

Figure 4.5: MLP layers[13]

Using Weka, the following parameters were used for MLP. batchSize = 100, learningRate = 0.3, momentum = 0.2, numDecimalPlaces = 2, seed = 0, trainingTime = 500, validationSetSize = 0 and validationTreshold = 20.

4.1.6. Bagging

Bagging is an ensemble machine learning algorithm, which uses sampling with replacement on the training set and will be applied to J48[14] [25].

Using the J48 algorithm with Bagging, the algorithm works similar to random forest. The main difference is Bagging selects from all the features to find the best split. Random Forest on the other hand, uses a random number of features to select from.

Figure 4.6 shows the three main steps of bagging. First the dataset is split in various smaller datasets. These datasets are built with replacements, so instances can appear more than once in the datasets. Then for each dataset a classifier is built and the results of each classifier is combined.[15]

[11] https://weka.sourceforge.io/doc.packages/multiLayerPerceptrons/weka/classifiers/functions/activation/ApproximateSigmoid.html
[12] https://machinelearningmastery.com/neural-networks-crash-course/
[13] https://becominghuman.ai/multi-layer-perceptron-mlp-models-on-real-world-banking-data-f6dd3d7e998f
[14] https://en.wikipedia.org/wiki/Bootstrap_aggregating
[15] https://medium.com/data-science-group-iitr/bagging-unraveled-8141ca078ccc

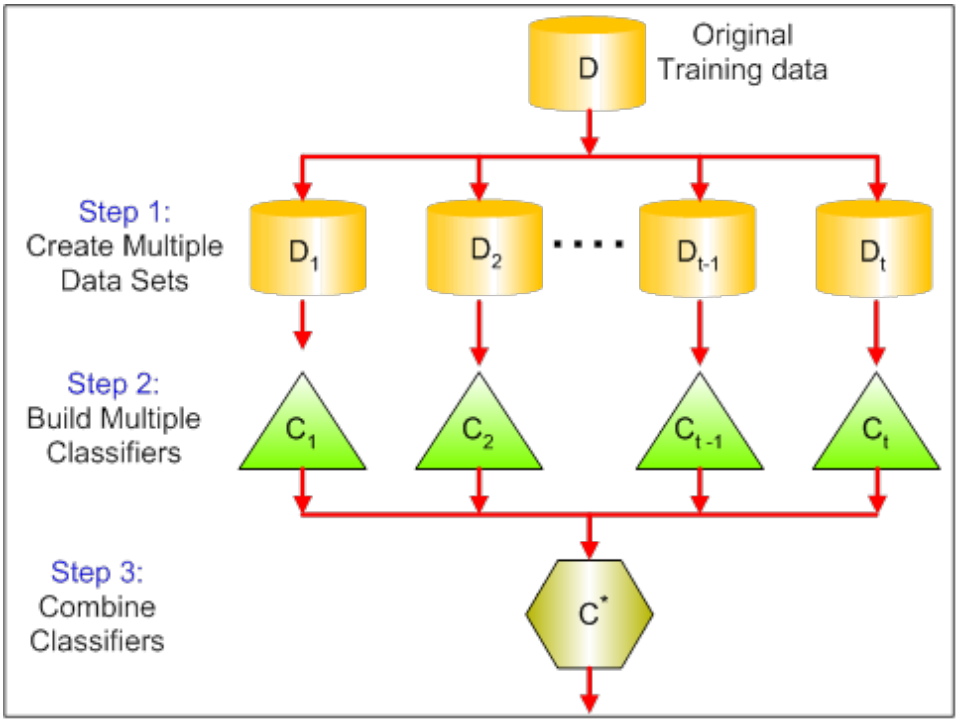

Figure 4.6: Bagging[16]

In Weka, the bagSizePercent and batchSize are set to 100. The number of execution slots and seed are set to 1, with the number of iterations being set to 10. As mentioned before, the J48 algorithm is used with Bagging in Weka.

[16] https://medium.com/data-science-group-iitr/bagging-unraveled-8141ca078ccc

Algorithm	Type	Parameters
J48/C4.5	Decision Tree	confidence factor=0.25 min number instances=2
Random Forest	Ensemble/Tree	bagSizePercent=100 batchSize=100 execution slots=1 min number instances =1 seed=1 min variance splitting=0.001 randomly chosen attributes=0
Naive Bayes	Bayesian	batchSize=100 numDecimalPlace=2
SVM	Regression	batchSize = 100 cachesize = 40 coef0 = 0 cost = 1 degree = 3 eps = 0.001 gamma = 0 loss = 0.1 nu = 0.5 numDecimalPlaces = 2 seed = 1
MLP	Artificial Neural Network	batchSize = 100 learningRate = 0.3 momentum = 0.2 numDecimalPlaces = 2 seed = 0 trainingTime = 500 validationSetSize = 0 validationTreshold = 20
Bagging(J48)	Ensemble	bagSizePercent=100 batchSize=100 execution slots=1 seed=1 iterations=10

Table 4.1: The chosen machine learning algorithms

Each of these classifiers will be tested by using cross-validation with 10 folds. By using k-fold cross validation, the classifiers can be evaluated on each data point with low bias. Each data point will be used for both training and testing, depending on the iteration [58]. Within each iteration, SMOTE/ClassBalancer will be applied to the training set. SMOTE will balance the data with a ratio of 54% FALSE and 46% TRUE. Figure 4.7 shows this process. Weka (version 3.8.3) [24] will be used to build the models. Because of the non-deterministic character of k-fold cross validation, we tried it multiple times with some classifiers. However, this did not give a large enough difference in performance, so we did not apply it to this research.

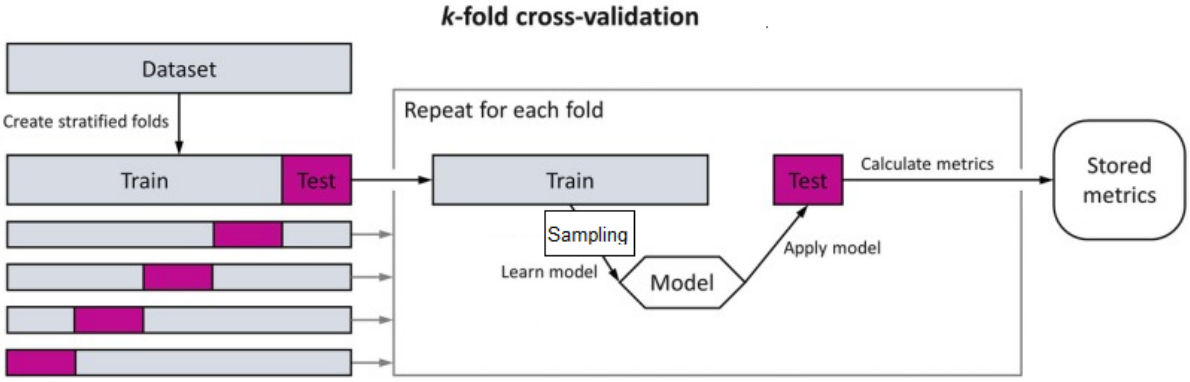

Figure 4.7: Classification schema with k-fold cross-validation and SMOTE[35]

Regarding evaluating the results of the classifiers, some metrics will not hold up because of the imbalance class problem. Accuracy, for instance, is not a good measurement when the data is imbalanced [78] [28]. A high accuracy could occur, because the model decided in favour of the class with the higher occurrence. This could mean that the model just deems everything FALSE, which will give it a high accuracy, but will not help predict real vulnerabilities.

Given the related work above and the benchmarking metrics described in Verma [81] and Haixiang et al. [29], the following benchmarking metrics will be used for performance comparison, with TP = True Positive, FP = False Positive and FN = False Negative:

- **Precision =** $\frac{TP}{TP+FP}$
- **Recall =** $\frac{TP}{P} = \frac{TP}{TP+FN}$
- **F-Measure =** $2 * \frac{prec*recall}{prec+recall} = \frac{2*TP}{2*TP+FN+FP}$
- **ROC AUC =** Integral of ROC, gives the trade-off between True Positive Rate and False Postive Rate.

Precision gives information about the amount of false positives and is an appropriate evaluation metric when minimizing the false positives is the top priority. Having the issue of a high false positive rate right now with Fortify, it is important to try get a low false positive rate.

The false negative rate is also crucial, because these are the actual vulnerabilities that are deemed not vulnerable. It is thus essential to keep this rate at close to zero, for else true vulnerabilities will be missed and these could be exploited in the future. In this case, recall is the appropriate evaluation metric.

Because both precision and recall are important in this case, we will use f-measure to compare the performance of the various classifiers, for f-measure combines both precision and recall by calculating the harmonic mean of the precision and recall.

The ROC AUC gives the trade-off between true positive rate and false positive rate. It tells the probability that a randomly chosen positive instance is ranked higher than a randomly chosen negative instance by using different thresholds. This metric can thus tell how good the model is at ranking predictions[17].

4.2. RQ2

Two different datasets were made, one based on class level and one on method level. To see how the two different models compare in performance, both datasets are used with the various classifiers stated above. With SMOTE, the distribution will be 58% FALSE and 42% TRUE. The benchmark metrics from above will also be used to compare the performance between the two datasets.

[17] https://neptune.ai/blog/f1-score-accuracy-roc-auc-pr-auc#3

4.3. RQ3

The top 5 most common software vulnerability types for each of the two datasets will be used to make 10 different datasets. 5 on class level and 5 on method level. The classifiers discussed in section 4.1 are then used on these 10 datasets, with 10-fold cross-validation. The results are then compared to the two datasets, class level and method level, containing all software vulnerabilities, to see whether it is better to create datasets by vulnerability type or datasets with various vulnerability types.

For the class-level dataset, the top 5 vulnerabilities are:

1. **System Information Leak: Internal**
2. **J2EE Bad Practices: Threads**
3. **Log Forging**
4. **Password Management: Hardcoded Password**
5. **Path Manipulation**

The datasets are first normalized using the method discussed in section 3.5.1. For each of the datasets, feature selection must then be performed using mRMR. Because the datasets have different values and number of instances, the scores may vary in value. The criterion for selecting the right number of features in this case is therefore selecting the top 10 highest ranked features. This gives the following top 10's features for each of the datasets.

System Information Leak: Internal	J2EE Bad Practices: Threads	Log Forging	Password Management: Hardcoded Password	Path Manipulation
assignmentsQty	loc	loc	abstractMethods	wmc
totalMethods	uniqueWordsQty	totalMethods	modifiers	assignmentsQty
rfc	totalMethods	assignmentsQty	lcom	loc
wmc	wmc	wmc	wmc	totalMethods
totalFields	totalFields	uniqueWordsQty	staticMethods	variablesQty
loc	privateFields	rfc	totalMethods	finalFields
variablesQty	returnQty	totalFields	privateMethods	mathOperationsQty
privateFields	privateMethods	returnQty	stringLiteralsQty	uniqueWordsQty
returnQty	assignmentsQty	variablesQty	returnQty	rfc
privateMethods	defaultMethods	comparisonsQty	rfc	staticFields

Table 4.2: Top 10 features for the top 5 vulnerabilities in the class-level dataset

For the method-level dataset, the top 5 vulnerabilities are:

1. **System Information Leak: Internal**
2. **J2EE Bad Practices: Threads**
3. **Password Management: Hardcoded Password**
4. **Log Forging**
5. **Path Manipulation**

Using feature selection, the following top 10s features are ranked highest by mRMR.

System Information Leak: Internal	J2EE Bad Practices: Threads	Password Management: Hardcoded Password	Log Forging	Path Manipulation
loc	wmc	variables	wmc	assignmentsQty
maxNestedBlocks	loc	maxNestedBlocks	loc	maxNestedBlocks
variables	maxNestedBlocks	loc	maxNestedBlocks	loc
rfc	startLine	assignmentsQty	assignmentsQty	wmc
startLine	rfc	startLine	rfc	startLine
wmc	variables	cbo	startLine	rfc
assignmentsQty	mathOperationsQty	wmc	variables	variables
stringLiteralsQty	uniqueWordsQty	rfc	uniqueWordsQty	mathOperationsQty
cbo	assignmentsQty	comparisonsQty	modifiers	tryCatchQty
nested	returns	stringLiteralsQty	mathOperationsQty	comparisonsQty

Table 4.3: Top 10 features for the top 5 vulnerabilities in the method-level dataset

Table 4.4 shows the SMOTE ratio for these datasets.

Dataset(Vulnerability)	Class FALSE(%)	Class TRUE(%)	Method FALSE(%)	Method TRUE(%)
System Information Leak: Internal	54	46	58	42
J2EE Bad Practices: Threads	57	43	57	43
Log Forging	54	46	-	-
Password Management: Hardcoded Password	58	42	58	42
Path Manipulation	54	46	56	44

Table 4.4: SMOTE ratios for the datasets

With the datasets ready and the top features known, we can build the classifiers.

4.3.1. Statistical Comparison

To compare the performance of the different classifiers and different datasets, statistical tests are computed on the f-measure. The Friedman test is used with the Nemenyi test as post-hoc analysis [20]. This hypothesis test is preferred over other non-parametric (data is not normally distributed) tests, when the same parameter has been measured under different conditions on the same subject[18]. In our case, the dependent parameter would be the f-measure and the dependent conditions are the different classifiers. The subjects are the independent classifiers.

The Friedman test is a non-parametric statistical test. It determines if there is a statistically significant difference between the performance of the classifiers/datasets. If p-value≤ 0.05, then the difference is statistically significant. In this case, a post-hoc analysis can be performed to find the groups of data that differ. The Nemenyi test makes pair-wise tests of the performance. If p-value> 0.05, the difference is not statistically significant. This means that the performance of the comparisons on the groups of data is similar and the groups are statistical incompatible[19].

The Friedman test consists of the following 5 steps when comparing the datasets [66]:

1. Name the number of datasets (≥ 3) k and blocks (e.g. classifiers) n
2. Rank the data within each block (e.g. rank the datasets for each classifier)
3. Add the ranks for each dataset separately; name the sums $T1, T2, ..., Tk$
4. Calculate the Friedman F_r statistic, which is distributed as chi-square, by

$$F_r = \frac{12}{nk(k+1)}(T_1^2 + T_2^2 + \cdots + T_k^2) - 3n(k+1) \tag{4.4}$$

[18] https://sixsigmastudyguide.com/friedman-non-parametric-hypothesis-test/
[19] https://en.wikipedia.org/wiki/Nemenyi_test

5. F_r is is approximately chi-squared (χ^2) distributed and the p-value is given by $P(\chi^2_{k-1} \geq F_r)$[20] [62]

If the Friedman test indicates significance, the Nemenyi test can be employed. The critical difference can then be calculated by [62]:

$$\left|\frac{R_i}{n_i} - \frac{R_j}{n_j}\right| > \frac{q_\infty; k; \alpha}{\sqrt{2}} \sqrt{\frac{k(k+1)}{6n}} \tag{4.5}$$

k and n are again the datasets and blocks, respectively. R_j and n_j denote the sum of Friedman-ranks and the sample size of the j-th group. q has a studentized range q distribution[21]. The critical values for q are presented in the Studentized Range q Table[22], with significance level α, k and $df = \infty$. If the inequality is met, then a difference between two groups is significant on the level of α [19].[23]

[20] https://en.wikipedia.org/wiki/Friedman_test

[21] https://www.real-statistics.com/one-way-analysis-of-variance-anova/kruskal-wallis-test/nemenyi-test-after-kw/

[22] https://www.real-statistics.com/statistics-tables/studentized-range-q-table/

[23] https://www.rdocumentation.org/packages/PMCMR/versions/4.3/topics/posthoc.friedman.nemenyi.test

5

Research Results

Having executed the experiments, this chapter will focus on the results of the experiments. The results for each research question will be shown and displayed in the next three sections, with the research questions being the following:

RQ1: *How accurate are supervised machine learning methods in detecting true vulnerabilities from static analyzer warnings using a dataset from ING?*

RQ2: *How do classification methods perform at different levels of granularity on a dataset from ING?*

RQ3: *How accurate is a model built on a dataset with various types of vulnerabilities compared to models built on datasets per vulnerability type?*

The results will also be analysed and discussed in this chapter.

5.1. RQ1

5.1.1. Class-level Dataset

Table 5.1 shows the results of the different classifiers using the class-level dataset.

Classifier	Precision	Recall	F-Measure	ROC AUC	Time(s)
Random Forest(SMOTE)	0,526	0,685	0,595	0,923	9.24
J48(SMOTE)	0,507	0,731	0,599	0,838	3.39
Naive Bayes(SMOTE)	0,315	0,184	0,232	0,592	2.07
SVM(SMOTE)	0,534	0,043	0,079	0,519	54.49
MLP(SMOTE)	0,334	0,550	0,416	0,765	31.43
Bagging(SMOTE)	0,519	0,708	0,599	0,922	10.94
Random Forest(ClassBalancer)	0,536	0,714	0,612	0,922	3
J48(ClassBalancer)	0,483	0,782	0,598	0,835	0.59
Naive Bayes(ClassBalancer)	0,297	0,149	0,199	0,578	0.06
SVM(ClassBalancer)	0,130	0,512	0,208	0,519	17.76
MLP(ClassBalancer)	0,227	0,656	0,337	0,728	14.72
Bagging(ClassBalancer)	0,514	0,777	0,618	0,925	4.07

Table 5.1: Performance of classifiers on class dataset

From this table, it is clear that Bagging in combination with ClassBalancer, gives both the highest f-measure and ROC AUC. Thus, it can be considered the best overall performing classifier, given the class dataset. Random Forest with ClassBalancer gives the highest

precision, whereas J48 with ClassBalancer gives the highest recall. The fastest classifier is Naive Bayes with ClassBalancer.

ClassBalancer can be deemed the better sampling technique, compared to SMOTE in this case. Using ClassBalancer gives the highest precision, recall, f-measure, and ROC AUC.

With a higher recall than precision, Bagging with ClassBalancer has a lower false negative rate compared to the false positive rate. So true vulnerabilities are more likely to be found, but at the cost of an increase in false vulnerabilities.

Looking at the lowest values, we see that SVM with ClassBalancer has the lowest precision and SVM with SMOTE has the lowest recall and f-measure and the highest run time. SVM also has the lowest ROC AUC, thus SVM performs the worst with the class dataset.

Compared to Fortify as a stand alone, we see that the precision of each classifier (using Fortify data) is higher than the precision of Fortify (0.122). This means that the classifiers are better at classifying the positives, which are true vulnerabilities, than Fortify alone. The highest precision is 0.536 and is of the Random Forest(ClassBalancer) classifier. The lowest precision is also still higher than that of Fortify, which is 0.130 for the SVM(ClassBalancer) classifier.

Looking at the curve in figure 5.1, it is clear that to get 80% of the true positives, around 20% of the sample size needs to be checked. This implies that the pentesters and developers at ING only need to manually check 20% of the Fortify warnings to find 80% of the true vulnerabilities.

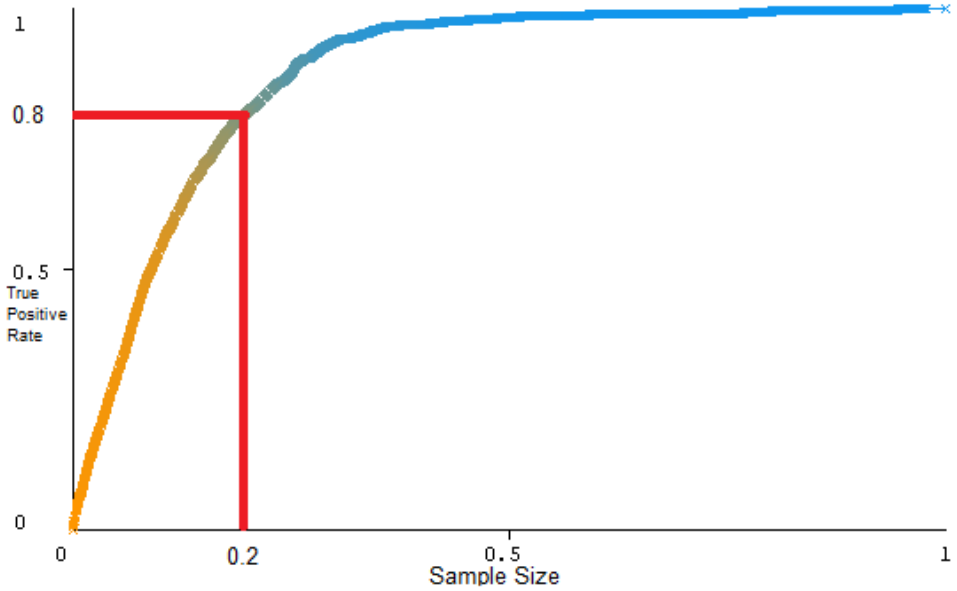

Figure 5.1: TPR vs Sample Size curve Bagging with ClassBalancer for class dataset

The precision, recall, and f-measure of the top 5 vulnerabilities for the class dataset are listed in table 5.2.

Vulnerability Type	Precision	Recall	F-Measure
System Information Leak: Internal	0,145	0,391	0,216
J2EE Bad Practices: Threads	0,529	0,885	0,662
Log Forging	0,378	0,689	0,488
Password Management: Hardcoded Password	0,071	0,5	0,125
Path Manipulation	0,337	0,806	0,475

Table 5.2: Precision, recall and f-measure of the top 5 vulnerabilities in the class dataset

From this table, it is clear that Bagging with ClassBalancer is better at finding J2EE Bad Practices: Threads vulnerabilities, but bad at finding actual Password Management:Hardcoded Password vulnerabilities. The model also performs not that well on the other top vulnerability types, with a f-measure of < 0.5. This could be because these types of vulnerabilities are not reflected by code metrics. A hardcoded password, for instance, is not really represented by any of the code metrics used in this dataset.

Figures A.1-A.4 are a graphical view of the different performance of the classifiers and how different evaluation metrics compare with each other. Tables A.3-A.14 are the confusion matrices for the classifiers. Figure A.9 shows the time for each classifier to be build by Weka.

5.1.2. Method-level Dataset

Table 5.3 shows the results of the different classifiers using the method dataset. Because SVM took too much time to execute (> 3 days), the results could not be retrieved and therefore are not present in the table.

Classifier	Precision	Recall	F-Measure	ROC AUC	Time(s)
Random Forest(SMOTE)	0,345	0,512	0,412	0,909	86.59
J48(SMOTE)	0,216	0,553	0,310	0,801	20.43
Naive Bayes(SMOTE)	0,093	0,055	0,304	0,649	2.5
SVM(SMOTE)	-	-	-	-	-
MLP(SMOTE)	0,064	0,455	0,113	0,741	123.13
Bagging(SMOTE)	0,257	0,538	0,348	0,892	26.23
Random Forest(ClassBalancer)	0,210	0,535	0,301	0,900	18.82
J48(ClassBalancer)	0,170	0,608	0,266	0,786	5.7
Naive Bayes(ClassBalancer)	0,056	0,289	0,094	0,661	0.16
SVM(ClassBalancer)	-	-	-	-	-
MLP(ClassBalancer)	0,018	0,898	0,035	0,521	55.84
Bagging(ClassBalancer)	0,194	0,585	0,291	0,875	53.11

Table 5.3: Performance of classifiers on method dataset

From this table, it is clear that Random Forest in combination with SMOTE, gives both the highest f-measure and ROC AUC. Thus, it can be considered the best overall performing classifier, given the method dataset. Random Forest with SMOTE gives the highest precision, whereas MLP with ClassBalancer gives the highest recall. The fastest classifier is Naive Bayes with ClassBalancer.

SMOTE can be deemed the better sampling technique compared to ClassBalancer in this case, although MLP with ClassBalancer has the highest recall.

With a higher recall than precision, Random Forest with SMOTE has a lower false negative rate compared to the false positive rate. Therefore, true vulnerabilities are more likely to be found, but at the cost of an increase in false vulnerabilities.

The overall best f-measure is $0.412 < 0.5$ and can thus be considered weak. Therefore, even though Random Forest with SMOTE performs the best, it is not a good model for clas-

sifying vulnerabilities given the method-level dataset. A reason could be the fact that the method-level dataset is much more skewed than the class-level dataset. For each class having multiple methods, with only one method being potentially vulnerable.

Looking at the lowest values, we see that MLP with ClassBalancer has the lowest precision, f-measure, and ROC AUC and Naive Bayes with SMOTE has the lowest recall. MLP with SMOTE has the highest run time.

The precision of the classifiers are again higher than the precision of Fortify as a stand alone(0.018), meaning that the classifiers combined with Fortify are better at classifying the true vulnerabilities, than Fortify alone. Only for the MLP(ClassBalacer) classifier is the precision equal to that of Fortify. The highest precision is 0.345 for the Random Forest(SMOTE) classifier.

The TPR vs Sample Size curve can be seen in figure 5.2. Looking at the ranking principle, it is clear that to get 80% of the true positives, 10% of the sample size needs to be checked. Thus only 10% of the Fortify warnings needs to be checked manually by pentesters, to get 80% of true vulnerabilities.

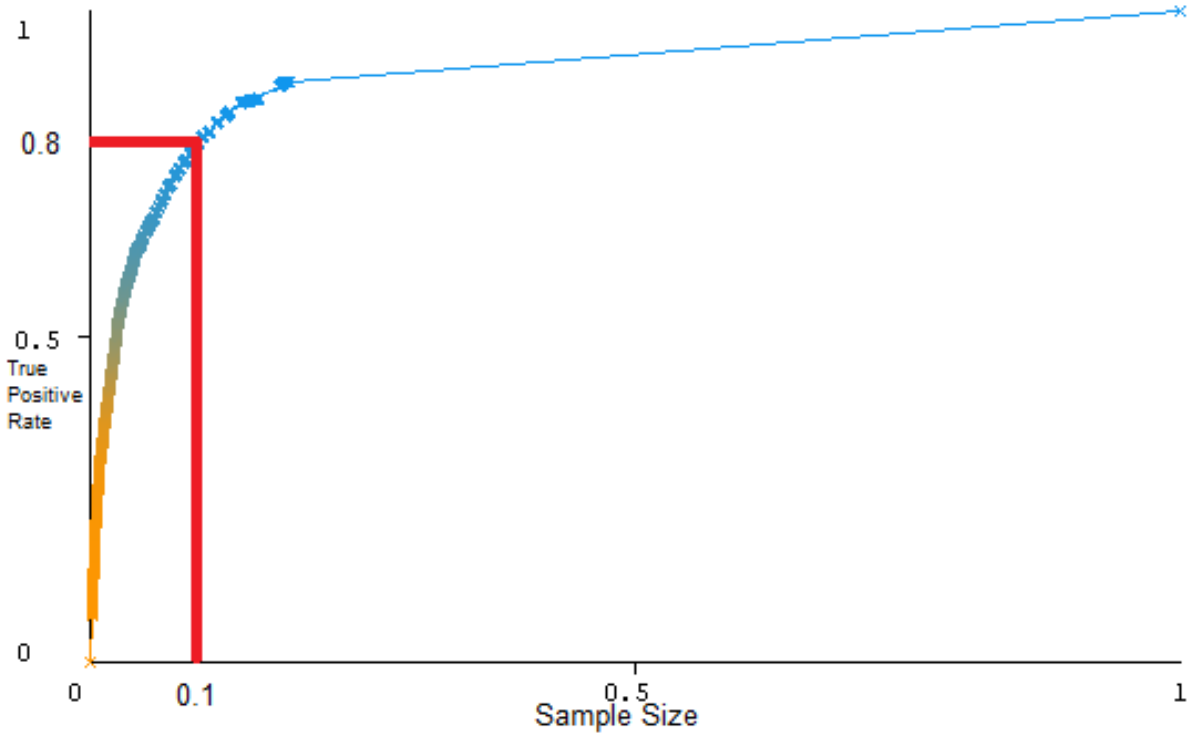

Figure 5.2: TPR vs Sample Size curve Random Forest with SMOTE for method dataset

The precision, recall, and f-measure of the top 5 vulnerabilities for the method dataset are listed in table 5.4.

Vulnerability Type	Precision	Recall	F-Measure
System Information Leak: Internal	0,084	0,247	0,125
J2EE Bad Practices: Threads	0,428	0,565	0,487
Password Management: Hardcoded Password	0	0	-
Log Forging	0,195	0,221	0,207
Path Manipulation	0,153	0,478	0,232

Table 5.4: Precision, recall and f-measure of the top 5 vulnerabilities in the method dataset

From this table, it is clear that Random Forest with SMOTE is the best at finding J2EE BadPractices: Threads vulnerabilities, but bad at finding actual Password Management:Hardcoded Password vulnerabilities. The model overall also performs not that well with the top 5 vulnerability types, with all the f-measures being < 0.5. Again, this probably has to do with the fact that some of these vulnerability types, like Password Management: Hardcoded Password are not represented well by code complexity metrics.

Figures A.5-A.8 are a graphical view of the different performance of the classifiers and how different evaluation metrics compare with each other.The confusion matrices for the classifiers are represented by tables A.15-A.24. Figure A.10 shows the time for each classifier to be built by Weka.

5.1.3. Research Question Answer

Given the results found in this section, we can now answer the following research question:

RQ1: *How accurate are supervised machine learning methods in detecting true vulnerabilities from static analyzer warnings using a dataset from ING?*

Looking at the f-measure, we see that the class-level dataset has the highest f-measure (0.618) for Bagging with ClassBalancer. This classifier also has the highest ROC AUC of 0.925. The classifier is therefore good at finding false vulnerabilities, but misses quite some of the true vulnerabilities from static analyzer warnings. Thus, the overall performance is well enough, but there is still room for improvement.

For the method-level dataset, the highest f-measure and ROC AUC was found by Random Forest with SMOTE, respectively, 0.412 and 0.909. Because of the very low f-measure (less than 0.5), the method-level dataset is not very accurate in finding true vulnerabilities from static analyzer warnings.

5.2. RQ2

Looking at the results from the previous section, we see that the best performing classifier for the class-level dataset is Bagging with ClassBalancer, having a f-measure of 0.618. For the method-level dataset, Random Forest with SMOTE has the highest f-measure of 0.412. The highest precision and ROC AUC was found using the class-level dataset, whereas the highest recall was found using the method-level dataset. Overall, the class-level dataset has a better performance than the method-level dataset.

The classifiers using class-level dataset are faster overall, compared to the method-level dataset. One reason for this difference is the fact that the method-level dataset is more than 8 times larger than the class-level dataset.

5.2.1. Research Question Answer

Given the research question and the results, we can now answer the following:

RQ2: *How do classification methods perform at different levels of granularity on a dataset from ING?*

The classifiers have an overall better performance on class level, compared to method level. The highest f-measure, precision and ROC AUC can be found using the class-level dataset, whereas the highest recall can be found using the method-level dataset.

5.3. RQ3

For each of the 5 vulnerability types, datasets were created. Both the class level and method level were filtered for each vulnerability type. The remaining instances were then used as the corresponding dataset. So for example, to create the Log Forging class-level dataset, all the instances with vulnerability type equal to Log Forging were taken from the class-level dataset. Each of these datasets were used on different classifiers to see which performs

better. Table 5.5 shows the f-measure of the top 5 vulnerabilities in the class-level dataset. Table 5.6 shows the time in seconds of the top 5 vulnerabilities in the class-level dataset. The Password Management: Hardcoded Password does not have SMOTE, because there are too few minority instances to use SMOTE.

Classifier	System Information Leak: Internal	J2EE Bad Practices: Threads	Log Forging	Password Management: Hardcoded Password	Path Manipulation
Random Forest(SMOTE)	0,181	0,659	0,447	-	0,641
J48(SMOTE)	0,189	0,652	0,365	-	0,605
Naive Bayes(SMOTE)	0,042	0,349	0,314	-	0,318
SVM(SMOTE)	0,058	0,374	0,321	-	0,329
MLP(SMOTE)	0,051	0,643	0,368	-	0,581
Bagging(SMOTE)	0,187	0,674	0,396	-	0,568
Random Forest(ClassBalancer)	0,024	0,681	0,400	0,000	0,623
J48(ClassBalancer)	0,048	0,662	0,362	0,000	0,530
Naive Bayes(ClassBalancer)	0,051	0,387	0,271	0,000	0,309
SVM(ClassBalancer)	0,051	0,332	0,304	0,089	0,314
MLP(ClassBalancer)	0,029	0,405	0,400	0,011	0,431
Bagging(ClassBalancer)	0,043	0,644	0,375	0,000	0,533

Table 5.5: F-Measure of classifiers on class-level dataset per vulnerability type

Classifier	System Information Leak: Internal	J2EE Bad Practices: Threads	Log Forging	Password Management: Hardcoded Password	Path Manipulation
Random Forest(SMOTE)	0.51	0.51	0.19	-	0.07
J48(SMOTE)	0.03	0.03	0.01	-	0.01
Naive Bayes(SMOTE)	0.02	0.02	0	-	0
SVM(SMOTE)	0.56	0.56	0.1	-	0.03
MLP(SMOTE)	2.69	2.69	1.07	-	0.55
Bagging(SMOTE)	0.23	0.23	0.12	-	0.03
Random Forest(ClassBalancer)	0.16	0.16	0.06	0.02	0.04
J48(ClassBalancer)	0.01	0.01	0.01	0	0
Naive Bayes(ClassBalancer)	0	0	0	0	0
SVM(ClassBalancer)	0.24	0.24	0.04	0.01	0.01
MLP(ClassBalancer)	1.61	1.61	0.62	0.48	0.33
Bagging(ClassBalancer)	0.12	0.12	0.05	0.01	0.02

Table 5.6: Time in s of classifiers on class-level dataset per vulnerability type

Figures A.11-A.15 show the time for each classifier to be build by Weka for each of the datasets.

Table 5.7 shows the f-measure for the top 5 vulnerability types in the method-level dataset. Table 5.8 shows the time in seconds of the the top 5 vulnerability types in the method-level dataset. The Password Management: Hardcoded Password does not have SMOTE and SVM with ClassBalancer, because there are too few minority instances to use those algorithms.

Classifier	System Information Leak: Internal	J2EE Bad Practices: Threads	Password Management: Hardcoded Password	Log Forging	Path Manipulation
Random Forest(SMOTE)	0,112	0,574	-	0,277	0,465
J48(SMOTE)	0,087	0,460	-	0,267	0,408
Naive Bayes(SMOTE)	0,014	0,070	-	0,063	0,163
SVM(SMOTE)	-	0,092	-	0,065	0,121
MLP(SMOTE)	0,024	0,198	-	0,120	0,188
Bagging(SMOTE)	0,081	0,498	-	0,275	0,458
Random Forest(ClassBalancer)	0,053	0,613	0,000	0,286	0,222
J48(ClassBalancer)	0,065	0,420	0,000	0,218	0,211
Naive Bayes(ClassBalancer)	0,013	0,078	0,000	0,068	0,157
SVM(ClassBalancer)	0,018	0,080	-	0,051	0,110
MLP(ClassBalancer)	0,004	0,015	0,000	0,020	0,030
Bagging(ClassBalancer)	0,061	0,515	0,000	0,262	0,202

Table 5.7: F-Measure of classifiers on method-level dataset per vulnerability type

Classifier	System Information Leak: Internal	J2EE Bad Practices: Threads	Password Management: Hardcoded Password	Log Forging	Path Manipulation
Random Forest(SMOTE)	18.45	6.62	-	2.46	0.98
J48(SMOTE)	2.35	0.82	-	0.18	0.08
Naive Bayes(SMOTE)	0.14	0.06	-	0.03	0.01
SVM(SMOTE)	-	43.68	-	8.93	1.8
MLP(SMOTE)	61.19	29.67	-	12.33	6.56
Bagging(SMOTE)	17.9	4.07	-	1.51	0.61
Random Forest(ClassBalancer)	4	1.35	0.16	0.58	0.3
J48(ClassBalancer)	0.5	0.21	0.01	0.05	0.04
Naive Bayes(ClassBalancer)	0.06	0.02	0.01	0.01	0.01
SVM(ClassBalancer)	120.82	17.03	-	3.41	0.71
MLP(ClassBalancer)	35.02	15.9	7	7.18	3.78
Bagging(ClassBalancer)	5.06	1.63	0.12	0.75	0.27

Table 5.8: Time in s of classifiers on method-level dataset per vulnerability type

Figures A.16-A.20 show the time for each classifier to be built by Weka for each of the datasets.

Looking at table 5.9, the following classifiers perform best for the 5 different vulnerability types on class level:

1. J48 with SMOTE
2. Random Forest with ClassBalancer
3. Random Forest with SMOTE
4. SVM with ClassBalancer
5. Random Forest with SMOTE

On method level, the following classifiers perform best for the 5 different vulnerability types:

1. Random Forest with SMOTE
2. Random Forest with ClassBalancer
3. Random Forest & J48 with ClassBalancer, comparing the ROC AUC
4. Random Forest with ClassBalancer
5. Random Forest with SMOTE

Because there are too few minority classes in the ”Password Management: Hardcoded Password” dataset, classifiers, classifies all the minority instances as negative.

Again Naive Bayes with ClassBalancer is the fastest in all cases or tied with other classifiers on both class and method level. For the class-level datasets, MLP with SMOTE takes the most time. On method level, SVM with ClassBalancer is the slowest for System Information Leak: Internal and MLP with ClassBalancer for Password Management: Hardcoded Password. SVM with SMOTE takes the most time for J2EE Bad Practices:Threads, Log Forging, and Path Manipulation..

Figure 5.3 shows the comparison of the precision and figure 5.4 shows the comparison of the recall between the class-level, method-level, and the vulnerability per type datasets. For the comparison, we used the precision and recall of the classifiers with the highest f-measure.

It is clear from figure 5.4, that the recall is overall better in the class-level dataset. Only for J2EE Bad Practices Threads and Password Management: Hardcoded Password does the class-level per vulnerability type dataset have a better recall.

For precision, the class-level per vulnerability type datasets perform best, with Password Management: Hardcoded Password being the exception. In that case, the class-level dataset has a slightly higher precision.

The highest precision for each of the datasets per vulnerability type is also higher than that of Fortify as stand alone, with Password Management: Hardcoded Password being the exception again.

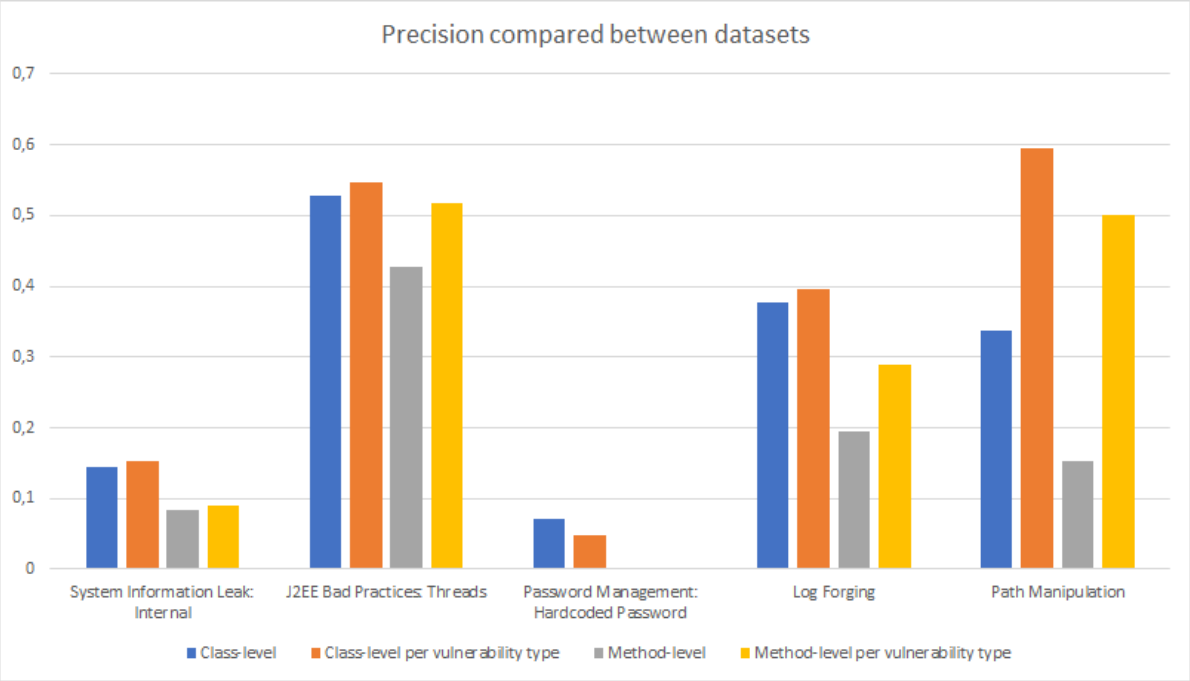

Figure 5.3: Comparison of precision between datasets

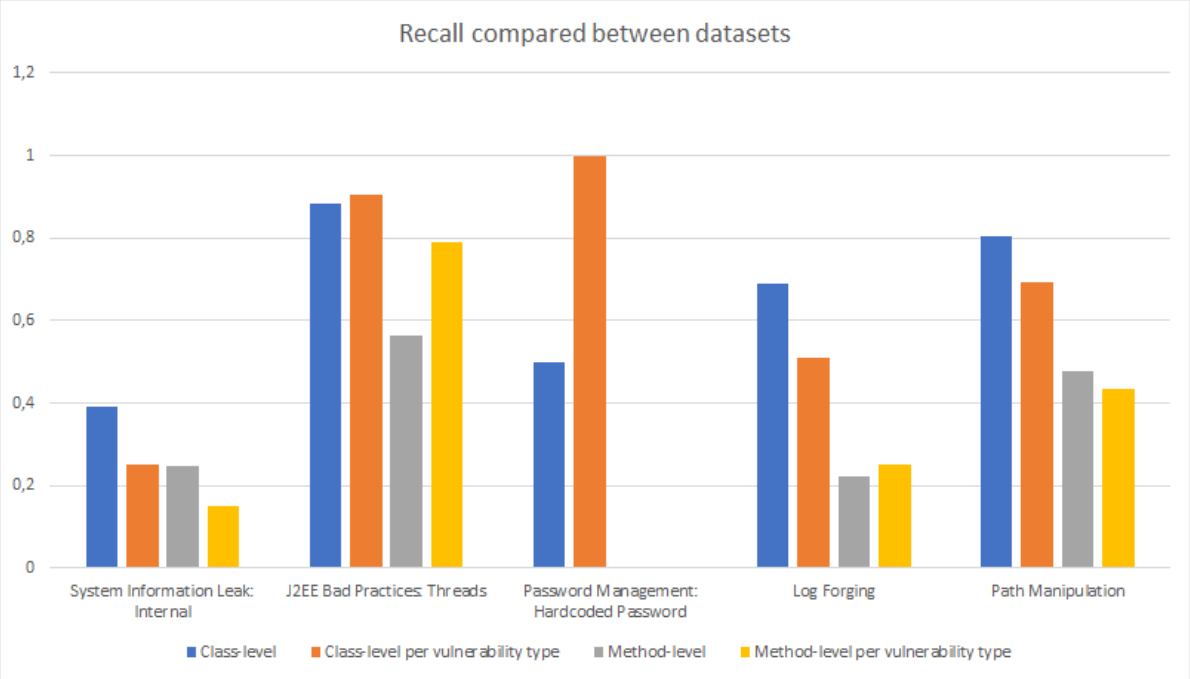

Figure 5.4: Comparison of recall between datasets

Figure 5.5 shows the comparison of the f-measure between the class-level, method-level, and the individual vulnerability type datasets. From this figure, it is clear that the class-level dataset performs best for the System Information Leak: Internal, Password Management: Hardcoded Password and Log Forging vulnerabilities. The class-level per individual vulnerability type datasets perform better in case of the J2EE Bad Practices: Threads and Path Manipulation vulnerabilities.

Figure 5.5: Comparison of f-measure between datasets

5.3.1. Overall Classifier comparison

In order to compare the performance between all the classifiers and all the datasets, the Friedman test method with the post-hoc test (Nemenyi) from Demšar [20] is used. For this test, the data from table 5.9 is used. In this dataset, each row represents one of the datasets and each column represents the classifiers.

Performing the Friedman test gives p-value $= 9.234e^{-10} < \alpha = 0.05$. Thus, the differences between the f-measure values are statistically significant. We use the post-hoc test to get a ranking. From the ranking in table 5.10, it is clear that Random Forest with SMOTE has the best performance and SVM with SMOTE has the worst performance. In figure 5.6 you can see that the critical distance is 4.810. This means that classifiers with that amount of distance between them have a high enough statistical difference. So, Random Forest with SMOTE has a statistical difference with everything above MLP with SMOTE, meaning that the difference is not by chance alone. The grey area shows the classifiers that have a low critical distance with the best performing classifier, Random Forest with SMOTE.

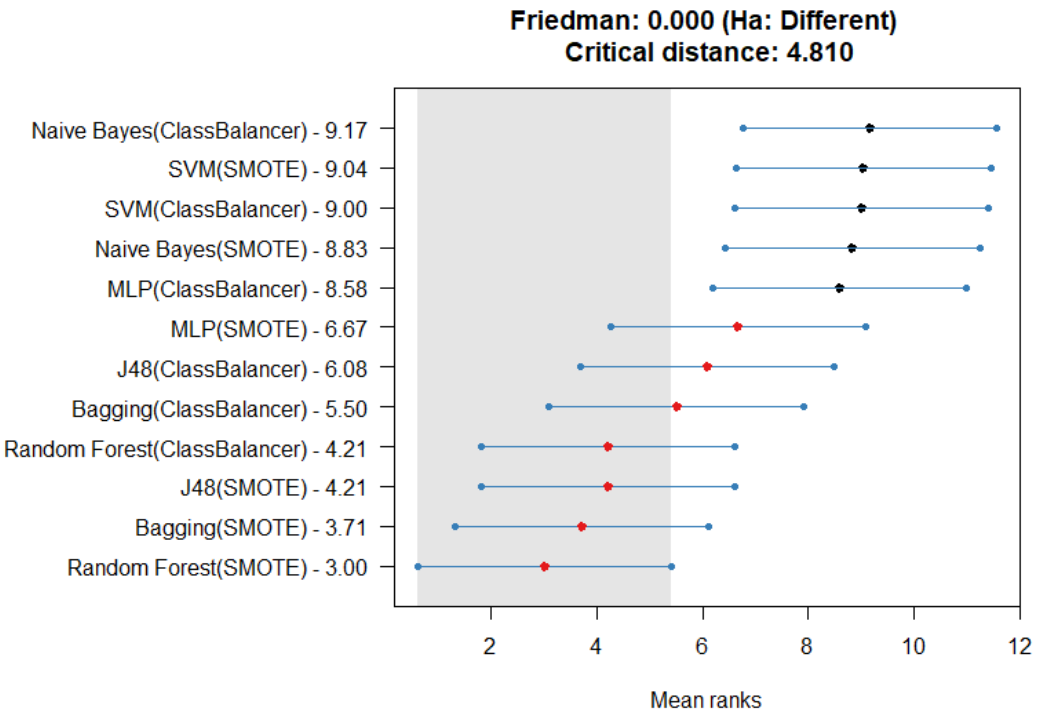

Figure 5.6: Ranking of classifiers using f-measure

5.3.2. Overall Dataset comparison

Next to the classifiers, the datasets are also compared to see which one performs best. For this, table 5.11 is used. This table consists of the ranking of each dataset per classifier. Each row represents a classifier and each column represents a dataset.

Using the Friedman test again, p-value $= 2.2e^{-16} < \alpha = 0.05$, thus the differences between the f-measure values are statistically significant. From the post-hoc tests it is clear that the class J2EE Bad Practices: Threads dataset performs the best. Both Password Management: Hardcoded Password datasets performed the worst with the Method System Information Leak: Internal as third worst. Between the class-level and the method-level datasets, the class-level dataset has a higher rank than the method-level dataset. The ranking of the post-hoc tests can be seen in table 5.12. You can see from figure 5.6, that the critical distance is 4.810. This means that the datasets with that amount of distance have a large enough statistical difference. So, the class-level J2EE Bad Practices: Threads dataset has a statistical difference with everything above method-level J2EE Bad Practices: Threads, meaning that the difference is not by chance alone. Every dataset in the grey area, however, has a low critical distance with the best performing dataset. Meaning that they do not have a statistical difference with the class-level J2EE Bad Practices: Threads dataset. Thus, the difference between class level and method level are not statically different enough, even though the class-level dataset has an overall better performance. The same goes for the best two performing vulnerability types (J2EE Bad Practices: Threads and Path Manipulation), compared to the class-level and method-level dataset.

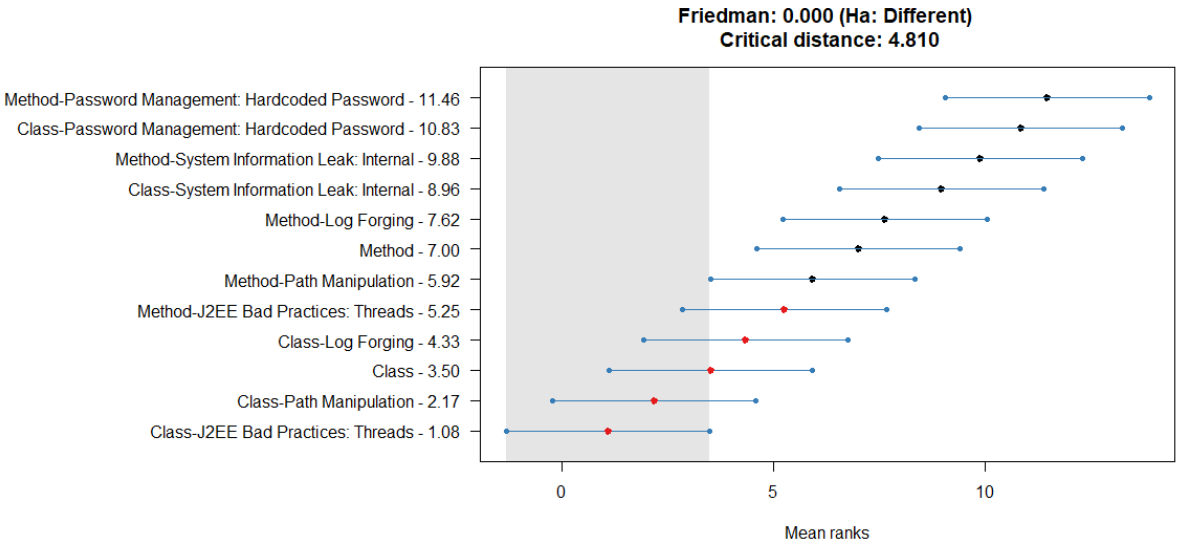

Figure 5.7: Ranking of datasets using f-measure

5.3.3. Research Question Answer

Taking in the results from this section, the last research question can be answered:

RQ3: *How accurate is a model built on a dataset with various types of vulnerabilities compared to models built on datasets per vulnerability type?*

Depending on the type of vulnerability, datasets per vulnerability type can have better performance than a dataset with various vulnerability types. J2EE Bad Practices: Threads and Path Manipulation have an overall better performance, than the datasets with the various vulnerability types. This might be due to the fact that these types of vulnerabilities are better represented by the code metrics used in the datasets.

Classifier	Random Forest (SMOTE)	J48 (SMOTE)	Naive Bayes (SMOTE)	SVM (SMOTE)	MLP (SMOTE)	Bagging (SMOTE)	Random Forest (ClassBalancer)	J48 (ClassBalancer)	Naive Bayes (ClassBalancer)	SVM (ClassBalancer)	MLP (ClassBalancer)	Bagging (ClassBalancer)
Class	0.595(6.0)	0.599(3.5)	0.232(9.0)	0.079(12.0)	0.416(7.0)	0.599(3.5)	0.612(2.0)	0.598(5.0)	0.199(11.0)	0.208(10.0)	0.337(8.0)	0.618(1.0)
Method	0.412(1.0)	0.31(5.0)	0.304(6.0)	0(11.5)	0.113(9.0)	0.348(3.0)	0.301(7.0)	0.266(8.0)	0.094(10.0)	0(11.5)	0.035(2.0)	0.291(7.0)
Class-System Information Leak: Internal	0.181(3.0)	0.189(1.0)	0.042(10.0)	0.058(4.0)	0.051(5.5)	0.187(2.0)	0.024(12.0)	0.048(8.0)	0.051(5.5)	0.051(5.5)	0.029(11.0)	0.043(9.0)
Class-J2EE Bad Practices: Threads	0.659(4.0)	0.652(5.0)	0.349(11.0)	0.374(10.0)	0.643(7.0)	0.674(2.0)	0.681(1.0)	0.662(3.0)	0.387(9.0)	0.332(12.0)	0.405(8.0)	0.644(6.0)
Class-Log Forging	0.447(1.0)	0.365(7.0)	0.314(10.0)	0.321(9.0)	0.368(6.0)	0.396(4.0)	0.4(2.5)	0.362(8.0)	0.271(12.0)	0.304(11.0)	0.4(2.5)	0.375(5.0)
Class-Password Management: Hardcoded Password	0(3.5)	0(3.5)	0(3.5)	0(3.5)	0(3.5)	0(3.5)	0(3.5)	0(3.5)	0(3.5)	0.089(1.0)	0.011(2.0)	0(3.5)
Class-Path Manipulation	0.641(1.0)	0.605(3.0)	0.318(10.0)	0.329(9.0)	0.581(4.0)	0.568(5.0)	0.623(2.0)	0.53(7.0)	0.309(12.0)	0.314(11.0)	0.431(8.0)	0.533(6.0)
Method-System Information Leak: Internal	0.112(1.0)	0.087(2.0)	0.014(9.0)	0(12.0)	0.024(7.0)	0.081(3.0)	0.053(6.0)	0.065(4.0)	0.013(10.0)	0.018(8.0)	0.004(11.0)	0.061(5.0)
Method-J2EE Bad Practices: Threads	0.574(2.0)	0.46(5.0)	0.07(11.0)	0.092(8.0)	0.198(7.0)	0.498(4.0)	0.613(1.0)	0.42(6.0)	0.078(10.0)	0.08(9.0)	0.015(12.0)	0.515(3.0)
Method-Password Management: Hardcoded Password	0	0	0	0	0	0	0	0	0	0	0	0
Method-Log Forging	0.277(2.0)	0.267(4.0)	0.063(10.0)	0.065(9.0)	0.12(7.0)	0.275(3.0)	0.286(1.0)	0.218(6.0)	0.068(8.0)	0.051(11.0)	0.02(12.0)	0.262(5.0)
Method-Path Manipulation	0.465(2.0)	0.408(3.0)	0.163(8.0)	0.121(10.0)	0.188(7.0)	0.458(1.0)	0.222(4.0)	0.211(5.0)	0.157(9.0)	0.11(11.0)	0.03(12.0)	0.202(6.0)

Table 5.9: F-Measure of classifiers with ranking

Classifier	Random Forest (SMOTE)	J48 (SMOTE)	Naive Bayes (SMOTE)	SVM (SMOTE)	MLP (SMOTE)	Bagging (SMOTE)	Random Forest (ClassBalancer)	J48 (ClassBalancer)	Naive Bayes (ClassBalancer)	SVM (ClassBalancer)	MLP (ClassBalancer)	Bagging (ClassBalancer)
p-value(rank)	0.3823333(1.0)	0.3525(3.0)	0.2068333(8.0)	0.1386667(12.0)	0.2651667(7.0)	0.3673333(2.0)	0.3363333(4.0)	0.3226667(6.0)	0.167(10.0)	0.164(11.0)	0.2028333(9.0)	0.3285(5.0)

Table 5.10: Post-hoc tests ranking of classifiers

Dataset	Class	Method	Class- System Information Leak: Internal	Class- J2EE Bad Practices: Threads	Class- Log Forging	Class- Password Management: Hardcoded Password	Class- Path Manipulation	Method- System Information Leak: Internal	Method- J2EE Bad Practices: Threads	Method- Password Management: Hardcoded Password	Method- Log Forging	Method- Path Manipulation
Random Forest (SMOTE)	0.595(3.0)	0.412(7.0)	0.181(9.0)	0.659(1.0)	0.447(6.0)	0(11.5)	0.641(2.0)	0.112(10.0)	0.574(4.0)	0(11.5)	0.277(8.0)	0.465(5.0)
J48 (SMOTE)	0.599(3.0)	0.31(7.0)	0.189(9.0)	0.652(1.0)	0.365(6.0)	0(11.5)	0.605(2.0)	0.087(10.0)	0.46(4.0)	0(11.5)	0.267(8.0)	0.408(5.0)
Naive Bayes (SMOTE)	0.232(5.0)	0.304(4.0)	0.042(9.0)	0.349(1.0)	0.314(3.0)	0(11.5)	0.318(2.0)	0.014(10.0)	0.07(7.0)	0(11.5)	0.063(8.0)	0.163(6.0)
SVM (SMOTE)	0.079(6.0)	0(9.5)	0.058(8.0)	0.374(1.0)	0.321(3.0)	0(9.5)	0.329(2.0)	0(9.5)	0.092(5.0)	0(9.5)	0.065(7.0)	0.121(4.0)
MLP (SMOTE)	0.416(3.0)	0.113(8.0)	0.051(9.0)	0.643(1.0)	0.368(4.0)	0(11.5)	0.581(2.0)	0.024(10.0)	0.198(5.0)	0(11.5)	0.12(7.0)	0.188(6.0)
Bagging (SMOTE)	0.599(2.0)	0.348(7.0)	0.187(9.0)	0.674(1.0)	0.396(6.0)	0(11.5)	0.568(3.0)	0.081(10.0)	0.498(4.0)	0(11.5)	0.275(8.0)	0.458(5.0)
Random Forest (ClassBalancer)	0.612(4.0)	0.301(7.0)	0.024(10.0)	0.681(1.0)	0.4(6.0)	0(11.5)	0.623(2.0)	0.053(5.0)	0.613(3.0)	0(11.5)	0.286(8.0)	0.222(9.0)
J48 (ClassBalancer)	0.598(2.0)	0.266(6.0)	0.048(10.0)	0.662(1.0)	0.362(5.0)	0(11.5)	0.53(3.0)	0.065(9.0)	0.42(4.0)	0(11.5)	0.218(7.0)	0.211(8.0)
Naive Bayes (ClassBalancer)	0.199(4.0)	0.094(6.0)	0.051(9.0)	0.387(1.0)	0.271(3.0)	0(11.5)	0.309(2.0)	0.013(10.0)	0.078(7.0)	0(11.5)	0.068(8.0)	0.157(5.0)
SVM (ClassBalancer)	0.208(4.0)	0(11.5)	0.051(8.5)	0.332(1.0)	0.304(3.0)	0.089(6.0)	0.314(2.0)	0.018(10.0)	0.08(7.0)	0(11.5)	0.051(8.5)	0.11(5.0)
MLP (ClassBalancer)	0.337(4.0)	0.035(5.0)	0.029(7.0)	0.405(2.0)	0.4(3.0)	0.011(10.0)	0.431(1.0)	0.004(11.0)	0.015(9.0)	0(12.0)	0.02(8.0)	0.03(6.0)
Bagging (ClassBalancer)	0.618(2.0)	0.291(6.0)	0.043(10.0)	0.644(1.0)	0.375(5.0)	0(11.5)	0.533(3.0)	0.061(9.0)	0.515(4.0)	0(11.5)	0.262(7.0)	0.202(8.0)

Table 5.11: F-Measure of datasets with ranking

Dataset	Class	Method	Class- System Information Leak: Internal	Class- J2EE Bad Practices: Threads	Class- Log Forging	Class- Password Management: Hardcoded Password	Class- Path Manipulation	Method- System Information Leak: Internal	Method- J2EE Bad Practices: Threads	Method- Password Management: Hardcoded Password	Method- Log Forging	Method- Path Manipulation
p-value(rank)	0.42(3.0)	0.2478333(7.0)	0.118(9.0)	0.5585(1.0)	0.3685(4.0)	0(11.5)	0.507(2.0)	0.053(10.0)	0.3153333(5.0)	0(11.5)	0.1778333(8.0)	0.3005(6.0)

Table 5.12: Post-hoc tests ranking of datasets

6

Discussion

This chapter will discuss the results from the previous chapter, by summing up the main findings and comparing the results with the related work. Next to that, the implications of the results are discussed.

6.1. Main findings

Bagging with ClassBalancer is the best classifier for the class dataset. Comparing the f-measure gives the highest f-measure for Bagging with ClassBalancer technique. This also gave the best ROC AUC score. The highest precision could be found by Random Forest with ClassBalancer and the highest recall with J48 with ClassBalancer.

Random Forest with SMOTE is the best classifier for the method dataset. Comparing the f-measure gives the highest f-measure for Random Forest with SMOTE technique, which also gave the best ROC AUC score and the highest precision. The highest recall could be found by MLP with ClassBalancer.

Class-level dataset performs better than method-level dataset. The highest f-measure was found in the class-level dataset. Using the post-hoc test to compare the overall performance, the class-level dataset outperformed the method-level dataset again. The statistical difference was however not large enough.

Random Forest with SMOTE is the overall best algorithm. Comparing the f-measures of all the datasets and algorithms using the post-hoc test, the highest ranked algorithm is Random Forest with SMOTE.

SMOTE is the better overall technique for sampling. Looking at the ranking given by the post-hoc test for comparing the classifiers, the top two techniques uses SMOTE as sampling technique to overcome the class imbalance problem.

The Class-J2EE Bad Practises: Threads dataset gives the best overall performance. Using the post-hoc test to compare all the datasets, the Class-J2EE Bad Practises: Threads is ranked the highest. Thus, this dataset gives the highest f-measure.

Datasets per vulnerability perform better, depending on the vulnerability type. The datasets built for the vulnerabilities J2EE Bad Practises: Threads and Path Manipulation, have better performance than the datasets with all vulnerabilities. This applies for both the class as well as the method dataset. The other datasets per vulnerability type perform worse than the datasets with all the vulnerabilities.

Naive Bayes with ClassBalancer is the fastest. This algorithm takes the least time to

train and test and create a model in all different cases.

Overall performance is not very good. The highest f-measure is just under 0.7, which is for the Class-J2EE Bad Practises: Threads dataset. For the class dataset the highest f-measure is 0.62. These are both not just slightly good performance, but it still means that many cases are wrongly classified. In most cases of high f-measure, the recall score is good, but the precision is low. This means that the classifier detects many false vulnerabilities, but also predicts many true vulnerabilities as false vulnerabilities.

Less false positives and higher precision are found than stand-alone Fortify. The number of false positives compared to Fortify as stand alone is for both the class-level and method-level dataset lower. The precision is also higher for both the class-level and method-level dataset. Only for the method-level dataset is the lowest precision equal to that of Fortify. For the datasets per vulnerability, the highest precision are all higher than that of Fortify for both class and method. The only exception is the dataset for Password Management: Hardcoded Password. For both the class and method dataset of that vulnerability, the precision is lower than the precision of Fortify as stand alone.

6.2. Comparison with Related Work

Ghaffarian and Shahriari [27] mentioned, many studies were conducted on using code metrics to predict vulnerabilities. The following topics were lacking in current research, and were tackled in this research:

- **Imbalance class data:** Most research does not deal with the class imbalance problem. They either do not have that problem or do not deal with it. In this research we overcome this problem by using two sampling techniques, SMOTE and ClassBalancer.
- **Cross-project studies:** Studies usually use code metrics from the same project. Moshtari et al. [50] showed promising results for both within-project as well as cross-project vulnerability detection. Within-project did have a better result. Our study created datasets from various projects within ING.
- **Class level:** Catal and Diri [12] found that most research, uses code metrics on method level. We compared both class and method-level datasets with code metrics. In our case the class-level datasets perform better than the method-level datasets. Thus, more research should be done on class level, for this results in better performance of detecting true vulnerabilities from static analyzer warnings.
- **Closed sourced:** Open source projects are mostly used in studies for vulnerability detection with code metrics [12]. In our case, we used data from industry.

Overall, our conclusion is similar to that of related work, even though we predict true vulnerabilities from static analyzer warnings. Code metrics can help find vulnerabilities, but they still miss many vulnerabilities and code metrics that are more security related might help overcome this problem.

6.3. Implications

Looking at the results, we found that supervised learning in combination with code metrics does indeed help in finding vulnerabilities. To that end, we can sum up the implications for design and recommendations for the pentesters and developers.

6.3.1. Design

The following design implications can be deducted from the results of this study:

- **Class level:** When creating a code metrics dataset for detecting vulnerabilities, it is better to extract code metrics on class-level rather than method-level. Manually checking a class does, however, take more time than manually checking a method.

- **Random Forest:** Choosing the right supervised learning technique is crucial to getting a good performance. Random Forest was deemed the overall best classifier.
- **SMOTE:** For sampling, the better technique between ClassBalancer and SMOTE in this case is SMOTE. Using SMOTE for sampling will help overcome the class imbalance problem.
- **Dataset per vulnerability type:** When using the supervised learning model with Fortify, it might help to create datasets based on vulnerability type. This might increase the performance depending on the vulnerability compared to using a dataset with all kinds of vulnerabilities.

6.3.2. Recommendations

With Fortify giving such a high false positive rate, it slows down the process of deploying software. With the results from this study at hand, we can give the following recommendations to both pentesters and developers:

- **Log negatives:** Missing both false and true negatives makes it hard to decide whether or not Fortify is doing a good job. Pentesters need to ensure that classes deemed not vulnerable by Fortify are really not vulnerable. These classes need to be added to the dataset. This will also help train a supervised learning model by adding this information to the dataset.
- **Extract code metrics:** Developers can extract the code metrics on class level themselves after writing code to create a dataset. Additional code and security metrics can be researched and added. This dataset can then be used for aiding vulnerability detection in combination with Fortify. By making developers deliver the dataset of code metrics, the workload for pentester would not increase when using supervised learning with Fortify.
- **Use a supervised learning algorithm:** Pentesters should use a supervised learning algorithm next to using Fortify. This way there are fewer false positives they have to manually check. By expanding the dataset given by developers with data from Fortify, they can either use one of the machine learning techniques from this research or do research in a supervised learning technique with even better performance. Either way, the speed of deployment will increase when combining Fortify with supervised learning.

7

Threats to Validity

During the research, some threats to validity was observed. This chapter will discuss these limitations by separating the threats to validity in two different categories: internal and external validity.

7.1. Internal Validity

Internal validity concerns the conditions of the experiment and how that influence the outcome.

The labels for the datasets were provided by the pentesters at ING. These labels were added manually and can thus be biased as what the pentesters deem vulnerable or not. Fortify indicates what might be vulnerable in the code and why, but it is up to the developers and pentesters to figure out what they deem actual vulnerable. Thus, the labels for the dataset are not 100% independent and unbiased.

Because the data was extracted from the Fortify web server, the false and true negatives were missing, for the Fortify web server only has the true and false positives. The true vulnerabilities that Fortify did not detect were not accessible to us and could thus not be included in this research. Next to the false negatives, true negatives are also missing. This because Fortify does not state what it actually deems not vulnerable. If we had full access to the projects, we could have deemed all the other classes that Fortify did not flag as true negative in case these classes were checked by the pentesters. Thus, the full performance of Fortify could not be taken into account, which makes it difficult to compare the performance of Fortify with the other classifiers.

Another internal threat is the fact that the datasets are unbalanced. This problem would cause the models being biased toward the majority class. To overcome this problem, special sampling techniques were used like SMOTE, so that the data would be more in balance.

To compare the performance of the classifiers, an evaluation metric must be selected. With different evaluation metrics available and each of them representing a different aspect of the performance, it is hard to choose one to compare the classifiers. After going through different literature, the f-measure metric seems the most used for comparing the performance of different classifiers when dealing with imbalanced data.

Because of the non-deterministic nature of Random Forest and k-fold cross-validation, it is better to perform the experiment multiple times and then look at the average. After seeing no significant difference in the performance, when doing some of the experiments multiple times, we dropped this part of experiment.

7.2. External Validity

External validity concerns the generalization of the outcome of the experiment.

Because the study was done with data from ING, it is important to consider that there are some limitations to the generalization of the experiment. Because the data comes from one

organization, it has to be considered that the results of this research might be more valid for similar companies who operate in the same way and are specialized in the same type of field as ING.

Next to that, the data was generated from the static analysis tool Fortify. The performance and results found in this research might thus be different from other static analysis tools.

8

Conclusion & Future Work

This research consisted of predicting true vulnerabilities from static analyzer warnings, using supervised learning in an industrial setting. With the evaluated results at hand, the initial research questions can thus be answered.

> **RQ1:** *How accurate are machine learning methods in detecting true vulnerabilities from static analyzer warnings using a dataset from ING?*

Six different classifiers have been used, with two different sampling methods. From these classifiers, it is clear that Bagging in combination with ClassBalancer gives the best results in terms of f-measure for the class dataset. For the method level, the best performing classifier was Random Forest with SMOTE.

With a f-measure of 0.618, the model for the class-level dataset can be considered slightly good. The model for the method-level dataset has a f-measure of 0.412, which is weak.

> **RQ2:** *How do classification methods perform at different levels of granularity with a dataset from ING?*

Two datasets were created, one on class level and one on method level. Bagging with SMOTE was considered the best overall classifier between the class and method datasets. Comparing the two datasets, the class-level dataset scored an overall higher f-measure compared to method-level dataset.

> **RQ3:** *How accurate is a model built on the entire dataset compared to several models built per vulnerability type in detecting true vulnerabilities from static analyzer warnings using a dataset from ING?*

Comparing the two datasets with the individual datasets per vulnerability, gives Random Forest with SMOTE as the best overall classifier. The class-level J2EE Bad Practices: Threads dataset is deemed the dataset with the best results. Overall, the class-level dataset performs better, except for the Password Management: Hardcoded Password datasets, due to the fact that there are too few true vulnerabilities in these datasets.

In the end, the performance of these classifiers and datasets can be considered bad to slightly good. This because the overall performance of the classifiers are not really high. Compared to the outcome of only using the Fortify tool, currently used by ING, the false positives are lower and the precision of the classifiers are actually quite higher. Using supervised learning combined with Fortify could aid in speeding up the deployment process within ING and could potentially save some time and effort for pentesters and developers alike.

8.1. Future Work

Even though this research is complete, there are several ways these research can be either improved on or extended. This chapter will discuss the various limitations and topics that

could enhance this research by future work.

8.1.1. Limitations

First the limitations of this research are discussed. These limitations could be improved on in future work.

Full Projects

The datasets used in this research, were built by extracting data from the static analysis tool Fortify. This tool consists of Java classes and vulnerabilities within these classes. The full projects that these classes originated from were however, not accessible. Thus, coupling metrics were mostly incomplete. For future work, it would therefore be interesting to use the full projects of the software in development, instead of only the classes there are flagged by Fortify and compare those results with the current results.

Negatives in Fortify

From the Fortify tool, it was possible to extract false and true vulnerabilities. These are the false and true positives. However, there was no access to the false negatives. These are the classes that Fortify did not flag as vulnerable, but the pentester or developer found vulnerable by either manually checking or using some other tool. Fortify does have a feature to add these findings, but unfortunately these were restricted and could therefore not be extracted. True negatives for the class-level dataset would be the other classes within the full project, for which access was also restricted as discussed in the previous subsection 8.1.1. For the method-level dataset, all the methods that do not have a vulnerability could be considered true negative.

8.1.2. Security Metrics

The datasets used in this study only used code metrics. Because we are dealing with vulnerability warnings, it might be better to use security-related metrics. As related work has shown, security metrics tend to give better results. Thus, for future research, security metrics should be used to perform this kind of study.

8.1.3. Fuzzing

When access to full projects is granted and they can be compiled and executed, it would be interesting to combine the machine learning prediction with fuzzing. By using machine learning to predict the high risk vulnerabilities and fuzzing for validating these predictions. This way, the whole process of finding vulnerabilities can be automated. The advantage of fuzzing is that it has low false positives, for all the findings are actual crashes. The downside is that it can take quite some time to go through all the random inputs. The advantage of machine learning is that it is fast. However, it can give many false positives. By using the output of the classifier as input for fuzzing, the fuzzing can be sped up.

8.1.4. Deep learning

This research focused only on supervised classification. Man [41] used unsupervised clustering for predicting true vulnerabilities from static analyzer warnings, using the class-level dataset. For future work, one could consider using deep learning. Deep learning has the potential to have the best results. The downsides of deep learning is that it is hard to explain the results, for it is a black box, and it is expensive, for deep learning requires much power to perform and a larger set of data points.

A

Additional Graphs & Tables

A.1. Code metrics

Metric	Description
Type	The type of the instance.
CBO(Coupling Between Objects)	Counts the number of dependencies a class has, ignoring dependencies to Java itself.
WMC(Weight Method Class)	Counts the number of branch instructions in a class.
DIT(Depth Inheritance Tree)	Counts the number of ”fathers” a class has.
RFC(Response For Class)	Counts the number of unique method invocations in the class.
LCOM(Lack of Cohesion of Methods)	Number of method pairs who are not similar.
totalMethods	Total amount of methods in the class.
staticMethods	Total amount of static methods in the class.
publicMethods	Total amount of public methods in the class.
privateMethods	Total amount of private methods in the class.
protectedMethods	Total amount of protected methods in the class.
defaultMethods	Total amount of default methods in the class.
abstractMethods	Total amount of abstract methods in the class.
finalMethods	Total amount of final methods in the class.
synchronizedMethods	Total amount of synchronized methods in the class.
totalFields	Total amount of fields in the class.
staticFields	Total amount of static fields in the class.
publicFields	Total amount of public fields in the class.
privateFields	Total amount of private fields in the class.
protectedFields	Total amount of protected fields in the class.
defaultFields	Total amount of default fields in the class.
finalFields	Total amount of final fields in the class.
synchronizedFields	Total amount of synchronized fields in the class.
NOSI(Number of Static Invocations)	Counts the number of invocations to static methods.
LOC(Lines Of Code)	Counts the lines of actual code in the class.
returnQty	Counts the number of return instructions.
loopQty	Counts the number of loops.
comparisonsQty	Counts the number of comparisons.
tryCatchQty	Counts the number of try/catches.
parenthesizedExpsQty	Counts the number of expressions inside parenthesis.
stringLiteralsQty	Counts the number of string literals.
numbersQty	Counts the amount of numbers literals.
assignmentsQty	Counts the amount of assignment statements.
mathOperationsQty	Counts the number of math operations.
variablesQty	Counts the number of declared variables.
maxNestedBlocks	Highest number of blocks nested together.
anonymousClassesQty	Counts the number of anonymous classes.
subClassesQty	Counts the number of subclasses.
lambdasQty	Counts the number of lambda expressions.
uniqueWordsQty	Counts the number of unique words in the source code.
modifiers	Counts the number of modifiers in the class.
Vulnerable	Label of the instance. False if not vulnerable, True if vulnerable.
Vulnerability Type	The type of the vulnerability.
Vulnerability Location	The line number of the vulnerability.
Nested	The level of nesting of the line of code of the potential vulnerability in the code.

Table A.1: Metrics for the class-level dataset.

Metric	Description
method	The name of the method.
constructor	Boolean if the method is a constructor.
line	Line number of the method.
CBO(Coupling Between Objects)	Counts the number of dependencies a method has, ignoring dependencies to Java itself.
WMC(Weight Method Class)	Counts the number of branch instructions in a method.
RFC(Response For Class)	Counts the number of unique method invocations in the method.
LOC(Lines Of Code)	Counts the lines of actual code in the method.
returns	Counts the number of return instructions.
variables	Counts the number of declared variables.
parameters	Counts the number of parameters.
startLine	The line number where the method starts, which is the same as the line metric.
loopQty	Counts the number of loops.
comparisonsQty	Counts the number of comparisons.
tryCatchQty	Counts the number of try/catches.
parenthesizedExpsQty	Counts the number of expressions inside parenthesis.
stringLiteralsQty	Counts the number of string literals.
numbersQty	Counts the amount of numbers literals.
assignmentsQty	Counts the amount of assignment statements.
mathOperationsQty	Counts the number of math operations.
maxNestedBlocks	Highest number of blocks nested together.
anonymousClassesQty	Counts the number of anonymous classes.
subClassesQty	Counts the number of subclasses.
lambdasQty	Counts the number of lambda expressions.
uniqueWordsQty	Counts the number of unique words in the source code.
modifiers	Counts the number of modifiers in the class.
Vulnerable	Label of the instance. False if not vulnerable, True if vulnerable.
Vulnerability Type	The type of the vulnerability.
Vulnerability Location	The line number of the vulnerability.
Nested	The level of nesting of the line of code of the potential vulnerability in the code.

Table A.2: Metrics for the method-level dataset.

A.2. Research Results

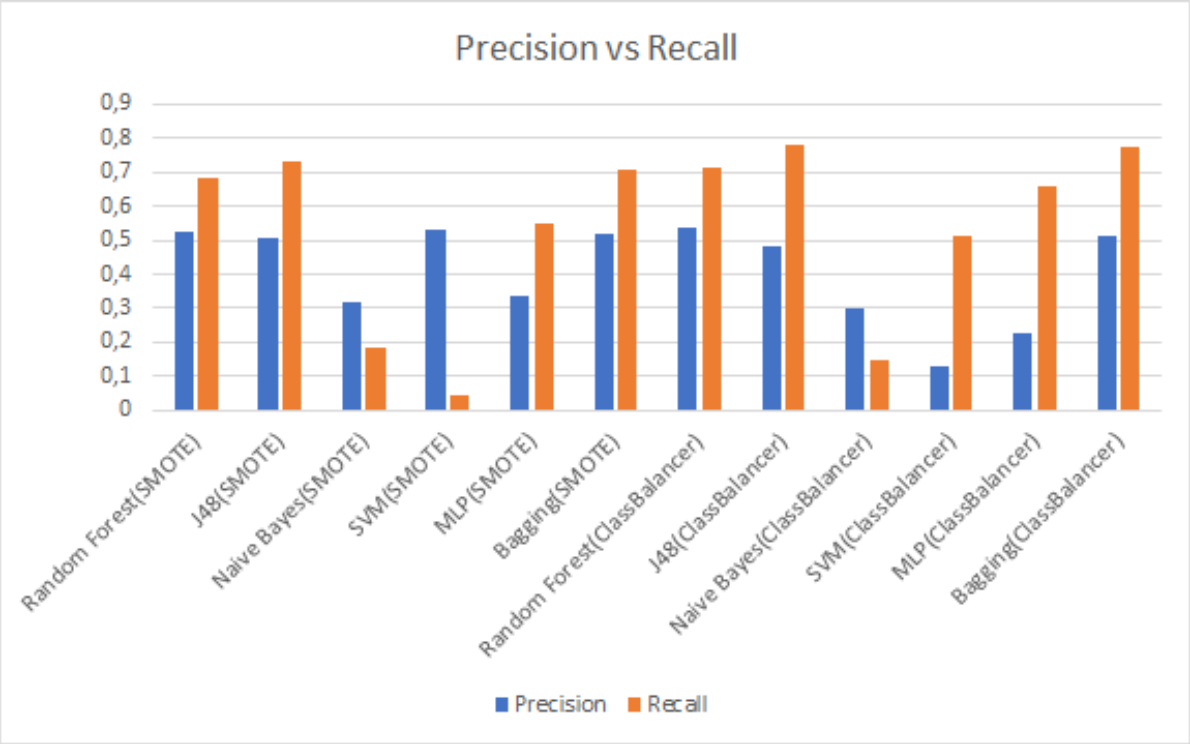

Figure A.1: Precision vs Recall class-level dataset

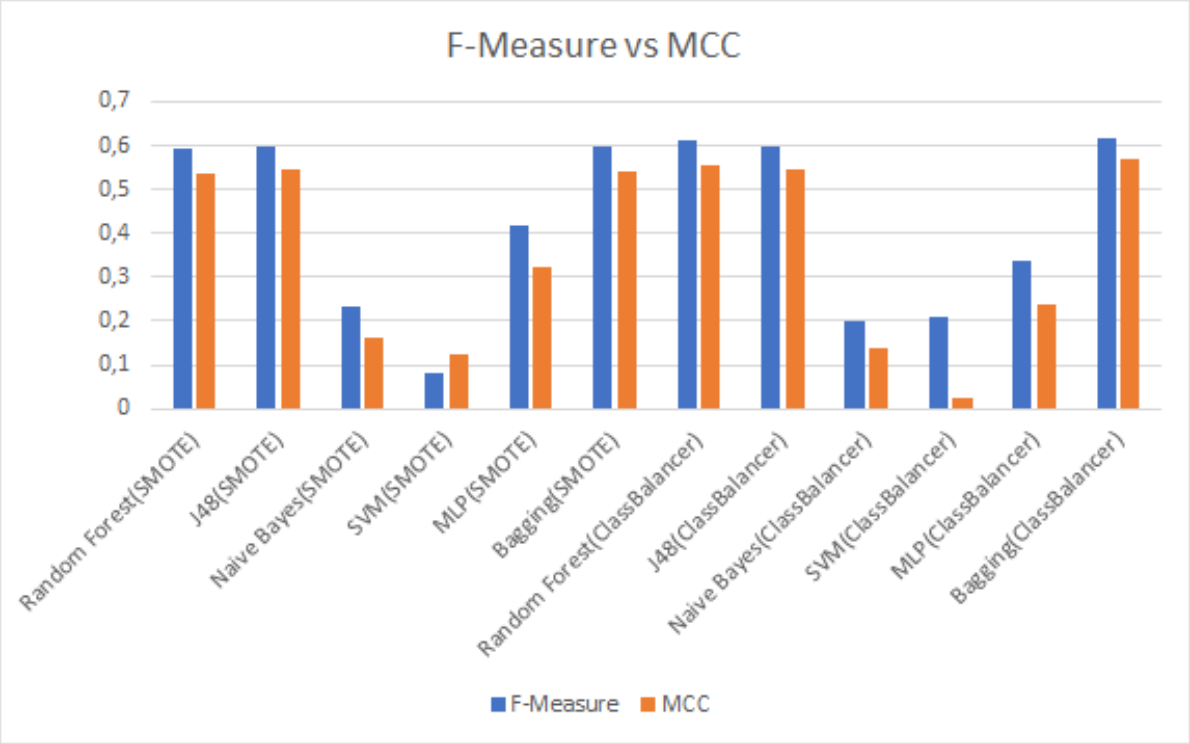

Figure A.2: F-measure vs MCC class-level dataset

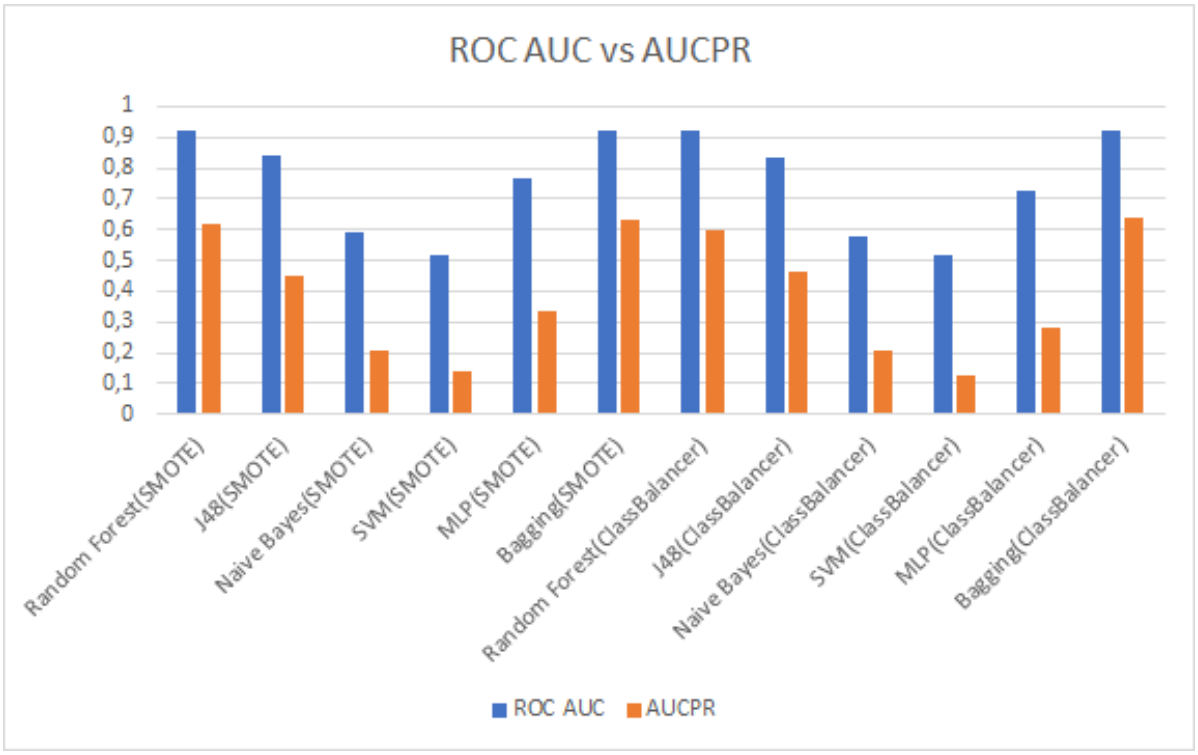

Figure A.3: ROC AUC vs AUCPR class-level dataset

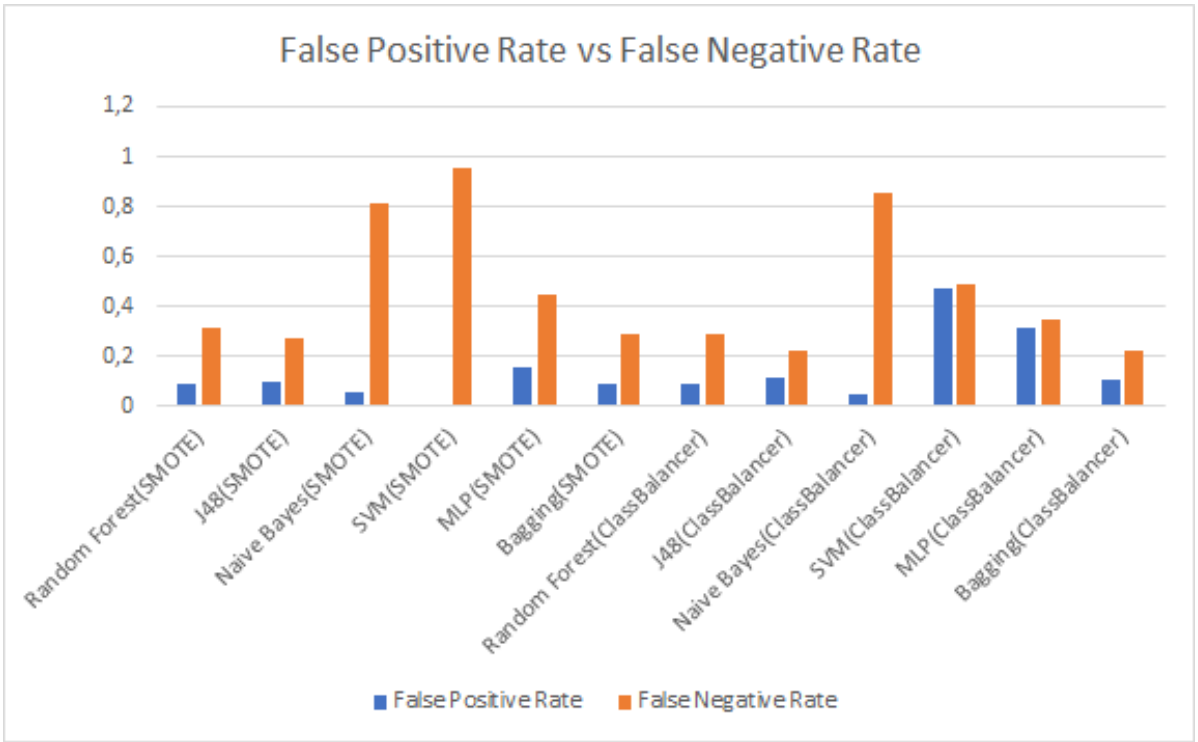

Figure A.4: False Positive Rate vs False Negative Rate class-level dataset

FALSE	TRUE	<– classified as
9682	911	**FALSE**
465	1009	**TRUE**

Table A.3: Confusion Matrix of Random Forest(SMOTE) on class-level dataset

FALSE	TRUE	<– classified as
9545	1048	**FALSE**
396	1078	**TRUE**

Table A.4: Confusion Matrix of J48(SMOTE) on class-level dataset

FALSE	TRUE	<– classified as
10005	588	**FALSE**
1203	271	**TRUE**

Table A.5: Confusion Matrix of Naive Bayes(SMOTE) on class-level dataset

FALSE	TRUE	<– classified as
10538	55	**FALSE**
1411	63	**TRUE**

Table A.6: Confusion Matrix of SVM(SMOTE) on class-level dataset

FALSE	TRUE	<– classified as
8979	1614	**FALSE**
663	811	**TRUE**

Table A.7: Confusion Matrix of MLP(SMOTE) on class-level dataset

FALSE	TRUE	<– classified as
9627	966	**FALSE**
431	1043	**TRUE**

Table A.8: Confusion Matrix of Bagging(SMOTE) on class-level dataset

FALSE	TRUE	<– classified as
9681	912	**FALSE**
421	1053	**TRUE**

Table A.9: Confusion Matrix of Random Forest(ClassBalancer) on class-level dataset

FALSE	TRUE	<– classified as
9361	1232	**FALSE**
321	1153	**TRUE**

Table A.10: Confusion Matrix of J48(ClassBalancer) on class-level dataset

FALSE	TRUE	<– classified as
10072	521	**FALSE**
1254	220	**TRUE**

Table A.11: Confusion Matrix of Naive Bayes(ClassBalancer) on class-level dataset

FALSE	TRUE	<– classified as
5560	5033	**FALSE**
719	755	**TRUE**

Table A.12: Confusion Matrix of SVM(ClassBalancer) on class-level dataset

FALSE	TRUE	<– classified as
7302	3291	**FALSE**
507	967	**TRUE**

Table A.13: Confusion Matrix of MLP(ClassBalancer) on class-level dataset

FALSE	TRUE	<– classified as
9509	1084	**FALSE**
329	1145	**TRUE**

Table A.14: Confusion Matrix of Bagging(ClassBalancer) on class-level dataset

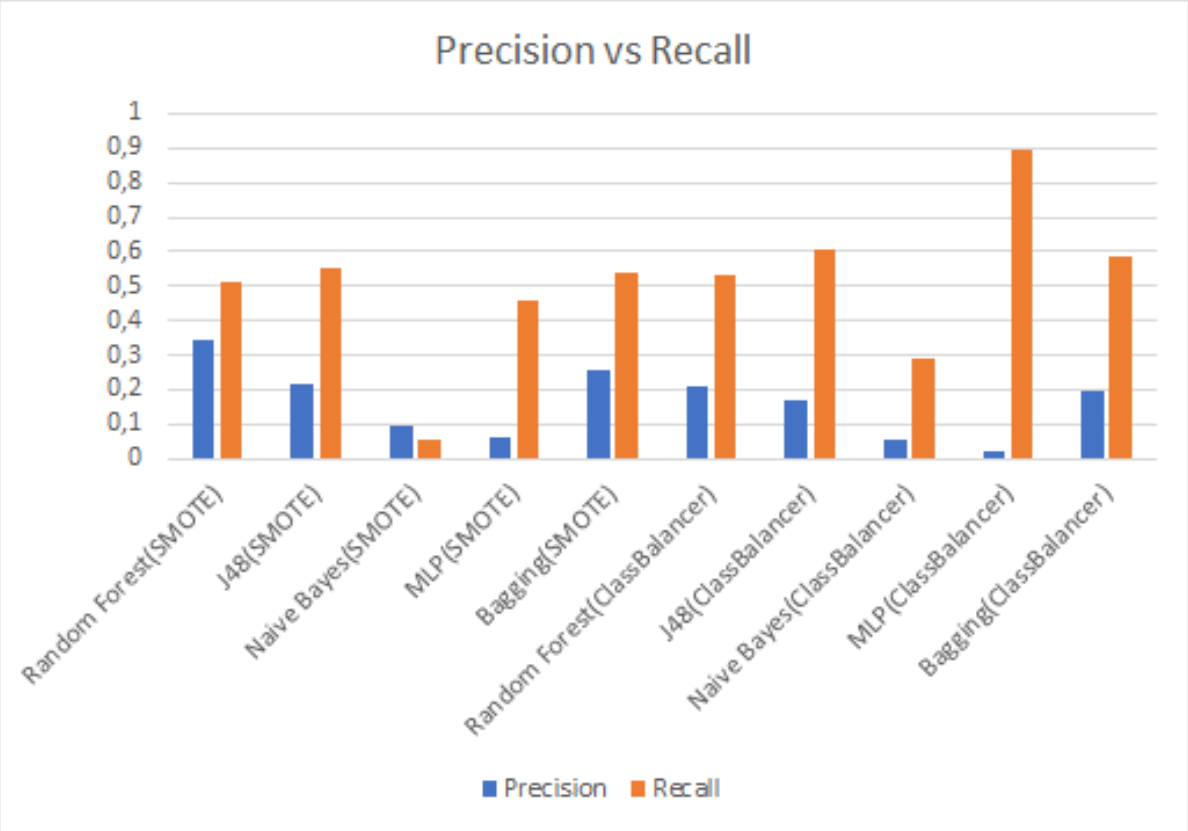

Figure A.5: Precision vs Recall method-level dataset

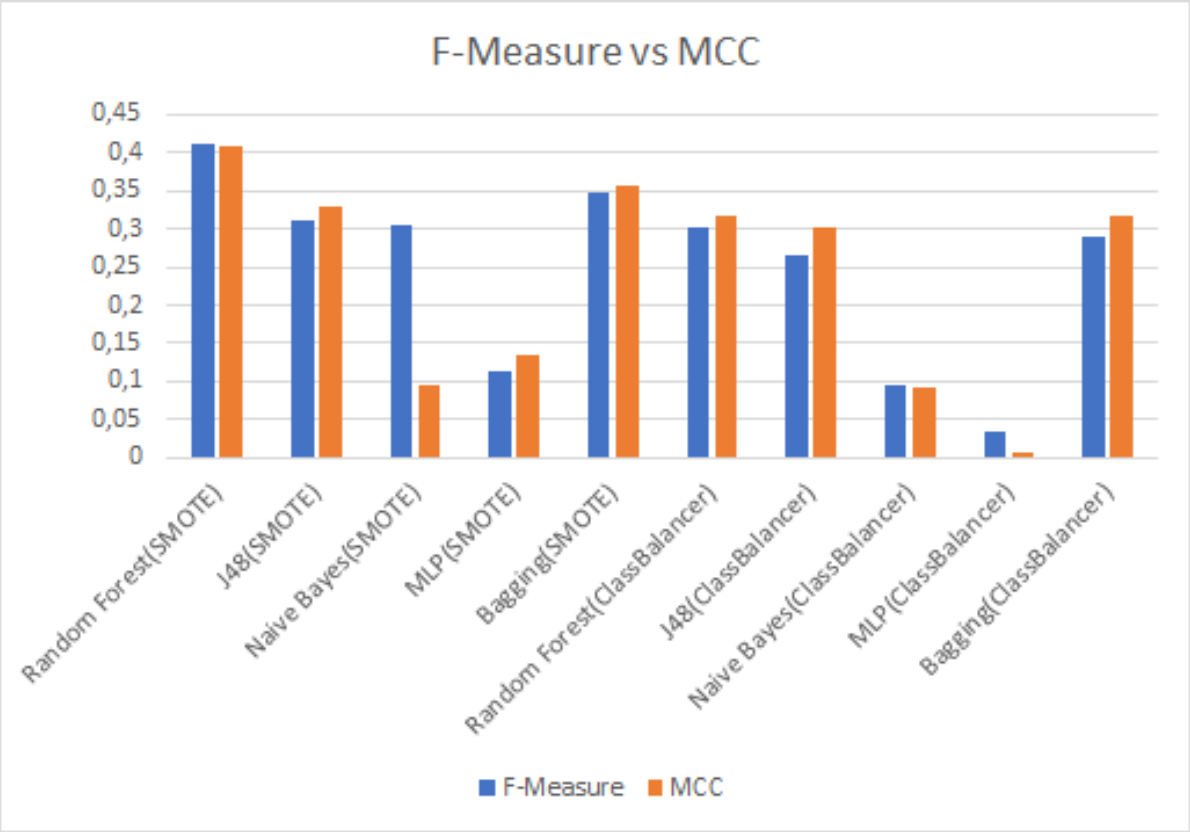

Figure A.6: F-measure vs MCC method-level dataset

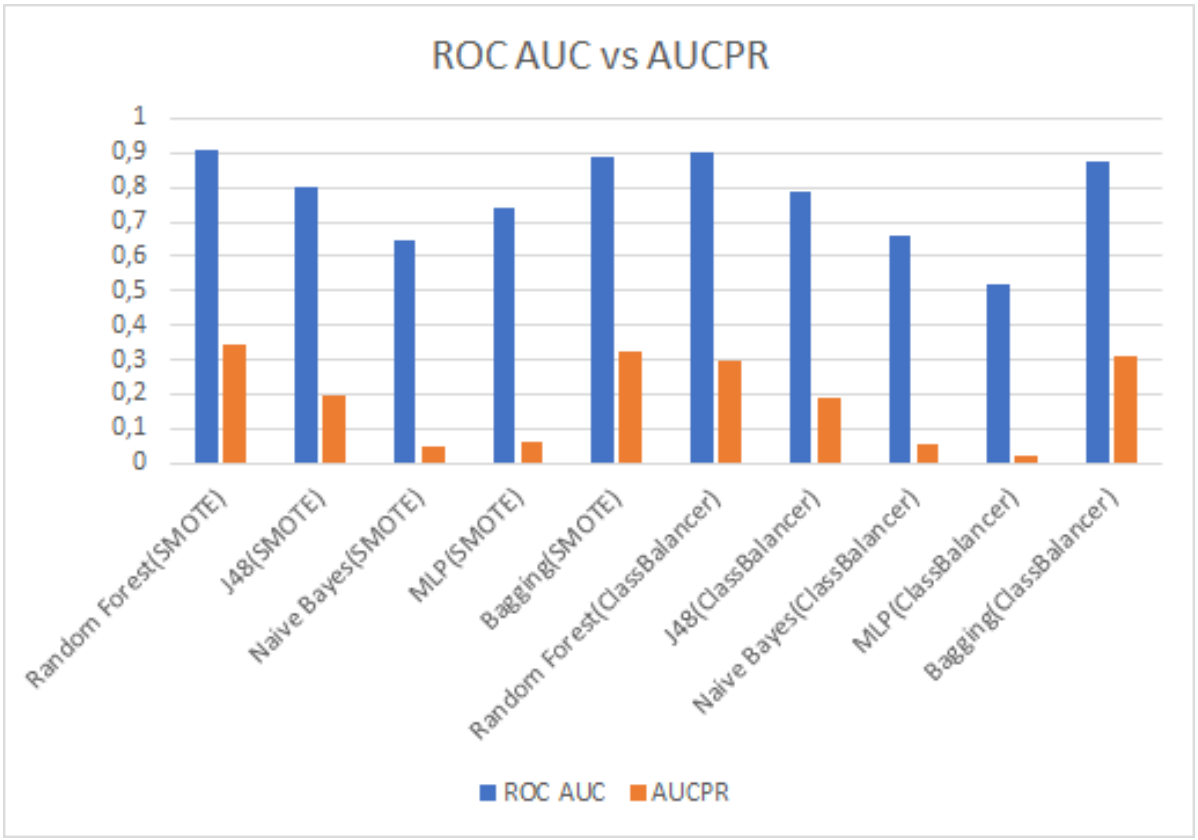

Figure A.7: ROC AUC vs AUCPR method-level dataset

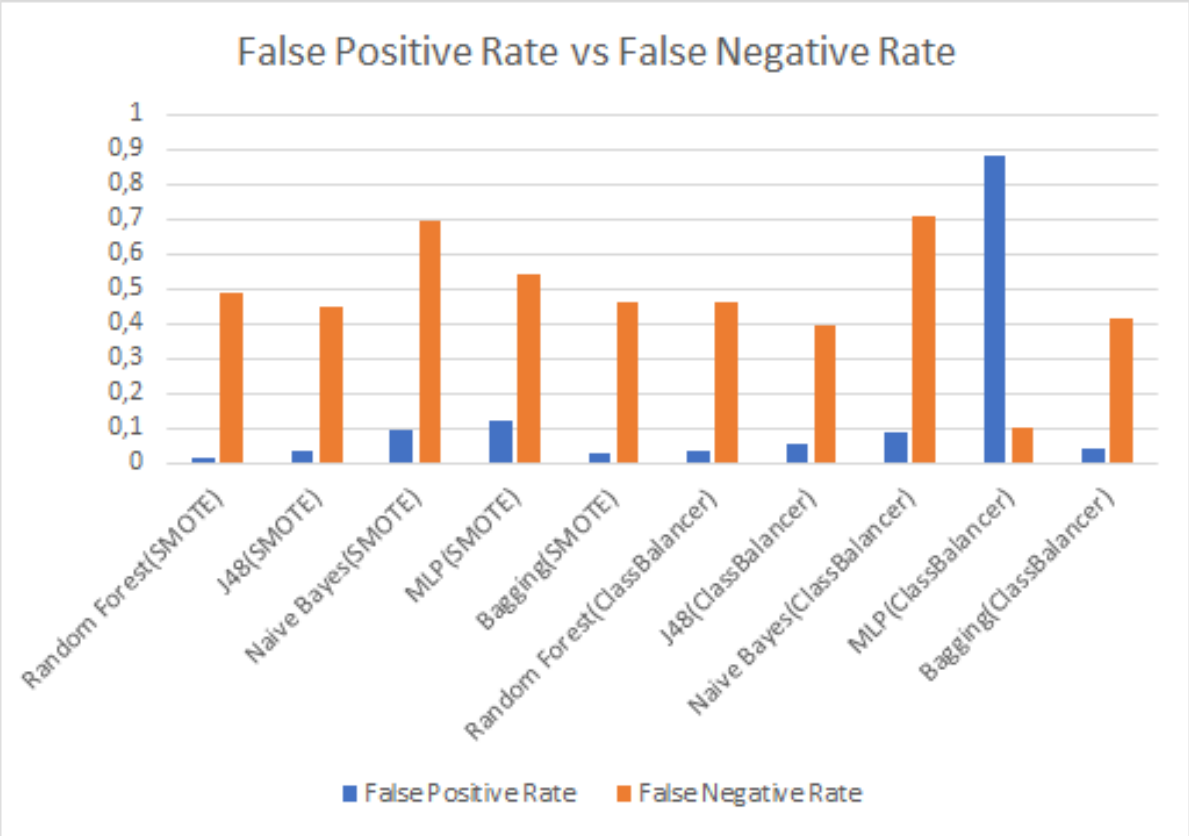

Figure A.8: False Positive Rate vs False Negative Rate method-level dataset

FALSE	TRUE	<– classified as
95976	1696	**FALSE**
849	892	**TRUE**

Table A.15: Confusion Matrix of Random Forest(SMOTE) on method-level dataset

FALSE	TRUE	<– classified as
94172	3500	**FALSE**
778	963	**TRUE**

Table A.16: Confusion Matrix of J48(SMOTE) on method-level dataset

FALSE	TRUE	<– classified as
88594	9078	**FALSE**
1212	529	**TRUE**

Table A.17: Confusion Matrix of Naive Bayes(SMOTE) on method-level dataset

FALSE	TRUE	<– classified as
86118	11554	**FALSE**
948	793	**TRUE**

Table A.18: Confusion Matrix of MLP(SMOTE) on method-level dataset

FALSE	TRUE	<– classified as
94969	2703	**FALSE**
805	936	**TRUE**

Table A.19: Confusion Matrix of Bagging(SMOTE) on method-level dataset

FALSE	TRUE	<– classified as
94163	3509	**FALSE**
810	931	**TRUE**

Table A.20: Confusion Matrix of Random Forest(ClassBalancer) on method-level dataset

FALSE	TRUE	<– classified as
92504	5168	**FALSE**
682	1059	**TRUE**

Table A.21: Confusion Matrix of J48(ClassBalancer) on method-level dataset

FALSE	TRUE	<– classified as
89190	8482	**FALSE**
1237	504	**TRUE**

Table A.22: Confusion Matrix of Naive Bayes(ClassBalancer) on method-level dataset

FALSE	TRUE	<– classified as
11301	86371	**FALSE**
178	1563	**TRUE**

Table A.23: Confusion Matrix of MLP(ClassBalancer) on method-level dataset

FALSE	TRUE	<– classified as
93437	4235	**FALSE**
723	1018	**TRUE**

Table A.24: Confusion Matrix of Bagging(ClassBalancer) on method-level dataset

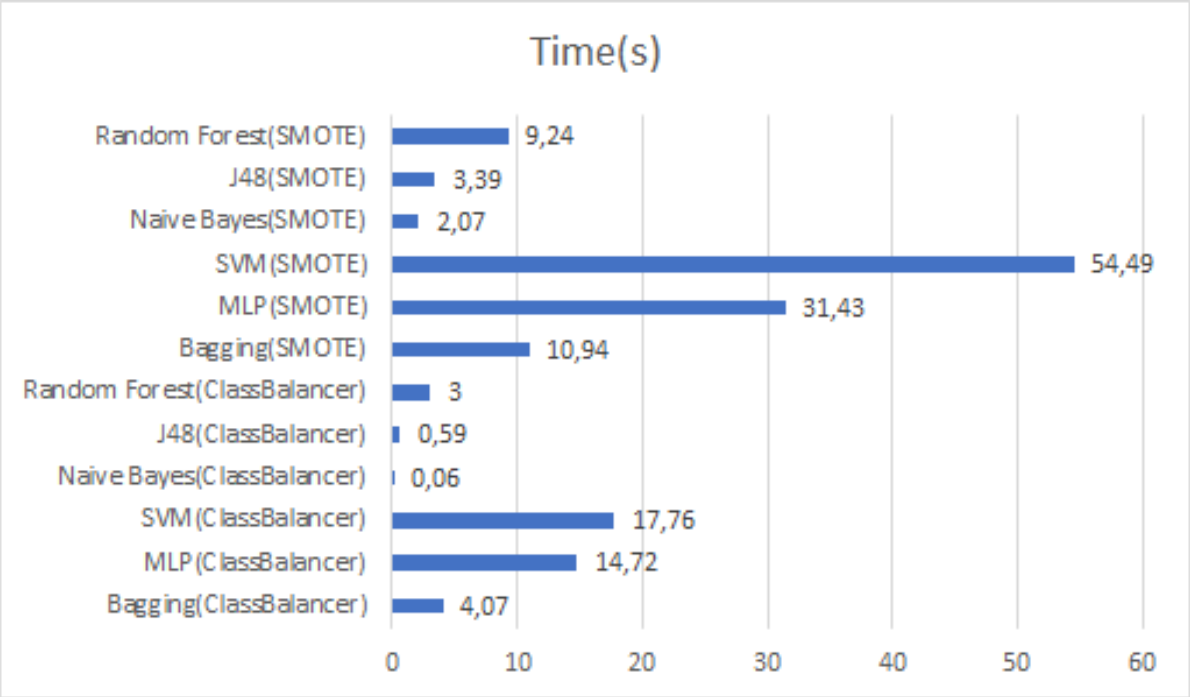

Figure A.9: Time in seconds for each classifier to be build with the class-level dataset

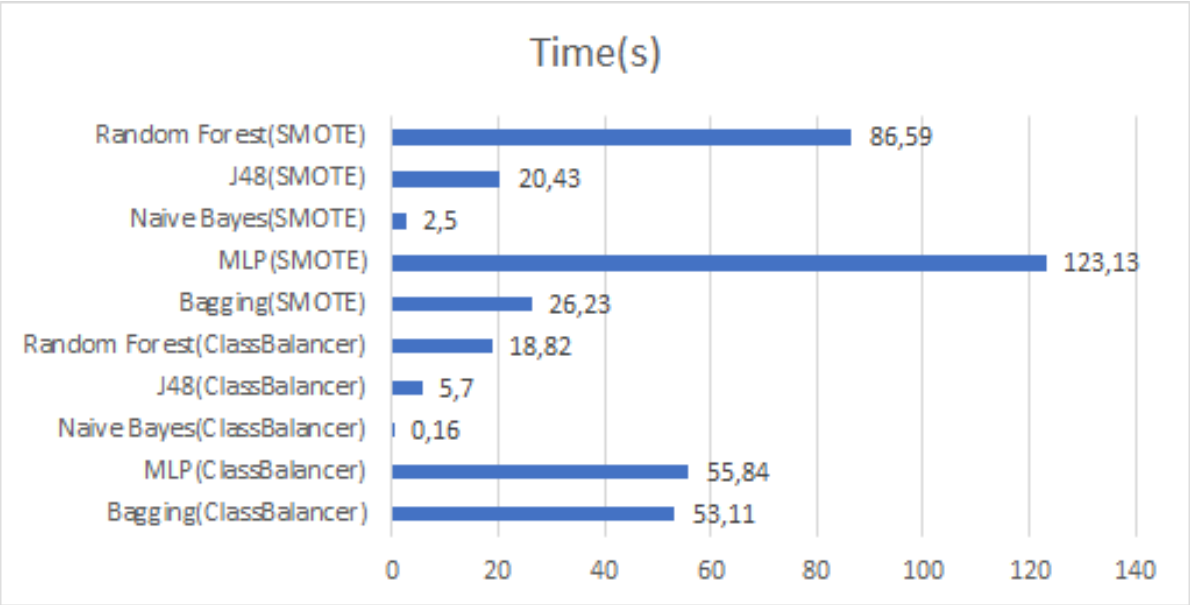

Figure A.10: Time in seconds for each classifier to be build with the method-level dataset

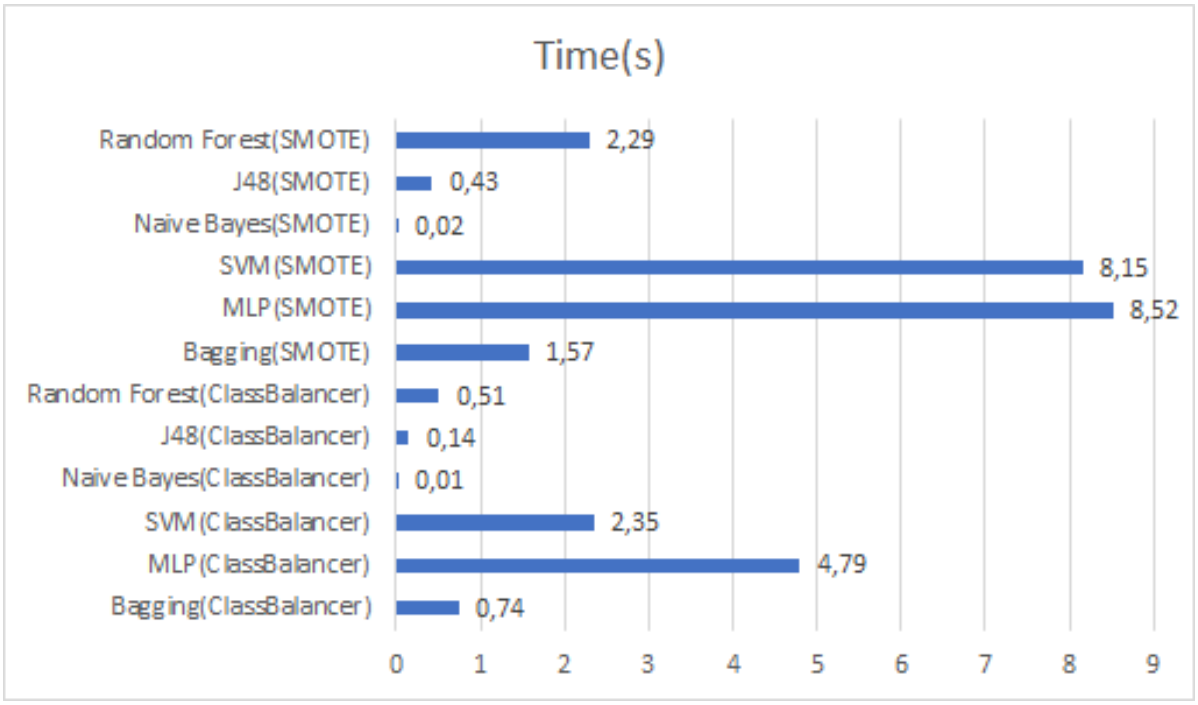

Figure A.11: Time in seconds for each classifier to be build with the class-level ”System Information Leak: Internal” dataset

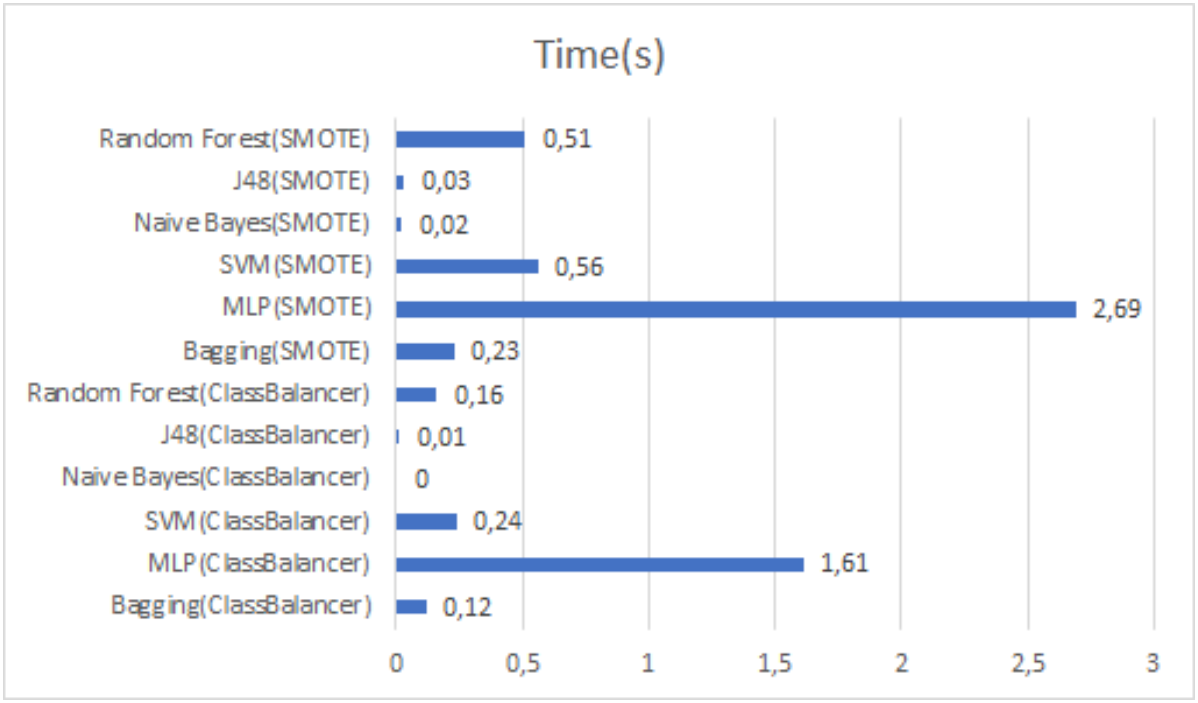

Figure A.12: Time in seconds for each classifier to be build with the class-level ”J2EE Bad Practices: Threads” dataset

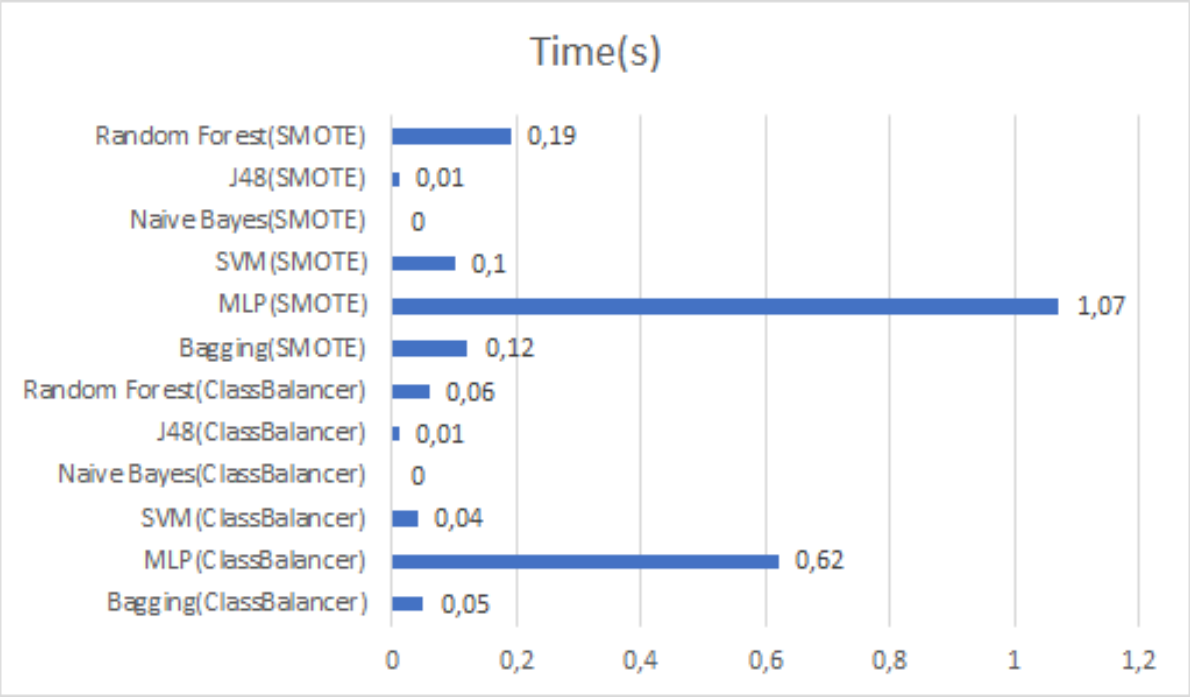

Figure A.13: Time in seconds for each classifier to be build with the class-level ”Log Forging” dataset

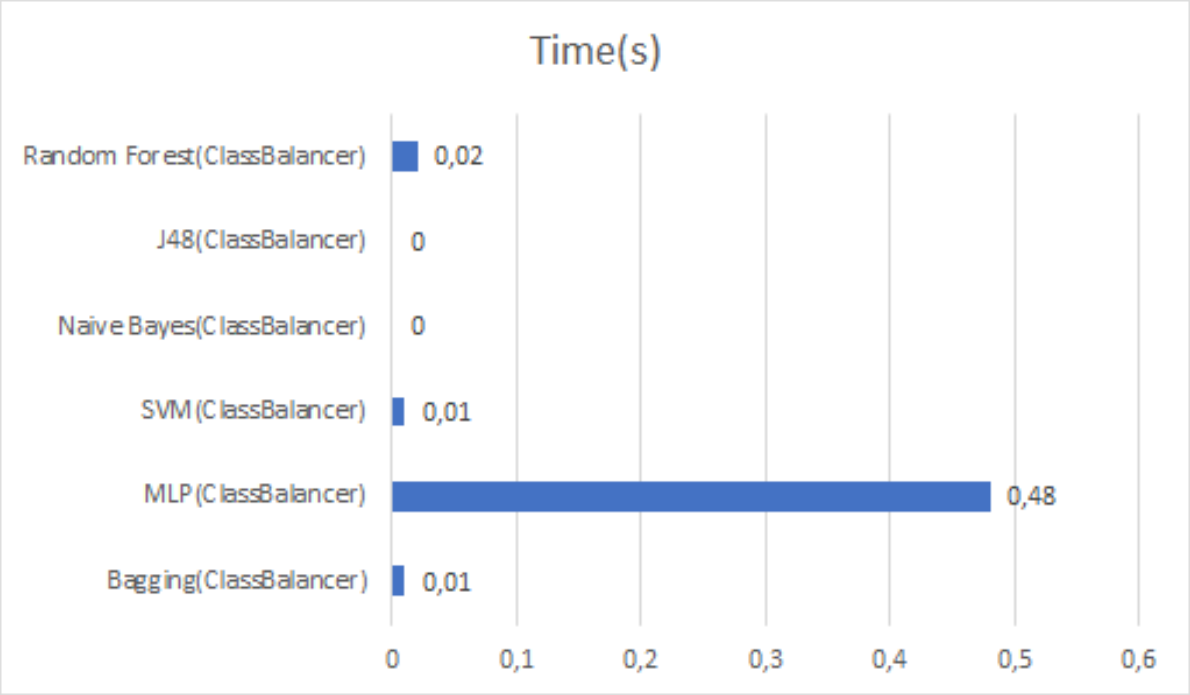

Figure A.14: Time in seconds for each classifier to be build with the class-level ”Password Management: Hardcoded Password” dataset

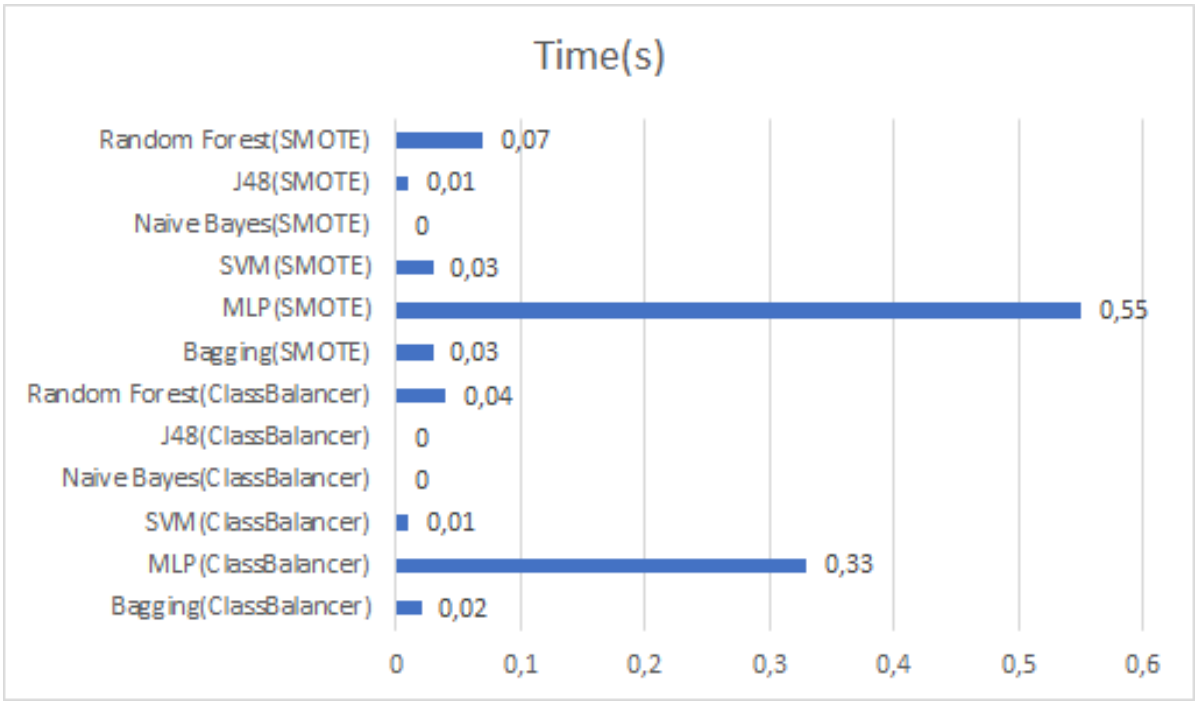

Figure A.15: Time in seconds for each classifier to be build with the class-level ”Path Manipulation” dataset

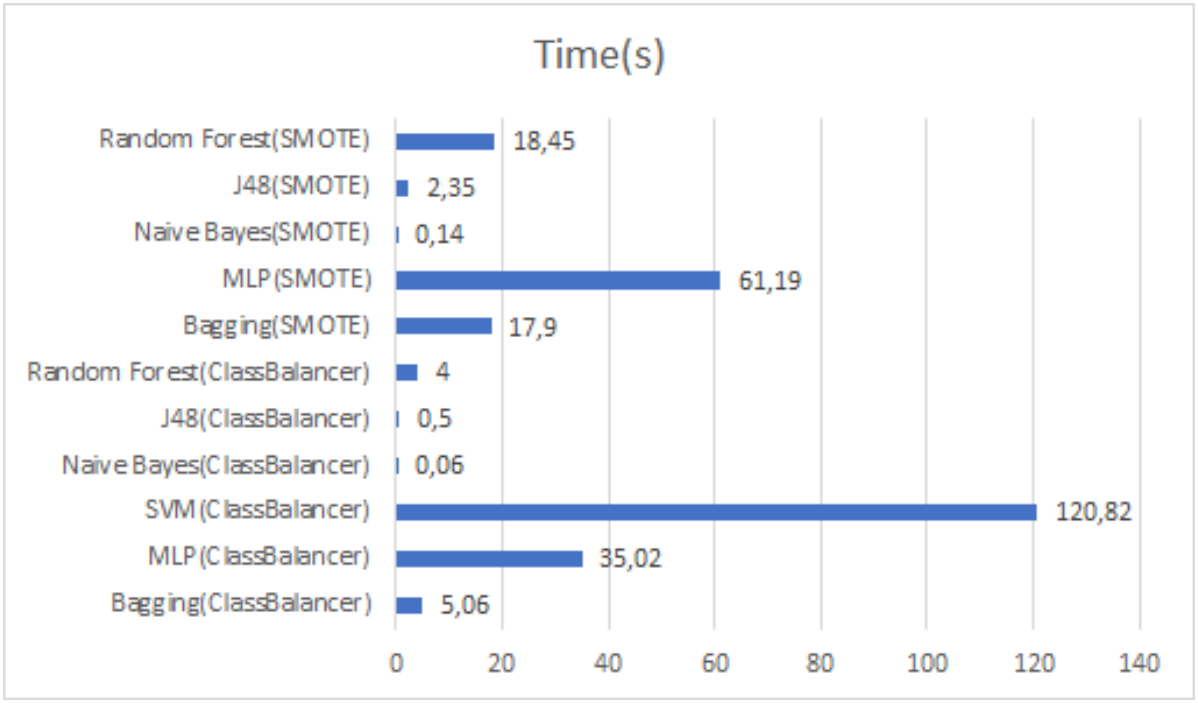

Figure A.16: Time in seconds for each classifier to be build with the method-level ”System Information Leak: Internal” dataset

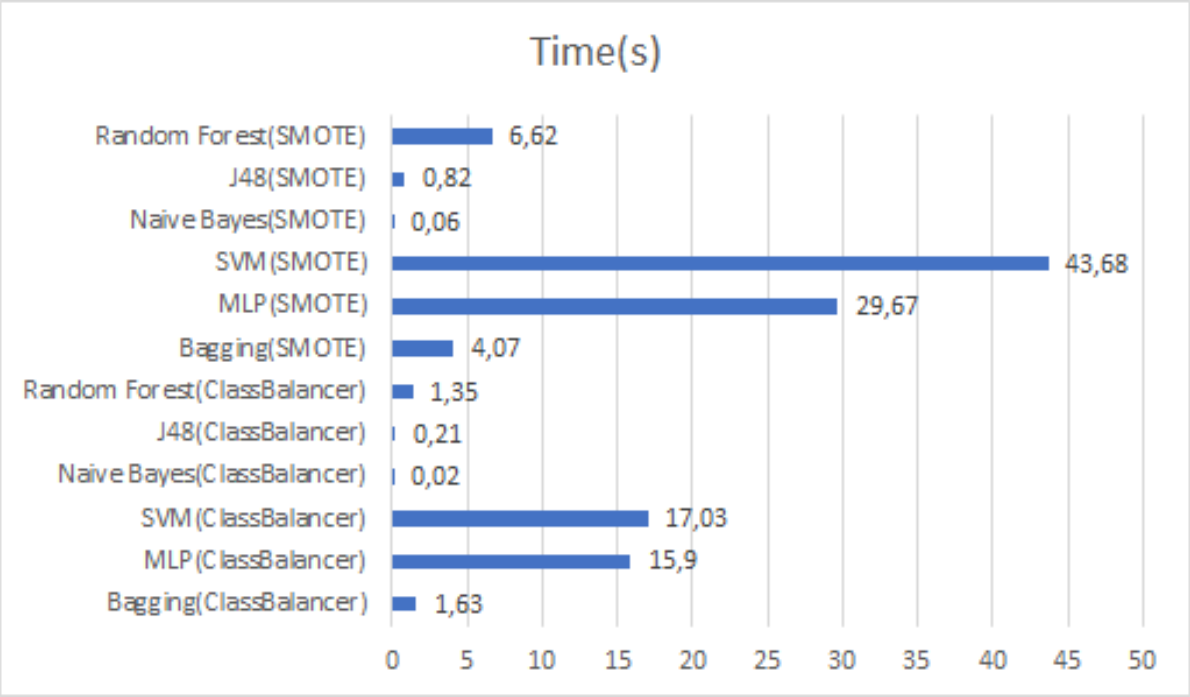

Figure A.17: Time in seconds for each classifier to be build with the method-level ”J2EE Bad Practices: Threads” dataset

Figure A.18: Time in seconds for each classifier to be build with the method-level ”Password Management: Hardcoded Password” dataset

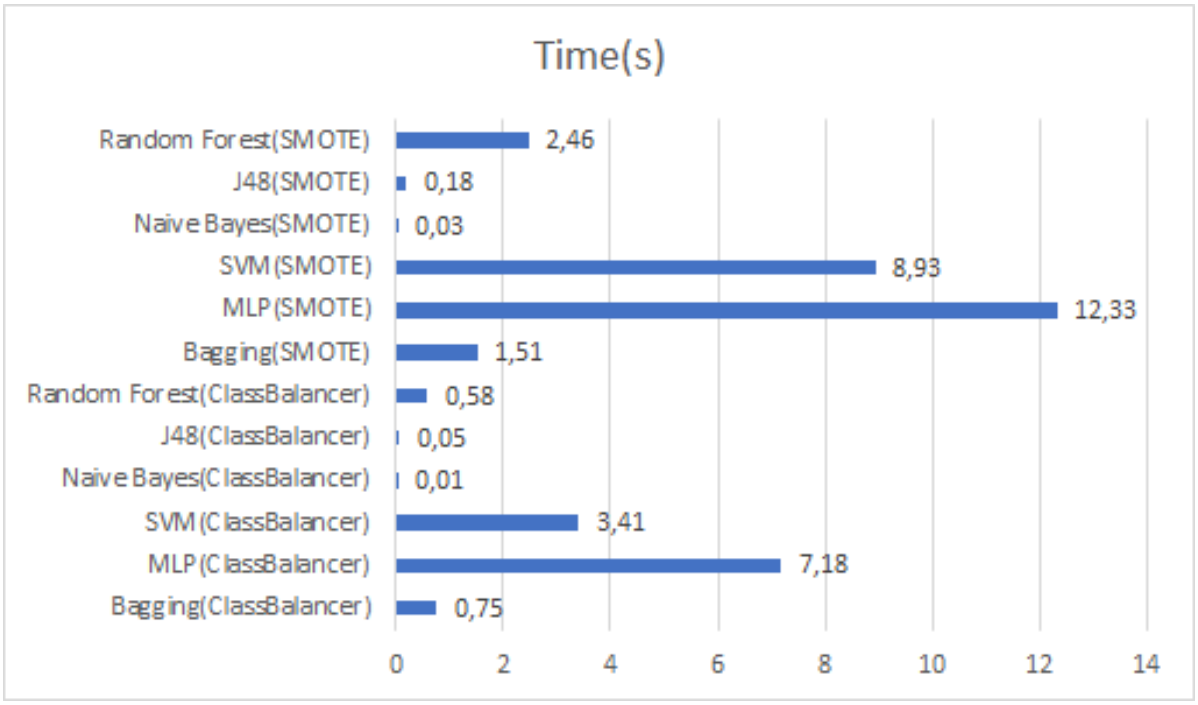

Figure A.19: Time in seconds for each classifier to be build with the method-level ”Log Forging” dataset

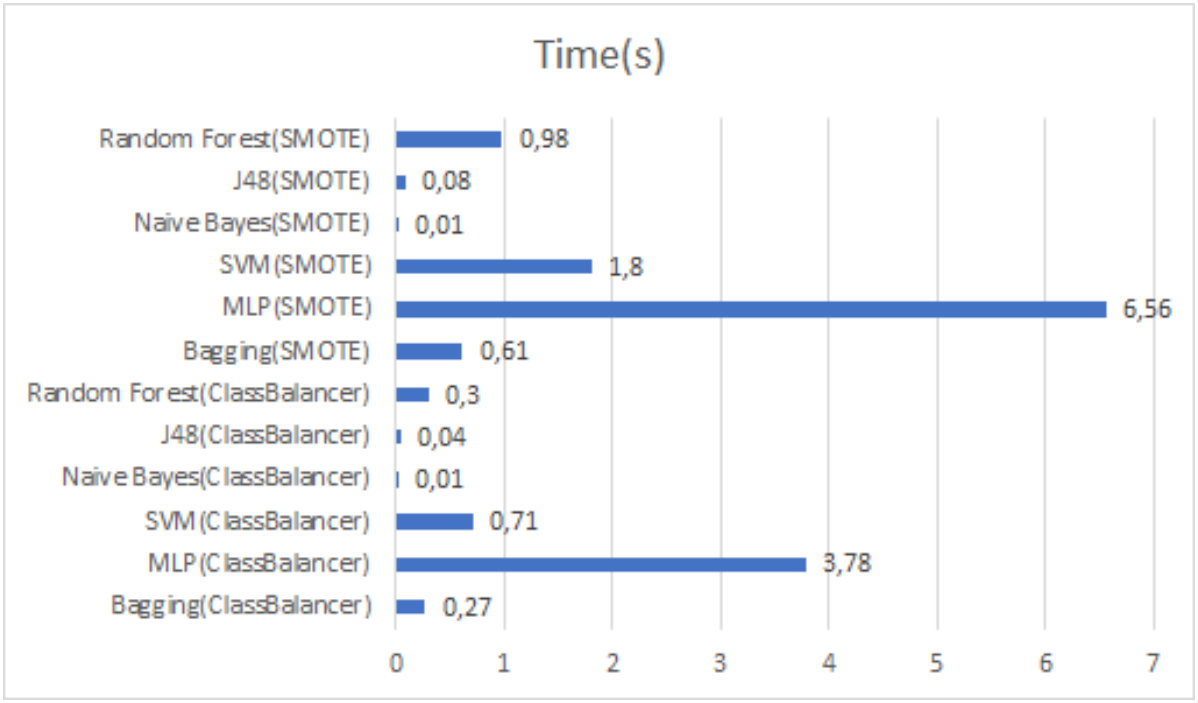

Figure A.20: Time in seconds for each classifier to be build with the method-level ”Path Manipulation” dataset

Classifier	Precision	Recall	F-Measure	ROC AUC	Time(s)
Random Forest(SMOTE)	0,147	0,234	0,181	0,797	2.29
J48(SMOTE)	0,152	0,250	0,189	0,582	0.43
Naive Bayes(SMOTE)	0,022	0,375	0,042	0,585	0.02
SVM(SMOTE)	0,033	0,250	0,058	0,573	8.15
MLP(SMOTE)	0,028	0,391	0,051	0,649	8.52
Bagging(SMOTE)	0,150	0,250	0,187	0,742	1.57
Random Forest(ClassBalancer)	0,013	0,125	0,024	0,727	0.51
J48(ClassBalancer)	0,027	0,281	0,048	0,568	0.14
Naive Bayes(ClassBalancer)	0,028	0,344	0,051	0,599	0.01
SVM(ClassBalancer)	0,027	0,375	0,051	0,594	2.35
MLP(ClassBalancer)	0,015	0,750	0,029	0,537	4.79
Bagging(ClassBalancer)	0,024	0,234	0,043	0,670	0.74

Table A.25: Performance of classifiers on class ”System Information Leak: Internal” dataset

Classifier	Precision	Recall	F-Measure	ROC AUC	Time(s)
Random Forest(SMOTE)	0,556	0,808	0,659	0,964	0.51
J48(SMOTE)	0,523	0,865	0,652	0,922	0.03
Naive Bayes(SMOTE)	0,388	0,317	0,349	0,644	0.02
SVM(SMOTE)	0,569	0,279	0,374	0,632	0.56
MLP(SMOTE)	0,505	0,885	0,643	0,944	2.69
Bagging(SMOTE)	0,537	0,904	0,674	0,967	0.23
Random Forest(ClassBalancer)	0,547	0,904	0,681	0,969	0.16
J48(ClassBalancer)	0,510	0,942	0,662	0,941	0.01
Naive Bayes(ClassBalancer)	0,588	0,288	0,387	0,596	0
SVM(ClassBalancer)	0,287	0,394	0,332	0,663	0.24
MLP(ClassBalancer)	0,261	0,904	0,405	0,907	1.61
Bagging(ClassBalancer)	0,497	0,913	0,644	0,962	0.12

Table A.26: Performance of classifiers on class ”J2EE Bad Practices: Threads” dataset

Classifier	Precision	Recall	F-Measure	ROC AUC	Time(s)
Random Forest(SMOTE)	0,397	0,511	0,447	0,916	0.19
J48(SMOTE)	0,300	0,467	0,365	0,700	0.01
Naive Bayes(SMOTE)	0,232	0,489	0,314	0,748	0
SVM(SMOTE)	0,239	0,489	0,321	0,684	0.1
MLP(SMOTE)	0,275	0,556	0,368	0,799	1.07
Bagging(SMOTE)	0,333	0,489	0,396	0,876	0.12
Random Forest(ClassBalancer)	0,400	0,400	0,400	0,901	0.06
J48(ClassBalancer)	0,317	0,422	0,362	0,676	0.01
Naive Bayes(ClassBalancer)	0,205	0,400	0,271	0,750	0
SVM(ClassBalancer)	0,206	0,578	0,304	0,703	0.04
MLP(ClassBalancer)	0,329	0,511	0,400	0,812	0.62
Bagging(ClassBalancer)	0,313	0,467	0,375	0,848	0.05

Table A.27: Performance of classifiers on class ”Log Forging” dataset

Classifier	Precision	Recall	F-Measure	ROC AUC	Time(s)
Random Forest(ClassBalancer)	0,000	0,000	0,000	0,723	0.02
J48(ClassBalancer)	0,000	0,000	0,000	0,495	0
Naive Bayes(ClassBalancer)	0,000	0,000	0,000	0,744	0
SVM(ClassBalancer)	0,047	1,000	0,089	0,957	0.01
MLP(ClassBalancer)	0,005	1,000	0,011	0,382	0.48
Bagging(ClassBalancer)	0,000	0,000	0,000	0,494	0.01

Table A.28: Performance of classifiers on class ”Password Management: Hardcoded Password” dataset

Classifier	Precision	Recall	F-Measure	ROC AUC	Time(s)
Random Forest(SMOTE)	0,595	0,694	0,641	0,877	0.07
J48(SMOTE)	0,575	0,639	0,605	0,814	0.01
Naive Bayes(SMOTE)	0,200	0,778	0,318	0,761	0
SVM(SMOTE)	0,209	0,778	0,329	0,712	0.03
MLP(SMOTE)	0,500	0,694	0,581	0,805	0.55
Bagging(SMOTE)	0,511	0,639	0,568	0,850	0.03
Random Forest(ClassBalancer)	0,585	0,667	0,623	0,864	0.04
J48(ClassBalancer)	0,468	0,611	0,530	0,767	0
Naive Bayes(ClassBalancer)	0,193	0,778	0,309	0,719	0
SVM(ClassBalancer)	0,194	0,833	0,314	0,708	0.01
MLP(ClassBalancer)	0,333	0,611	0,431	0,790	0.33
Bagging(ClassBalancer)	0,444	0,667	0,533	0,849	0.02

Table A.29: Performance of classifiers on class ”Path Manipulation” dataset

Classifier	Precision	Recall	F-Measure	ROC AUC	Time(s)
Random Forest(SMOTE)	0,089	0,151	0,112	0,842	18.45
J48(SMOTE)	0,056	0,192	0,087	0,760	2.35
Naive Bayes(SMOTE)	0,007	0,562	0,014	0,784	0.14
SVM(SMOTE)	-	-	-	-	-
MLP(SMOTE)	0,012	0,630	0,024	0,827	61.19
Bagging(SMOTE)	0,055	0,151	0,081	0,832	17.9
Random Forest(ClassBalancer)	0,030	0,233	0,053	0,847	4
J48(ClassBalancer)	0,035	0,438	0,065	0,710	0.5
Naive Bayes(ClassBalancer)	0,007	0,370	0,013	0,767	0.06
SVM(ClassBalancer)	0,009	0,781	0,018	0,800	120.82
MLP(ClassBalancer)	0,002	1,000	0,004	0,517	35.02
Bagging(ClassBalancer)	0,033	0,356	0,061	0,770	5.06

Table A.30: Performance of classifiers on method ”System Information Leak: Internal” dataset

Classifier	Precision	Recall	F-Measure	ROC AUC	Time(s)
Random Forest(SMOTE)	0,517	0,643	0,574	0,972	6.62
J48(SMOTE)	0,357	0,643	0,460	0,929	0.82
Naive Bayes(SMOTE)	0,038	0,417	0,070	0,799	0.06
SVM(SMOTE)	0,050	0,591	0,092	0,753	43.68
MLP(SMOTE)	0,112	0,826	0,198	0,913	29.67
Bagging(SMOTE)	0,388	0,696	0,498	0,967	4.07
Random Forest(ClassBalancer)	0,500	0,791	0,613	0,968	1.35
J48(ClassBalancer)	0,284	0,809	0,420	0,899	0.21
Naive Bayes(ClassBalancer)	0,043	0,417	0,078	0,799	0.02
SVM(ClassBalancer)	0,042	0,635	0,080	0,763	17.03
MLP(ClassBalancer)	0,008	1,000	0,015	0,545	15.9
Bagging(ClassBalancer)	0,380	0,800	0,515	0,947	1.63

Table A.31: Performance of classifiers on method ”J2EE Bad Practices: Threads” dataset

Classifier	Precision	Recall	F-Measure	ROC AUC	Time(s)
Random Forest(ClassBalancer)	0,000	0,000	0,000	0,496	0.16
J48(ClassBalancer)	0,000	0,000	0,000	0,496	0.01
Naive Bayes(ClassBalancer)	0,000	0,000	0,000	0,325	0.01
SVM(ClassBalancer)	-	-	-	-	-
MLP(ClassBalancer)	0,000	0,000	0,000	0,088	7
Bagging(ClassBalancer)	0,000	0,000	0,000	0,496	0.12

Table A.32: Performance of classifiers on method ”Password Management: Hardcoded Password” dataset

Classifier	Precision	Recall	F-Measure	ROC AUC	Time(s)
Random Forest(SMOTE)	0,290	0,265	0,277	0,923	2.46
J48(SMOTE)	0,205	0,382	0,267	0,793	0.18
Naive Bayes(SMOTE)	0,034	0,485	0,063	0,751	0.03
SVM(SMOTE)	0,034	0,603	0,065	0,715	8.93
MLP(SMOTE)	0,065	0,750	0,120	0,849	12.33
Bagging(SMOTE)	0,215	0,382	0,275	0,904	1.51
Random Forest(ClassBalancer)	0,333	0,250	0,286	0,912	0.58
J48(ClassBalancer)	0,150	0,397	0,218	0,687	0.05
Naive Bayes(ClassBalancer)	0,037	0,456	0,068	0,797	0.01
SVM(ClassBalancer)	0,027	0,706	0,051	0,721	3.41
MLP(ClassBalancer)	0,010	1,000	0,020	0,621	7.18
Bagging(ClassBalancer)	0,228	0,309	0,262	0,831	0.75

Table A.33: Performance of classifiers on method ”Log Forging” dataset

Classifier	Precision	Recall	F-Measure	ROC AUC	Time(s)
Random Forest(SMOTE)	0,500	0,435	0,465	0,827	0.98
J48(SMOTE)	0,368	0,457	0,408	0,791	0.08
Naive Bayes(SMOTE)	0,093	0,652	0,163	0,847	0.01
SVM(SMOTE)	0,067	0,609	0,121	0,750	1.8
MLP(SMOTE)	0,114	0,543	0,188	0,768	6.56
Bagging(SMOTE)	0,440	0,478	0,458	0,869	0.61
Random Forest(ClassBalancer)	0,149	0,435	0,222	0,815	0.3
J48(ClassBalancer)	0,132	0,522	0,211	0,748	0.04
Naive Bayes(ClassBalancer)	0,089	0,652	0,157	0,849	0.01
SVM(ClassBalancer)	0,060	0,652	0,110	0,761	0.71
MLP(ClassBalancer)	0,015	0,870	0,030	0,551	3.78
Bagging(ClassBalancer)	0,128	0,478	0,202	0,799	0.27

Table A.34: Performance of classifiers on method ”Path Manipulation” dataset

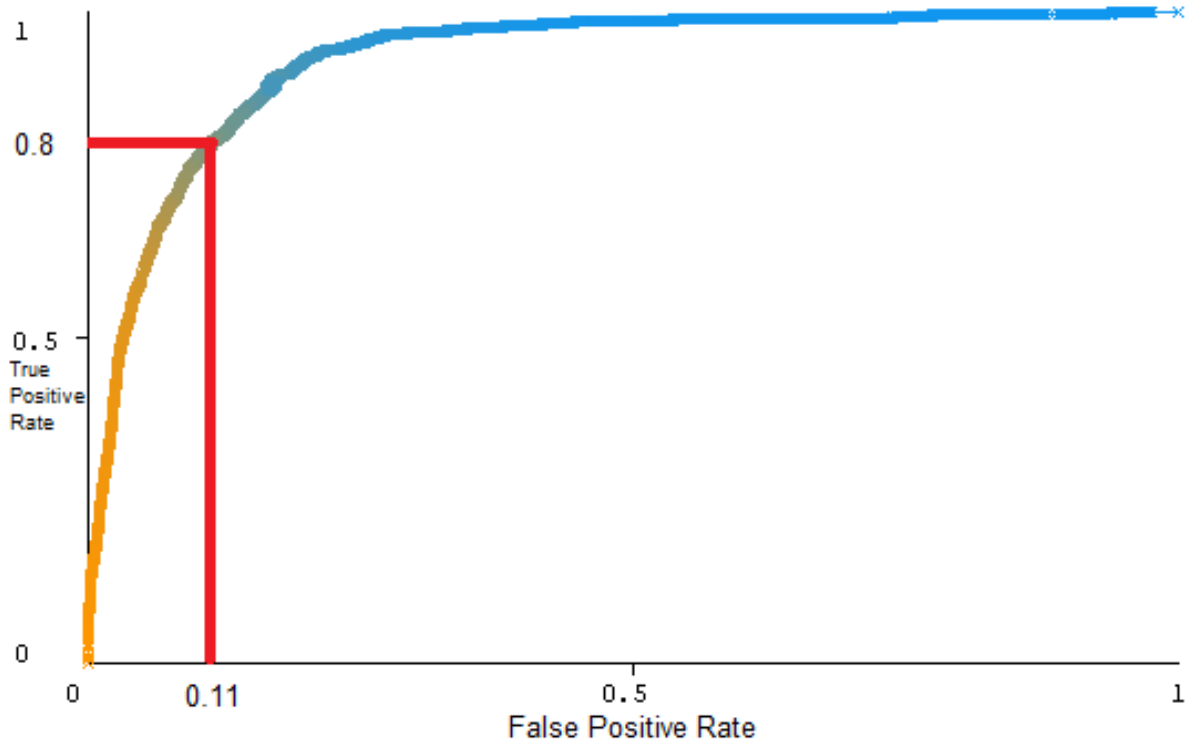

Figure A.21: ROC curve Bagging with ClassBalancer for class-level dataset

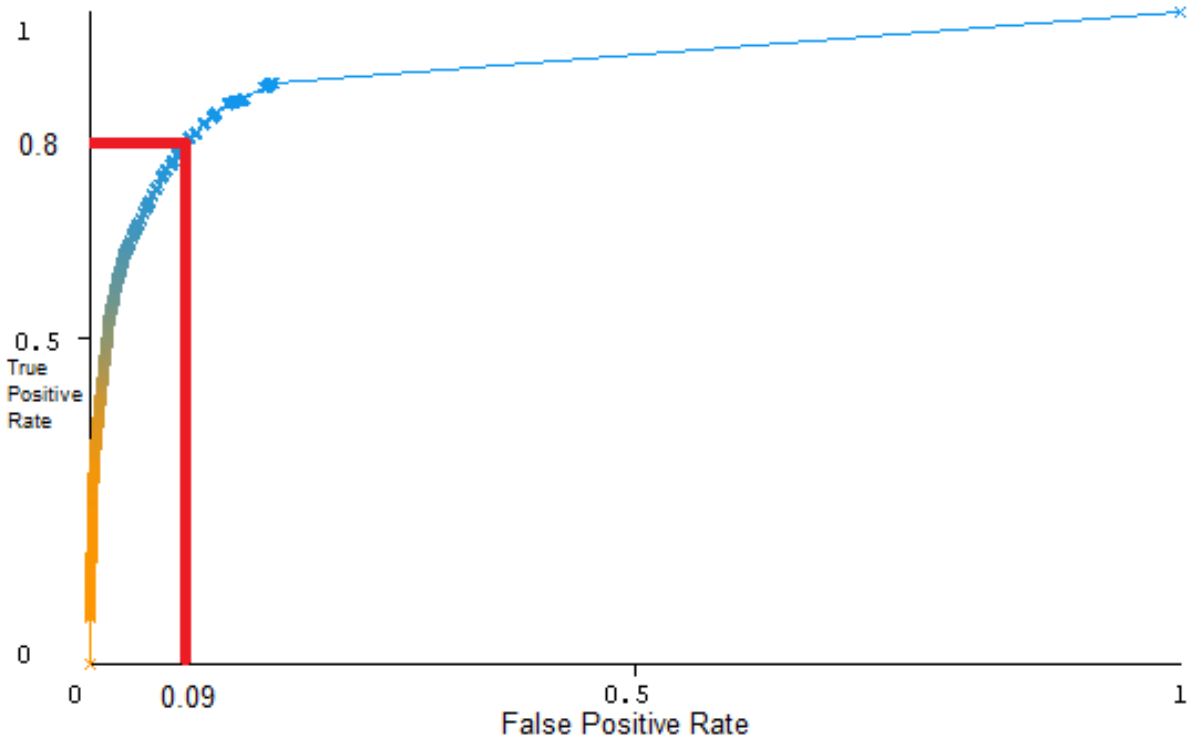

Figure A.22: ROC curve Random Forest with SMOTE for method-level dataset

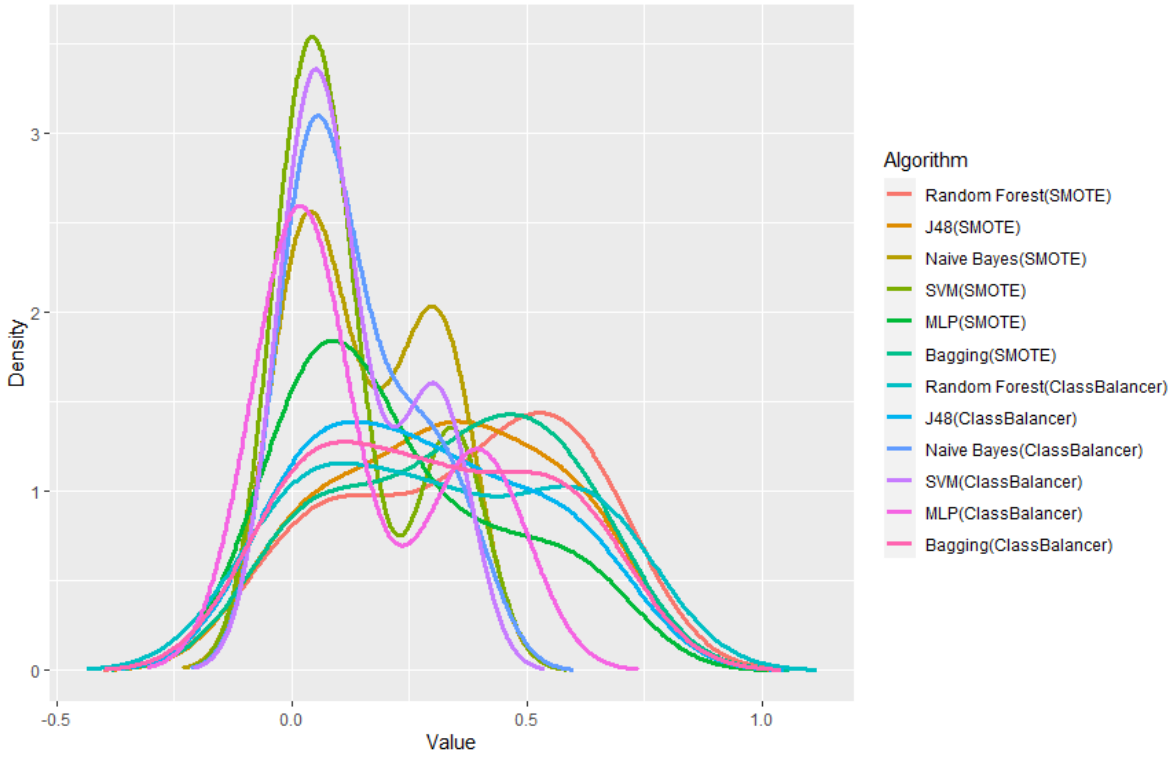

Figure A.23: Density of the classifiers

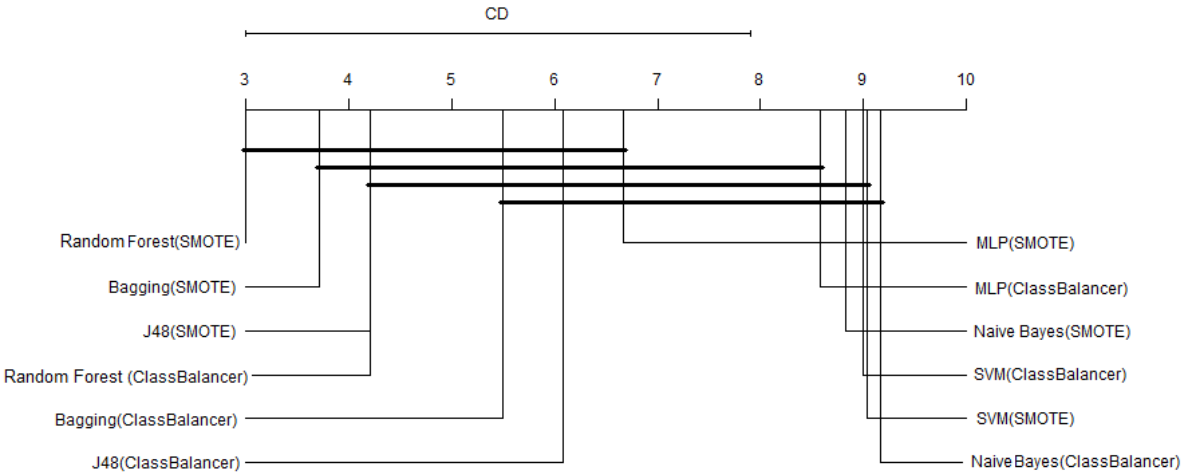

Figure A.24: CD plot of comparison between the classifiers

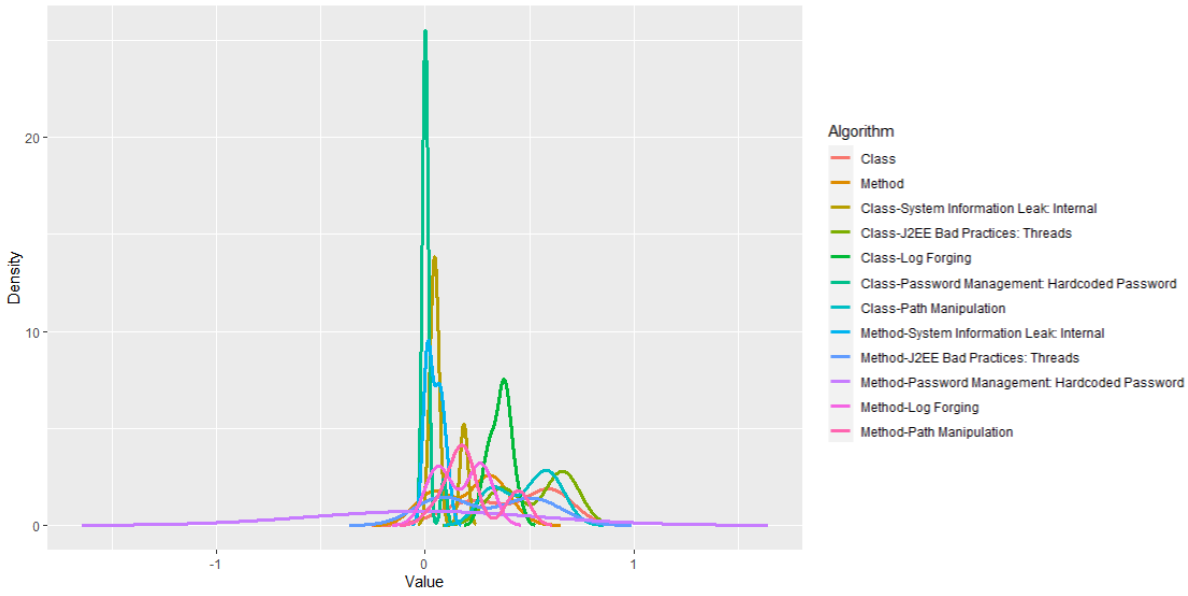

Figure A.25: Density of the datasets

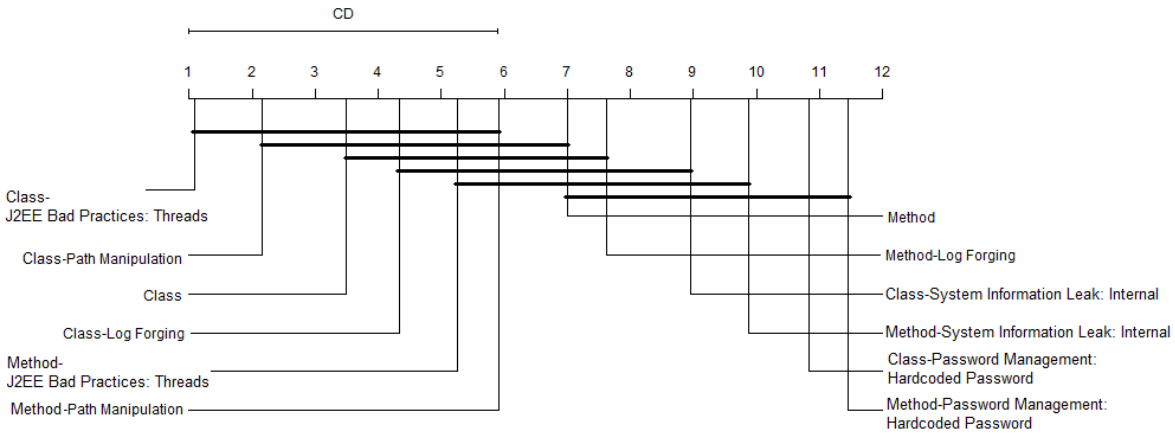

Figure A.26: CD plot of comparison between the datasets

Bibliography

[1] Saiqa Aleem, Luiz Fernando Capretz, and Faheem Ahmed. Benchmarking machine learning technologies for software defect detection. *arXiv preprint arXiv:1506.07563*, 2015.

[2] Areej Algaith. *Assessing the security benefits of defence in depth*. PhD thesis, City, University of London, 2019.

[3] Henrique Alves, Baldoino Fonseca, and Nuno Antunes. Experimenting machine learning techniques to predict vulnerabilities. In *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*, pages 151–156. IEEE, 2016.

[4] Richard Amankwah, Patrick Kwaku Kudjo, and Samuel Yeboah Antwi. Evaluation of software vulnerability detection methods and tools: A review. *International Journal of Computers and Applications*, 169(8):22–27, 2017.

[5] Maurício Aniche. *Java code metrics calculator (CK)*, 2015. Available in https://github.com/mauricioaniche/ck/.

[6] Nuno Antunes and Marco Vieira. Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services. In *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 301–306. IEEE, 2009.

[7] Hasty Atashzar, Atefeh Torkaman, Marjan Bahrololum, and Mohammad H Tadayon. A survey on web application vulnerabilities and countermeasures. In *2011 6th International Conference on Computer Sciences and Convergence Information Technology (ICCIT)*, pages 647–652. IEEE, 2011.

[8] Andrew Austin and Laurie Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 97–106. IEEE, 2011.

[9] Nor Fatimah Awang and Azizah Abd Manaf. Detecting vulnerabilities in web applications using automated black box and manual penetration testing. In *International Conference on Security of Information and Communication Networks*, pages 230–239. Springer, 2013.

[10] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated black-box web application vulnerability testing. In *2010 IEEE Symposium on Security and Privacy*, pages 332–345. IEEE, 2010.

[11] Amiangshu Bosu, Jeffrey C Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. Identifying the characteristics of vulnerable code changes: An empirical study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 257–268, 2014.

[12] Cagatay Catal and Banu Diri. A systematic review of software fault prediction studies. *Expert systems with applications*, 36(4):7346–7354, 2009.

[13] Girish Chandrashekar and Ferat Sahin. A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1):16–28, 2014.

[14] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[15] Brian Chess and Jacob West. *Secure programming with static analysis*. Pearson Education, 2007.

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

[17] Anuradha Chug and Shafali Dhall. Software defect prediction using supervised learning algorithm and unsupervised learning algorithm. 2013.

[18] Mark Curphey and Rudolph Arawo. Web application security assessment tools. *IEEE Security & Privacy*, 4(4):32–41, 2006.

[19] Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7(1):1–30, 2006. URL `http://jmlr.org/papers/v7/demsar06a.html`.

[20] Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine learning research*, 7(Jan):1–30, 2006.

[21] Maureen Doyle and James Walden. An empirical study of the evolution of php web application security. In *2011 Third International Workshop on Security Measurements and Metrics*, pages 11–20. IEEE, 2011.

[22] Shaza M Abd Elrahman and Ajith Abraham. A review of class imbalance problem. *Journal of Network and Innovative Computing*, 1(2013):332–340, 2013.

[23] Alberto Fernández, Sara del Río, Nitesh V Chawla, and Francisco Herrera. An insight into imbalanced big data classification: outcomes and challenges. *Complex & Intelligent Systems*, 3(2):105–120, 2017.

[24] Eibe Frank and A Mark. Hall, and ian h. witten (2016). the weka workbench. online appendix for" data mining: Practical machine learning tools and techniques, 2016.

[25] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[26] Vicente García, José Salvador Sánchez, and Ramón Alberto Mollineda. On the effectiveness of preprocessing methods when dealing with different levels of class imbalance. *Knowledge-Based Systems*, 25(1):13–21, 2012.

[27] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50(4):56, 2017.

[28] Qiong Gu, Li Zhu, and Zhihua Cai. Evaluation measures of the classification performance of imbalanced data sets. In *International symposium on intelligence computation and applications*, pages 461–471. Springer, 2009.

[29] Guo Haixiang, Li Yijing, Jennifer Shang, Gu Mingyun, Huang Yuanyue, and Gong Bing. Learning from class-imbalanced data: Review of methods and applications. *Expert Systems with Applications*, 73:220–239, 2017.

[30] Jacob A Harer, Louis Y Kim, Rebecca L Russell, Onur Ozdemir, Leonard R Kosta, Akshay Rangamani, Lei H Hamilton, Gabriel I Centeno, Jonathan R Key, Paul M Ellingwood, et al. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*, 2018.

[31] Jon Heffley and Pascal Meunier. Can source code auditing software identify common vulnerabilities and be used to evaluate software security? In *37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the*, pages 10–pp. IEEE, 2004.

[32] Martin Johns and Moritz Jodeit. Scanstud: a methodology for systematic, fine-grained evaluation of static analysis tools. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 523–530. IEEE, 2011.

[33] Maximilian Junker, Ralf Huuck, Ansgar Fehnker, and Alexander Knapp. Smt-based false positive elimination in static program analysis. In *International Conference on Formal Engineering Methods*, pages 316–331. Springer, 2012.

[34] Samina Khalid, Tehmina Khalil, and Shamila Nasreen. A survey of feature selection and feature extraction techniques in machine learning. In *2014 Science and Information Conference*, pages 372–378. IEEE, 2014.

[35] Pieter Kubben, Michel Dumontier, and Andre Dekker. *Fundamentals of Clinical Data Science*. Springer, 2019.

[36] Kaspersky Lab. Carbanak apt: The great bank robbery. *Securelist*, 2015.

[37] Joffrey L Leevy, Taghi M Khoshgoftaar, Richard A Bauder, and Naeem Seliya. A survey on addressing high-class imbalance in big data. *Journal of Big Data*, 5(1):42, 2018.

[38] Paige Leskin. The 21 scariest data breaches of 2018, Dec 2018. URL `https://www.businessinsider.nl/data-hacks-breaches-biggest-of-2018-2018-12/`.

[39] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, 2018.

[40] 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.

[41] KW Man. Predicting software vulnerabilities with unsupervised learning techniques. 2020.

[42] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[43] Steve Mansfield-Devine. The ashley madison affair. *Network Security*, 2015(9):8–16, 2015.

[44] Trust Tshepo Mapoka, Keneilwe Zuva, and Tranos Zuva. Hack the bank and best practices for secure bank.

[45] Ibéria Medeiros, Nuno F Neves, and Miguel Correia. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *Proceedings of the 23rd international conference on World wide web*, pages 63–74. ACM, 2014.

[46] Andrew Meneely and Laurie Williams. Strengthening the empirical analysis of the relationship between linus' law and software security. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2010.

[47] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodriguez Tejeda, Matthew Mokary, and Brian Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 65–74. IEEE, 2013.

[48] Savita Mohurle and Manisha Patil. A brief study of wannacry threat: Ransomware attack 2017. *International Journal of Advanced Research in Computer Science*, 8(5), 2017.

[49] Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie Williams. Challenges with applying vulnerability prediction models. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, pages 1–9, 2015.

[50] Sara Moshtari, Ashkan Sami, and Mahdi Azimi. Using complexity metrics to improve software security. *Computer Fraud & Security*, 2013(5):8–17, 2013.

[51] Iqbal Muhammad and Zhu Yan. Supervised machine learning approaches: A survey. *ICTACT Journal on Soft Computing*, 5(3), 2015.

[52] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[53] Paulo Nunes, Ibéria Medeiros, José Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. On combining diverse static analysis tools for web security: An empirical study. In *2017 13th European dependable computing conference (EDCC)*, pages 121–128. IEEE, 2017.

[54] Paulo Nunes, Ibéria Medeiros, José C Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. Benchmarking static analysis tools for web security. *IEEE Transactions on Reliability*, 67(3):1159–1175, 2018.

[55] Paulo Nunes, Ibéria Medeiros, José Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios. *Computing*, 101(2):161–185, 2019.

[56] National Institute of Standards & Technology. The economic impacts of inadequate infrastructure for software testing, may 2002. Accessed: 27th July 2019. Available at: `https://www.nist.gov/sites/default/files/documents/director/planning/report02-3.pdf`.

[57] Joonyoung Park, Inho Lim, and Sukyoung Ryu. Battles with false positives in static analysis of javascript web applications in the wild. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 61–70. IEEE, 2016.

[58] Refaeilzadeh Payam, Tang Lei, and Liu Huan. Cross-validation. *Encyclopedia of database systems*, pages 532–538, 2009.

[59] Hanchuan Peng, Fuhui Long, and Chris Ding. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on pattern analysis and machine intelligence*, 27(8):1226–1238, 2005.

[60] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 426–437, 2015.

[61] Liu Ping, Su Jin, and Yang Xinfeng. Research on software security vulnerability detection technology. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, volume 3, pages 1873–1876. IEEE, 2011.

[62] Thorsten Pohlert. The pairwise multiple comparison of mean ranks package (pmcmr). *R package*, page 27, 2016. `https://cran.r-project.org/web/packages/PMCMR/vignettes/PMCMR.pdf`.

[63] MC Prasad, Lilly Florence, and Arti Arya. A study on software metrics based software defect prediction using data mining and machine learning techniques. *International Journal of Database Theory and Application*, 8(3):179–190, 2015.

[64] Jose C Principe, Neil R Euliano, and W Curt Lefebvre. *Neural and adaptive systems: fundamentals through simulations*, volume 672. Wiley New York, 2000.

[65] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[66] RH Riffenburgh. Chapter 11—tests on ranked data. *Statistics in Medicine, 3rd ed.; Elsevier Inc.: San Diego, CA, USA*, pages 221–248, 2012.

[67] Daniel Rodriguez, Israel Herraiz, Rachel Harrison, Javier Dolado, and José C Riquelme. Preliminary comparison of techniques for dealing with imbalance in software defect prediction. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–10, 2014.

[68] Linda Rosenberg. Applying and interpreting object oriented metrics. In *Software Technology Conference, Utah, April 1998*, 1998.

[69] Joseph Ruthruff, John Penix, J Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 341–350. IEEE, 2008.

[70] Gary J Saavedra, Kathryn N Rodhouse, Daniel M Dunlavy, and Philip W Kegelmeyer. A review of machine learning applications in fuzzing. *arXiv preprint arXiv:1906.11133*, 2019.

[71] Riccardo Scandariato, James Walden, and Wouter Joosen. Static analysis versus penetration testing: A controlled experiment. In *2013 IEEE 24th international symposium on software reliability engineering (ISSRE)*, pages 451–460. IEEE, 2013.

[72] Hossain Shahriar and Mohammad Zulkernine. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Computing Surveys (CSUR)*, 44(3):11, 2012.

[73] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[74] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 315–317, 2008.

[75] Yonghee Shin and Laurie Williams. An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In *Proceedings of the 7th International Workshop on Software Engineering for Secure Systems*, pages 1–7, 2011.

[76] Yonghee Shin and Laurie Williams. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18(1):25–59, 2013.

[77] 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.

[78] Yanmin Sun, Andrew KC Wong, and Mohamed S Kamel. Classification of imbalanced data: A review. *International journal of pattern recognition and artificial intelligence*, 23 (04):687–719, 2009.

[79] Yeresime Suresh, Jayadeep Pati, and Santanu Ku Rath. Effectiveness of software metrics for object-oriented system. *Procedia technology*, 6:420–427, 2012.

[80] Jay-Evan J Tevis and John A Hamilton. Methods for the prevention, detection and removal of software security vulnerabilities. In *Proceedings of the 42nd annual Southeast regional conference*, pages 197–202. ACM, 2004.

[81] Archit Verma. Evaluation of classification algorithms with solutions to class imbalance problem on bank marketing dataset using weka. *International Research Journal of Engineering and Technology*, pages 54–60, 2019.

[82] James Walden, Jeff Stuckman, and Riccardo Scandariato. Predicting vulnerable components: Software metrics vs text mining. In *2014 IEEE 25th international symposium on software reliability engineering*, pages 23–33. IEEE, 2014.

[83] Wang Wei. Survey of software vulnerability discovery technology. In *2017 7th International Conference on Social Network, Communication and Education (SNCE 2017)*. Atlantis Press, 2017.

[84] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *The 10th Network & Distributed System Security Symposium 2003 (NDSS), San Diego, California, USA*, page 149. Internet Society, 2003.

[85] Guanhua Yan, Junchen Lu, Zhan Shu, and Yunus Kucuk. Exploitmeter: Combining fuzzing with machine learning for automated evaluation of software exploitability. In *2017 IEEE Symposium on Privacy-Aware Computing (PAC)*, pages 164–175. IEEE, 2017.

[86] Jinqiu Yang, Lin Tan, John Peyton, and Kristofer A Duer. Towards better utilizing static application security testing. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, pages 51–60. IEEE Press, 2019.

[87] Awad Younis, Yashwant Malaiya, Charles Anderson, and Indrajit Ray. To fear or not to fear that is the question: Code characteristics of a vulnerable functionwith an existing exploit. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 97–104, 2016.

[88] Chulmin Yun and Jihoon Yang. Experimental comparison of feature subset selection methods. In *Seventh IEEE International Conference on Data Mining Workshops (ICDMW 2007)*, pages 367–372. IEEE, 2007.

[89] 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.