TÉCNICO LISBOA

Automatic Bug Prioritization of SmartBugs Reports using Machine Learning

João Tiago Sousa Dinis

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. João Fernando Peixoto Ferreira
Prof. Rui Filipe Lima Maranhão de Abreu

Examination Committee

Chairperson: Profª. Valentina Nisi
Supervisor: Prof. João Fernando Peixoto Ferreira
Member of the Committee: Prof. João Carlos Serrenho Dias Pereira

October 2022

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Acknowledgments

I would like to thank my parents for their friendship, encouragement and caring over all these years, for always being there for me through thick and thin and without whom this project would not be possible. I would also like to thank my grandparents, aunts, uncles and cousins for their understanding and support throughout all these years.

If not for their nurture, I would have not started my journey on this university, nor would I have had a chance to write this thesis as I am now. A particularly large thanks to my grandfather, who used to tell me stories about about his job in communications before he passed away, to my uncle who, as an alumni of the same university, gave me the push I needed to start this course and lastly to my father who provided me with a place to learn.

I must also acknowledge all the professors that I had the pleasure to learn from throughout my time here. I thank you for all the skills, mindsets, and lessons that you have shared with me in these short 5 years we spent together.

I would also like to acknowledge my dissertation supervisors Prof. João Ferreira and Prof. Rui Maranhão for their insight, support and sharing of knowledge that has made this thesis possible.

Last but not least, to all my friends and colleagues that helped me grow as a person and were always there for me during the good and bad times in my life. Thank you.

To each and every one of you – Thank you.

Abstract

False positives are an inherent part of bug analysis tools, but developers lose faith in tools that present false positives too frequently. SmartBugs in particular, a framework designed to analyse Ethereum smart contracts, provides reports from 11 different analysis tools, and consequently reports many false positives. We extend SmartBugs with a bug prioritization algorithm that leverages machine learning and that sorts bug reports, ranking true positives above false positives, thus providing a better experience for developers. Using SARIF only features and ensemble regressor models we save up to 80% of a developer's time, an almost 5x gain in time efficiency in the process of analysing bug reports.

Keywords

SmartBugs, Bug Prioritization, Machine Learning

Resumo

Os falsos positivos são uma parte inerente das ferramentas de análise de bugs, no entanto programadores perdem a fé em ferramentas que apresentam falsos positivos com demasiada frequência. SmartBugs em particular, é um projeto que analisa contratos inteligentes em Ethereum, fornecendo relatórios de 11 ferramentas de análise diferentes e, consequentemente, relatando muitos falsos positivos. Estendemos o SmartBugs com um algoritmo de priorização de bugs que classifica relatórios de bugs usando aprendizagem de máquina, classificando os verdadeiros positivos acima dos falsos positivos, proporcionando assim uma melhor experiência para os programadores. Usando apenas característica SARIF e modelos compostos regressivos, conseguimos salvar até 80% do tempo de um programador, aumentando a eficiência de analisar bugs em quase 5x.

Palavras Chave

SmartBugs, Prioritização de Bugs, Aprendizagem de Máquina

Contents

1 Introduction **1**

1.1 Work Objectives . . . 4

1.2 Contributions . . . 5

1.3 Organization of the Document . . . 6

2 Background **7**

2.1 The Problem of False Positives . . . 9

2.2 Bug Prioritization . . . 9

2.3 SmartBugs . . . 10

2.3.1 Empirical Study . . . 12

3 Related Work **13**

3.1 An Empirical Assessment of Machine Learning approaches for Triaging Reports of a Java Static Analysis Tool . . . 15

3.2 Prioritizing Runtime Verification Violations . . . 17

4 Data Processing **23**

4.1 Data Preparation Pipeline . . . 25

4.2 Datasets . . . 25

4.3 From SARIF to bug reports CSV . . . 27

4.4 From metadata to solutions CSV . . . 28

4.4.1 SmartBugs $sb^{curated}$ dataset . . . 29

4.4.2 SolidiFI . . . 29

4.4.3 HuangGai . . . 30

4.4.4 Solutions CSV file organization and standards . . . 30

4.5 Final CSV . . . 33

5 Machine Learning Model **35**

5.1 Features . . . 37

5.2 Data preparation . . . 37

5.2.1 Data division . . . 38

5.2.2 Data balancing . . . 38
5.3 Machine learning models used . . . 39
5.3.1 Classification models . . . 40
5.3.2 Regression models . . . 40
5.3.3 Ensemble models . . . 40
6 Evaluation **43**
6.1 Evaluation metrics . . . 45
6.1.1 Classification Metrics . . . 45
6.1.2 APBD . . . 46
6.1.3 Prioritization graph . . . 47
6.2 Classification models . . . 48
6.3 Regression models . . . 51
6.4 Ensemble models . . . 53
7 Conclusion **57**
7.1 Contributions . . . 59
7.2 Answer to research questions . . . 59
7.3 Threats to validity . . . 60
7.4 Future work . . . 61
Bibliography **61**
A Mapping Tables **67**
A.1 Mapping between SmartBugs tools' vulnerabilities and DASP10 vulnerability types . . . 67
B Full results table **71**
B.1 Full results table, including training and testing metrics for all tested models. . . . 71

List of Figures

2.1 SmartBugs Architecture [1] 11
2.2 SmartBugs results in the $sb^{curated}$ dataset [2] 12
2.3 SmartBugs results in the sb^{wild} dataset [2] 12

3.1 Results from Koc et al. experiments [3] 18
3.2 RVPrio prioritization performance [4] 21

6.1 Example of a prioritization graph 48
6.2 Decision tree classifier model's prioritization graph 49
6.3 KNeighbors classifier model's prioritization graph 50
6.4 Decision tree Regressor model's prioritization graph 52
6.5 KNeighbors Regressor model's prioritization graph 52
6.6 Random Forest Classifier model's prioritization graph 55
6.7 Random Forest Regressor model's prioritization graph 55

List of Tables

3.1 Performance of different classifiers in RVPrio [4] 20
3.2 RVPrio percentage of true bugs revealed at different fractions of the violation list [4] . . . 21

4.1 Vulnerabilities mapping between SolidiFI and DASP10 30
4.2 Vulnerabilities mapping between HuangGai and DASP10 31
4.3 Metadata method per bug dataset 33

6.1 Testing metrics for each classification model tested 49
6.2 Time table for the Decision Tree Classifier model 50
6.3 Testing metrics for each regression model tested 51
6.4 Time table for the Decision Tree Regressor model 53
6.5 Testing metrics for each ensemble model tested 54
6.6 Time table for the Random Forest Regressor model 54

B.1 Full metrics for all models tested. TE stands for testing metrics and TR stands for training metrics 72

Acronyms

- **APBD**: Average Percentage of Bugs Detected
- **BoW**: Bag of Words
- **Daap**: Distributed Application
- **DASP**: Decentralized Applications Security Project
- **DASP10**: Decentralized Applications Security Project Top 10
- **GNN**: Graph Neural Network
- **GGNN**: Gated Graph Neural Network
- **HEF**: Hand-Engineered Features
- **LSTM**: Long Short Term Memory
- **ML**: Machine Learning
- **NLP**: Natural Language Processing
- **RFECV**: Recursive Feature Elimination with Cross Validation
- **RNN**: Recurrent Neural Network
- **RV**: Runtime Verification
- **SARIF**: Static Analysis Results Interchange Format
- **SMOTE**: Synthetic Minority Oversampling Technique
- **US**: United States
- **USD**: United States Dollars

1

Introduction

Contents

1.1 Work Objectives 4

1.2 Contributions 5

1.3 Organization of the Document 6

Bug analysis tools must find a balance between false positives and false negatives, therefore it is likely that any given analysis has a number of both. However, developers have a particular discontentment for the former. Each false positive bug report implies time wasted in manual analysis, and developers tend to ignore tools that waste their time. A report by Kremenek and Engler [5] showed that developers quit tools that present between 10 to 20 false positives in a row. For that reason it is crucial that bug reports are organized and prioritized by their likelihood of being true bugs, so that developers can focus their efforts on reports that are more likely to cause real issues.

In this project, we aim to improve the SmartBugs framework by extending it with a bug prioritization mechanism. SmartBugs [1] is an extensible and easy-to-use execution framework that simplifies the execution of analysis tools on smart contracts written in Solidity, thus allowing easy execution of multiple analysis tools. Solidity [6] is the main programming language through which smart contracts are written for the Ethereum blockchain. As of the time of writing, Ethereum sits as the second biggest blockchain-based platform, worth almost 500 billion US dollars. Most of this value comes from Ethereum's capacity to deploy distributed applications (Dapps) that are executed across a decentralised network of nodes. As a key component of a half a trillion dollar industry, it is crucial that smart contracts written in Solidity have no dangerous bugs that affect the integrity of the Dapps they define. Even the smallest of bugs could lead to catastrophic consequences, such as TheDAO exploit [7] that resulted in the theft of 50 million USD, and that could have been fixed by exchanging two lines of code. Unfortunately, writing bug-free smart contracts is no trivial feat, and SmartBugs was created in order to allow its users to analyse their smart contracts with multiple tools. As of today, SmartBugs makes use of 11 analysis tools, that give 11 different outputs for each smart contract analysed. However 11 different outputs means a much greater number of bug reports, both true and false positives.

An empirical study on the status of smart contracts in the Ethereum blockchain using SmartBugs [2] resulted in 93% of 47,587 analysed contracts to be marked as vulnerable, foreshadowing a large ratio of false positives. Another difficulty appears as the consequence of the variety of tools. Some of these 11 tools are static analysis tools (e.g. Securify [8]), some are dynamic analysis tools (e.g. Maian [9]) and some, like HoneyBadger [10], even focus on finding code structures that might look like bugs, but that are in fact not bugs at all. This means that whatever approach taken must consider this heterogeneity. Each tool has its own internal methodology and gives a different output. One of the features of SmartBugs is to parse every tool output and produce a consistent result in the Static Analysis Results Interchange Format (SARIF) [11] that provides the only common ground between all results. This heterogeneity proves to be both and advantage and a disadvantage that will be further discussed in future sections.

Previous work has been performed on the issue of bug prioritization, but all previous approaches focus on prioritization of bug reports created by a single tool. Since our goal is to improve SmartBugs, in this project we focus on prioritization of multiple bug reports from different tools. Our context opens

up new opportunities that can be explored to improve the prioritization process. For example, it brings the possibility of using consensus between multiple tools.

It seems reasonable to assume that if a certain bug is detected by multiple tools simultaneously, it is more likely to be a true bug. Alternatively, bugs reported by a single tool might prove to be less trustworthy reports. There also exist other tools such as eThor [12], that can probably guarantee the absence of Reentrancy bug and would therefore be useful in a consensus scenario. Like previous work that we will explore in a later section([4], [3]), we will also explore machine learning techniques to reach our solution.

1.1 Work Objectives

The main goal of this project is to extend the SmartBugs framework with a bug prioritization mechanism capable of ranking analysis reports created by multiple tools. In particular, we hope to be able to quickly present the vast majority of true positives from multiple different analysis tools to users who want to have their smart contracts analysed.

This is achieved with help from a machine learning model whose features are the SARIF output of each tool. As mentioned above, one of SmartBugs' features is to reduce all bug reports to the SARIF format. Short for Static Analysis Results Interchange Format, SARIF [11] is a format designed to integrate and standardize results from multiple analysis tools. Through the use of SARIF, every tool present in SmartBugs has an equivalent representation in the machine learning model. This is, all tools present the same features (the SARIF features) to the model, instead of using tool-specific features that might not be available to other tools. This provides the model with information on consensus between the analysis of each tool.

The model must then use this consensus information to learn how to prioritize bug reports, presenting those considered as most likely to be true positives towards the beginning of the prioritized list of bug reports. If the tool presented in this project achieves good results, than a developer tasked with analysing a list of prioritized bug reports provided by SmartBugs will have a much easier time doing so than another developer analysing the same, but non-prioritized, list of reports. The better prioritization is, the more time the developer saves while looking for bugs in his program, and the less likely he is to abandon the framework.

To achieve the best result possible in prioritization, it is important to understand what machine learning algorithm would be a better fit for our context. We experiment with multiple models and analyse their results. We try classification models, regressor models and ensemble models to know which one is the most suitable for this environment. The results also provide an analysis on the effectiveness of the SARIF format in this context. If SARIF provides enough information for consensus, than the machine

learning model is more likely to have good results, if it does not, the opposite may apply.

To test and train the machine learning model, this work must also have an annotated bug dataset. Our dataset was created by joining multiple other datasets together and is now available for use in other works. As it merges multiple other datasets into one, it became a necessity to parse them all to the same format. This format must be flexible enough to take advantage of consensus, something we found lacking in most bug datasets. Therefore our own format had to be created.

In conclusion, this work addresses the following research questions:

- **RQ1:** Does the SARIF format provide enough information on consensus to achieve reasonable results in bug prioritization?

- **RQ2:** Can consensus of bug reports between multiple tools be used to better predict true and false positives?

- **RQ3:** What are some effective machine learning algorithm to prioritize bug reports from multiple tools in the context of SmartBugs?

1.2 Contributions

This work makes contributions to the research of bug prioritization from reports provided by simultaneous analysis of multiple analysis tools. We explore the information present in the SARIF format and how it is useful as a feature for consensus in the context of our environment. We also explore how to use consensus between multiple tools to achieve efficient bug report prioritization, as well as an analysis of multiple machine learning models to learn which one is the most suitable this purpose.

We analyse classification and regression models, explain why the latter is more suitable than the former, and follow the leads of our related work to test the ensemble models theorized to be the most efficient. We contribute that, according to our findings, the Random Forest Regressor model is the most suitable machine learning model for this thesis. This model achieves an APBD value of 0.933 and is capable of finding all true bugs in the dataset after analysing only 21% of bug reports, an almost 5x increase over non-prioritization .

Additionally, our project contributes a new bug dataset of 1,657 faulty smart contracts to the community. This dataset is composed of multiple other datasets in a unique format. The format is more expressive than any other we have come across and we hope to see it implemented in other bug datasets to come.

1.3 Organization of the Document

The rest of this thesis is is organized as follows:

- Chapter 2 (Background), where relevant background information is presented in order to allow readers to better understand this work;
- Chapter 3 (Related Work), where previous work on the subject is discussed and analysed, explaining the reasoning behind this thesis' approach;
- Chapter 4 (Data Processing), where we discuss how information is prepared and propose a new standard for metadata in bug datasets;
- Chapter 5 (Machine Learning Model), where we discuss the models implemented and their features;
- chapter 6 (Evaluation), where we present our evaluation metrics as well as their results;
- Chapter 7 (Conclusion), where we present some final thoughts, the main contributions of this work and future work.

2

Background

Contents

2.1 The Problem of False Positives 9

2.2 Bug Prioritization 9

2.3 SmartBugs 10

This chapter will explore all concepts and projects necessary to understand the solution presented in the following chapters. Specifically, it focuses on the concept of true and false positives, bug prioritization, as well as the SmartBugs framework.

2.1 The Problem of False Positives

A true positive occurs when a report correctly (true) gives a positive (positive) result. In the context of this work a true positive refers to a bug report that is indeed a bug. A false positive occurs when a report incorrectly (false) gives a positive (positive) result. In the context of this work a false positive refers to a bug report that is not actually a bug. The same logic applies to true and false negatives.

Ideally a bug analysis tool would only give true positives and true negatives, meaning that it could find all bugs present in smart contracts and give no incorrect report. In reality however, both false positives and false negatives occur during the execution of most analysis tools. False negatives happen when a bug is not detected by the tool and false positives happen when an incorrect bug report is given.

False positives represent a great hurdle for widespread use of bug analysis tools. This is because false positives imply time wasted on manual analysis of nonexistent bugs. Legunsen et al. [13] reported spending 1,200 hours to inspect, discuss and patch 852 violations, averaging 1.4 hours per violation. Breno at all. found in their study [4], that there was little time difference in analysing a true or a false bug report, only that false positives skip the patch implementation phase. Therefore, each false positive has a significant time loss implication for the developer. If a bug analysis tool returns too many false positives, developers lose faith in said tool and abandon it due to its high time cost. Kremenek and Engler concluded in their study [5] that in order for a developer not to lose faith in a bug analysis tool, it must not present any false positives in the first 3 reports and have no more than 10-20 false reports in a row. Should any of these conditions be broken, there is a high chance that the developer will stop the inspection. In this work we will thrive to maintain these conditions.

2.2 Bug Prioritization

The issue of bug prioritization has existed ever since bugs came to be, that is to say since the beginning of computer science. Initially, when all bug reports were written manually by developers, the priority of a bug was a field that was manually filled by the developer in question. Taking into account how much impact and consequence it had on the system, developers would use their experience to decide on a suitable priority level. The issue with this approach is the heavy reliance on the developer's experience and familiarity with the system. After all, a developer with little to no experience and that is not familiar with the system could hardly be expected to give accurate priority values to bug reports.

This became apparent when bug reports became open to the public. Without the techinical expertise of developers and with no deep understanding of the inner works of a system, the general public was unable to give accurate levels of priority to their bug reports. To solve this issue, junior developers were hired to test bug reports and assign priorities to them. This approach however, still maintains the reliance on developer experience and is unable to keep up with large amount of bug reports. This issue was doubled when automatic bug analysis tools appeared and provided even larger number of bug reports, many of which are false positives.

As seen in the previous section, false bug reports come with high time cost for the developers charged with exploring them. As such it became necessary to find a way to prioritize large amounts of bug reports without depending on the experience of a single developer. Two main approaches were taken: Natural Language Processing (NLP) and Machine Learning (ML). NLP is used to read manual bug reports written by real users in an effort to understand what it says and then apply a suitable level of priority. Since this work focuses on bug reports provided by automatic analysis tool this approach is not as relevant, but some insights from it have been taken for the purpose of prioritizing bug reports from analysis tools.

Machine learning aproaches take information from a selected number of features and try to figure out patterns that could help predict the correct priority level of a bug report. It is an efficient way to process large amount of reports with high accuracy. This approach has been used to prioritize both manual reports by users and automatic reports by analysis tools, which are the most relevant for this work. Chapter 3 will take a closer look at some examples on how machine learning can be used to assign priorities to bug reports provided by analysis tools. The selected works analyse a variety of machine learning techniques and how they can be used for the purpose of bug prioritization.

2.3 SmartBugs

SmartBugs [1] is an extensible and easy-to-use execution framework that simplifies the execution of analysis tools on smart contracts written in Solidity. As described in the introduction chapter, SmartBugs was developed to satisfy a necessity to evaluate the validity of smart contracts written for the Ethereum blockchain. Its objective was to create an environment where users could test their smart contracts, developers could test their bug analysis tools, and researchers could perform empirical studies on the state of smart contracts in the blockchain.

SmartBugs architecture is composed of a web dashboard, a command line interface, the runner, the tool configurations, the bug analysis tools and datasets. The two aspects of greater importance are the tools and the datasets. Tools in SmartBugs are implemented as docker images stored in DockerHub [14]. Where images were available said images were used, otherwise they were created and publicized. The

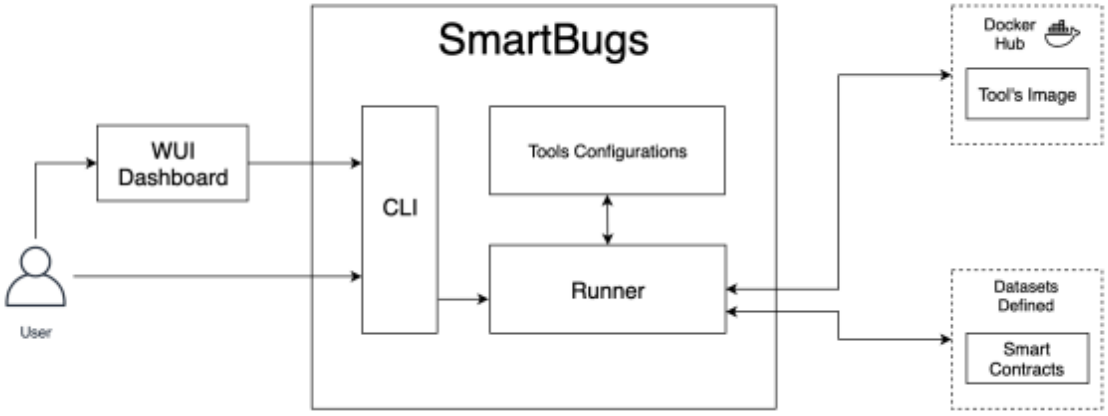

Figure 2.1: SmartBugs Architecture [1]

choice to use Docker images was made to ease the addition of new tools, allow the execution to be reproducible and to use the same execution environment for all tools.

The 11 tools currently supported are: HoneyBadger [10], Maian [9], Manticore [15], Mythril [16], Osiris [17], Oyente [18], Securify [8], Slither [19], Smartcheck [20], Solhint [21] and Conkas [22]. What is important to note here is the wide variety of bug analysis tools. Some are static analysis tools and some are dynamic, each with their own characteristics. HoneyBadger is a particularly interesting example, as it hopes to find not bugs, but honey pots. Honey pots are traps laid by developers that are made to look like bugs, but that are actually not bugs at all. The peculiarity of each tool is part of the appeal of SmartBugs, as it easily allows users to have their smart contracts analysed by multiple tools and angles. As mentioned above, adding more tools is relatively easy, and the number of tools has been steadily increasing with each published work on this framework. The approach presented in our solution must take this design philosophy into account.

The datasets are the smart contracts to be analysed by the tools. The $sb^{curated}$ dataset consists of 143 smart contracts with 208 tagged vulnerabilities. Contracts in this dataset are either real contracts that have been identified as vulnerable or contracts that have been purposely created to illustrate a vulnerability. Developers can use this dataset to test and compare their bug analysis tools with the rest of the market. It can also be used to rank and evaluate the 11 tools against a variety of known vulnerabilities. The next section will describe the results of such an evaluation.

The sb^{wild} dataset contains 47,398 contracts extracted from the Ethereum blockchain. The set of vulnerabilities of those contracts is unknown, as they have not been manually studied by experts. This dataset allows researchers to perform empirical analysis of the state of the blockchain at large. SmartBugs developers have performed an intensive analysis of this dataset that took over 564 days to run and that will be further discussed in the next section.

Category	HoneyBadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0/19 0%	0/19 0%	4/19 21%	4/19 21%	0/19 0%	0/19 0%	0/19 0%	4/19 21% (1)	2/19 11%	5/19 26%
Arithmetic	0/22 0%	0/22 0%	4/22 18%	15/22 68%	11/22 50% (2)	12/22 55% (2)	0/22 0%	0/22 0%	1/22 5%	19/22 86%
Denial Service	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/ 7 0%
Front Running	0/7 0%	0/7 0%	0/7 0%	2/7 29%	0/7 0%	0/7 0%	2/7 29%	0/7 0%	0/7 0%	2/ 7 29%
Reentrancy	0/8 0%	0/8 0%	2/8 25%	5/8 62%	5/8 62%	5/8 62%	5/8 62%	7/8 88% (2)	5/8 62%	7/ 8 88%
Time Manipulation	0/5 0%	0/5 0%	1/5 20%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	2/5 40% (1)	1/5 20% (1)	3/ 5 60%
Unchecked Low Calls	0/12 0%	0/12 0%	2/12 17%	5/12 42% (1)	0/12 0%	0/12 0%	3/12 25%	4/12 33% (3)	4/12 33% (1)	9/12 75%
Other	2/3 67%	0/3 0%	0/3 0%	0/3 0%	0/3 0%	0/3 0%	0/3 0%	3/3 100% (1)	0/3 0%	3/ 3 100%
Total	2/115 2%	0/115 0%	13/115 11%	31/115 27%	16/115 14%	17/115 15%	10/115 9%	20/115 17%	13/115 11%	48/115 42%

Figure 2.2: SmartBugs results in the $sb^{curated}$ dataset [2]

Category	HoneyBadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0 0%	44 0%	47 0%	1,076 2%	0 0%	2 0%	614 1%	2,356 4%	384 0%	3,801 8%
Arithmetic	1 0%	0 0%	102 0%	18,515 39%	13,922 29%	34,306 72%	0 0%	0 0%	7,430 15%	37,597 79%
Denial of Service	0 0%	0 0%	0 0%	0 0%	485 1%	880 1%	0 0%	2,555 5%	11,621 24%	12,419 26%
Front Running	0 0%	0 0%	0 0%	2,015 4%	0 0%	0 0%	7,217 15%	0 0%	0 0%	8,161 17%
Reentrancy	19 0%	0 0%	2 0%	8,454 17%	496 1%	308 0%	2,033 4%	8,764 18%	847 1%	14,747 31%
Time Manipulation	0 0%	0 0%	90 0%	0 0%	1,470 3%	1,452 3%	0 0%	1,988 4%	68 0%	4,069 8%
Unchecked Low Calls	0 0%	0 0%	4 0%	443 0%	0 0%	0 0%	592 1%	12,199 25%	2,867 6%	14,656 30%
Unknown Unknows	26 0%	135 0%	1,032 2%	11,126 23%	0 0%	0 0%	561 1%	9,133 19%	14,113 29%	28,355 59%
Total	46 0%	179 0%	1,203 2%	22,994 48%	14,665 30%	34,764 73%	8,781 18%	22,269 46%	24,906 52%	44,589 93%

Figure 2.3: SmartBugs results in the sb^{wild} dataset [2]

2.3.1 Empirical Study

SmartBugs developers used their platform to make an empirical review of the automated analysis tools available to SmartBugs at the time. The authors applied their framework both to the $sb^{curated}$ dataset and to the sb^{wild} dataset. Figure 2.2 shows the results of the $sb^{curated}$ dataset. Some conclusions can be drawn from this information. First, it is clear that tools have their strengths and weaknesses, better at finding bugs in some categories when compared to others. Second, we can notice that denial of service bugs were never found, showing a lack of tools capable of finding vulnerabilities of this type. And third, only 42% of all bugs were found, leaving more than half yet to be reported. This once again shows the current limitations of the available tools.

Figure 2.3 shows the results for the sb^{wild} dataset. This dataset contains more than 47 thousand contracts and it took 564 days and 3 hours to complete a total of 428,337 analyses by all tools. The analysis wielded some very interesting results. First it can be noted that 93% of contracts were marked as vulnerable, with the Oyente tool alone reporting 72% of contracts to have arithmetic vulnerabilities. This suggests a high number of false positives. As described in the previous section, false positives are one of the biggest hurdles that must be surpassed for these tools to achieve widespread use. Another interesting takeaway is that only a small number of reports were in consensus of 4 or more tools (937 for Arithmetic and 133 for Reentrancy). Most reports came from a single tool.

3

Related Work

Contents

3.1 An Empirical Assessment of Machine Learning approaches for Triaging Reports of a Java Static Analysis Tool 15

3.2 Prioritizing Runtime Verification Violations 17

A lot of work has been done on the issue of bug prioritization over the years. As far back as 2013, Alenezi and Banitaan [23] developed a system to help developers triage bug reports manually written by users by leveraging machine learning. Their study took care to analyse what features and classifiers to choose, a trend that would continue.

While work continues on bug prioritization of manual reports, this thesis is focused on bug prioritization of reports provided by automatic analysis tools. Far less work have been done on this side of the issue, but recent studies provide some good insights. This chapter will analyse some of these studies.

3.1 An Empirical Assessment of Machine Learning approaches for Triaging Reports of a Java Static Analysis Tool

Koc et al. explored in their aptly named report "An Empirical Assessment of Machine Learning approaches for Triaging Reports of a Java Static Analysis Tool" [3] how effective different machine learning techniques are at classifying a bug report as a true or a false positive. Their motivation, as well as ours, was to lower the cost that false positives represent to developers in order to make static analysis tools more useful. The goal of this study is to build a classifier that identifies a bug report as either a true or a false positive. They studied the following approaches:

- **Learning with Hand-engineered Features (HEF):** This method represents the approach used by the majority of studies up to this point. Hand-engineered features are features manually created by experts to provide information on weather a certain bug report represents a true or a false positive. By design, this approach never looks at the source code that is represented by the hand-engineered feature, as classifiers only ever see the hand-engineered features themselves and not the source code that triggered it. Another thing to take into consideration is the fact that using hand-engineered features limits the performance of the machine learning model to the information provided by said features. For their work, Koc et all. adapted and used the hand-engineered tools from a previous study by Trip et al. [24]

- **Program Slicing for Summarization:** This technique is used as an intermediate step in the following approaches, and refers to a way to grab a summary of the program that caused the bug report. To do this Koc et al. use a method called backward slicing with help from Joana [25], a program analysis framework for Java. Backward slices are a way to isolate what part of the program could have influence over a certain event. In this context it means that the classifier will only look at the part of code that could effect the bug report, not introducing noise from the rest of the program.

- **Bag of Words (BoW):** The idea behind this approach, inspired in text classification problems, is to used tokens extracted from the backward slice of the program as the features for the machine learning classifier. From here on we refer to tokens as ”words” as that is how this technique is usually implemented. It is used in this study in two different variants: as a binary vector of words (BoW-Occ), where for each slice we mark words that appear in the slice as 1 and words that don't appear in said slice as 0. The second variant counts the frequency of words, keeping track of how many times each word exists in the program slice (BoW-Freq). Bag of Words ignores order, only keeping track of what words / frequency of words exist in the program slice.

- **Recurrent Neural Networks (RNN):** In contrast to the previous approach, order in RNN is important. RNN processes a sequence of words where features take into consideration not only the current word, but all previous words. That is to say that the meaning of a word (in this case related to true and false positives) takes into consideration the meaning of all words analysed before it. This an intuitive approach for text analysis as the final word of a sentence changes meaning based on the rest of the words in the same sentence, and the same applies to code analysis. More specifically, this work uses a version of RNN known as LSTM (Long short term memory), suitable for the long sequences expected in this context. To use LSTM, authors needed to transform program slices into sequences of tokens by including the following transformations: Data Cleansing and Tokenization (LSTM-Raw), Abstracting Numbers and String Literals (LSTM-ANS), Abstracting Program-specific Words (LSTM-APS) and Extracting English Words From Identifiers (LSTM-Ext). Each stage of the transformation is represented as a different approach in the results section.

- **Graph Neural Networks (GNN):** While RNN sees programs as a sequence of tokens, GNN does something similar but represented as a graph. Authors chose to focus their efforts on a variation of GNNs called Gated Graph Neural Networks (GGNN). GGNNs had been used to learn properties about programs, but had not yet been applied to classify static analysis reports or to learn from program slices. The authors used three different ways to initialize the input node representations: Using Kind, Operation, and Type Fields (KOT), Extracting a Single Item in Addition to KOT (KOTI) and Node Encoding Using Embeddings (Enc). Once again, each variant is considered a different approach in the results section.

The static analysis tool selected for the study was FindSecBugs (version 1.4.6) [26], a popular security checker for Java web applications. Two benchmarks were chosen as well, the OWASP Benchmark [27], used to benchmark previous static analysis tools in the past, and one manual benchmark created and labelled by the authors (RW). The manual benchmark was divided into two to express two different scenarios. RW-Rand is meant to mimic a scenario where a program is developed with help from a static tool, this is the tool is constantly run on the same program and is able to learn program

specific identifiers. Second scnenario happens when the analysis tool is used to analyse a new program, and therefpore must train in one program to be later applied to another, this scenario is represented in RW-PW. The metrics tested are precision, to measure how many of the true positives classifications were correct, recall to measure how many of the true positives were correctly classified and accuracy to measure how many samples were correctly classified in total.

The results presented in figure 3.1 show the various approaches and their variants. HEF variants refer to different classifiers tested while BoW, RNN and GNN variants are the ones presented above. The results show LSTM to have the best overall performance throughout the experiment, with LSTM-Raw able to reach 100% accuracy for the OWASP dataset. This proves there is ample credibility to this triage technique. Looking more carefully at results we can see the gap between LSTM and other approaches increase in the RW-PW dataset, meaning that LSTM proves more adaptable at triaging bug reports of new programs when compared to other approaches, with GNN as a close second. Authors theorise this is because these are the two approaches that keep the most information from the initial program, keeping both code and order/relationship.

We can also analyse the effect of different levels of data preparation in performance. The goal goal of data preparation is to provide the most effective use of information that is available in the program context. LSTM-Ext produced the best overall accuracy of all datasets. LSTM-Raw did achieve 100% accuracy for the OWASP dataset but authors conclude this has to do with the OWASP benchmark itself. Specifically, in order to test static analysis tools OWASP has variables with names such as ”safe”, ”unsafe” and ”tainted” that giveaway the answer and that LSTM-raw utilizes to ”cheat”. Even for less successful approaches, GNN variants with better data preparation (GGNN-Enc) prove more accurate then their less prepared counterparts. This proves the significant positive influence that data preparation can have on the accuracy of the models.

In summary, this study proved the validity of triaging bug reports from static analysis tools with machine learning, as well as multiple techinical conclusions. For their context, authors found the recurrent neural network of LSTM to be the most accurate of all aproaches and that data preparation can have great positive influence in the accuracy of the solution.

3.2 Prioritizing Runtime Verification Violations

This study by Miranda et al. [4] has a slightly different nuance from the last. While Koc et al. were looking to triage their bugs and drop those considered as false positives, Miranda et al. only wish to prioritize them. This is, they will re-organize bug reports and change their order based on the probability of being a true or a false bug. Reports considered more likely to be true will be put towards the beginning of the list, while reports less likely to be true will be presented last. This way, no incorrectly identified true positive

dataset	approach	recall		precision		accuracy	
OWASP	*LSTM-Raw*	100.00	0	100.00	0	100.00	0
	LSTM-ANS	99.15	0.74	98.74	0.42	99.37	0.42
	LSTM-Ext	98.94	1.90	99.57	0.44	99.16	1.16
	LSTM-APS	98.30	0.42	99.14	0.21	98.53	0.27
	BoW-Occ	97.90	0.45	97.90	1.25	97.47	0.74
	BoW-Freq	97.90	0.45	97.00	0.25	97.26	0.31
	GGNN-Enc	92.00	5.00	94.00	5.25	94.00	1.60
	HEF-J48	88.50	1.65	75.10	0.50	79.96	0.21
	GGNN-KOTI	78.50	6.25	81.00	2.50	79.00	1.95
	HEF-RandomForest	85.50	1.65	74.10	0.65	78.32	0.50
	GGNN-KOT	80.00	3.25	77.50	2.00	78.00	0.95
	*HEF-K**	84.70	2.05	73.60	0.90	77.68	1.37
	HEF-MLP	79.10	7.00	70.90	2.10	73.00	1.27
RW-Rand	*LSTM-Raw*	90.62	2.09	86.49	3.52	89.33	2.19
	LSTM-Ext	90.62	4.41	85.29	3.20	89.04	1.90
	LSTM-APS	91.43	4.02	86.11	3.99	87.67	2.85
	LSTM-ANS	89.29	2.86	84.21	3.97	87.67	1.59
	BoW-Freq	86.10	2.30	87.90	1.85	87.14	1.85
	BoW-Occ	84.40	4.45	87.50	3.85	85.53	2.45
	GGNN-KOTI	83.00	4.50	84.00	3.50	84.21	1.55
	*HEF-K**	80.00	3.95	85.70	2.30	84.00	0.89
	HEF-RandomForest	75.00	1.40	84.40	3.20	84.00	0.93
	GGNN-KOT	89.00	7.00	80.00	7.00	83.56	3.48
	GGNN-Enc	80.00	6.00	78.00	4.50	82.19	3.63
	HEF-J48	78.10	2.15	82.40	0.90	81.33	0.92
	HEF-MLP	71.40	2.80	86.20	6.10	81.33	1.97
RW-PW	*LSTM-Ext*	78.57	12.02	76.19	5.20	80.00	4.00
	LSTM-APS	70.27	14.59	76.47	6.70	78.48	3.33
	LSTM-ANS	62.16	25.58	75.76	7.02	74.68	3.85
	LSTM-Raw	67.57	31.91	79.66	8.40	74.67	4.08
	GGNN-Enc	77.00	36.00	75.00	19.50	74.67	5.89
	GGNN-KOT	77.00	29.50	72.00	16.25	74.00	5.84
	HEF-MLP	58.10	14.65	70.40	9.40	73.08	7.76
	GGNN-KOTI	65.00	33.50	75.00	11.00	72.02	5.12
	*HEF-K**	66.10	24.50	60.60	14.90	68.00	9.75
	HEF-J48	60.70	11.65	72.70	12.80	65.33	8.04
	HEF-RandomForest	62.50	24.30	60.30	5.55	63.44	2.67
	BoW-Occ	50.00	12.90	65.00	22.30	51.32	4.61
	BoW-Freq	47.80	16.50	65.70	14.70	51.25	8.55

Figure 3.1: Results from Koc et al. experiments [3]

will be lost to the user. Therefore, strictly speaking, the authors of this paper are not trying to classify bugs as true or false, but instead trying to calculate likelihood of being true positives and then using that information to organize a list of prioritized bug reports. This approach has the same advantages as the previous one, saving precious developer time, while it avoids the downside of missing true bugs that have been incorrectly identified as false positives.

To do this, authors prepared five probability categories with the values of: very-low, low, medium, high and very-high probability of being true bugs. The various bug reports were then classified to one of these categories with help from a classifier machine learning model, and then later prioritized according to these values.

For a quick introduction of Runtime Verification (RV), it is a dynamic (while the last study was static) analysis technique that works by keeping track of properties throughout a test's execution. Should any of the defined proprieties be violated, this is not conform to a predefined specification, then a violation is raised, warning the user of a possibility of a bug. The problem is that, according to previous studies [28], only 10% of reports presented by runtime verification represent true bugs. As such, authors of this paper developed RVPrio, an automated approach for prioritizing RV violations in order of likelihood of being true bugs. Using RVPrio, developers can focus their time on the violations with the highest probability of being true.

As with any machine learning project, the first step is to choose which features to use. Authors selected features in a 2-step process. First, features are extracted from properties associated with violations and from the source code that triggered violations. This is inline with the conclusions from Koc et al.'s study, which concluded that features selected from source code were of more use than those that were made artificially. However, this resulted in a very large number of features, which can have a negative effect in performance. To solve this issue authors used Recursive Feature Elimination with Cross-Validation (RFECV), an algorithm that discards bad features to select a final, more concise, set of useful features.

With features selected, authors tested different binary classifiers to see how good they were at predicting if violations represent true or false bugs. To do this, they used the following metrics: Precision as a proxy for measure of amount of false positives, Recall as a proxy for measure of amount of false negatives and F1-Score an average of both. Higher is better for all cases, and they are compared to a DummyClassifier that chooses at random and therefore scores 0.5 in all metrics. Results are presented in table 3.1.

Analysing the results, we can see that the top four results (excluding logistic regression) are ensembles. Ensembles are machine learning models that are trained by joining the results of multiple other models. This is something suggested by Koc et al. during the future work section of the previous study. There he suggested that ensemble classifiers might provide better classification performance

Classifier	Precision	Recall	F1-score
Gradient Boosting Classifier	0.74	0.73	0.73
Logistic Regression	0.76	0.70	0.71
Random Forest	0.73	0.70	0.71
AdaBoost	0.74	0.68	0.70
Decision Tree	0.70	0.68	0.69
Gaussian Processes	0.67	0.65	0.66
Neural Net	0.74	0.65	0.64
Nearest Neighbors	0.67	0.63	0.64
Naive Bayes	0.62	0.75	0.55
Linear SVM	0.70	0.54	0.53
DummyClassifier	0.50	0.50	0.50

Table 3.1: Performance of different classifiers in RVPrio [4]

when compared to simple classifiers, and RVPrio confirms this to be the case.

Authors also tested the end result of RVPrio as a prioritizer of violations. Since Gradient Boosting Classifier had the best performance, that is the variant chosen for the following results presented in figure 3.2. The test had a list of RV violations, some true and some false, and the objective is to prioritize the list in such a way that true positives are at the top. The x axis has the number of violations analysed and the y axis has the percentage of true bugs revealed. Different strategies are represented by the differently colored lines.

The blue line represents an optimal prioritization, where all true bugs are shown at the beginning of the list, represented by the linear increase in true bugs found as the violations are analysed. After all true bugs are found only false reports remain, and as such the line remains horizontal after it reaches 100%. The red line represents random prioritization. In this case true and false reports are randomly mixed together and we would have to, on average, analyse half the violations to find half the bugs, explaining the constant trend of the line. Spikes on the line are due to randomness. Lastly, the green line represents RVPrio. As can be seen, the line closely, but not perfectly, resembles the optimal blue line. Most bugs are quickly revealed in the first analysed violations, with only a few being incorrectly put towards the end of the list. RVPrio reached 90% effectiveness of the optimal prioritization.

The advantages of this prioritization are best illustrated in the table of table 3.2. Here it can be seen that if a developer would only analyse 5% of the violations in the violation list using random prioritization, then he would have only found 4.17% of true bugs. However, analysing the same 5% of violations in a list prioritized with RVPrio, than the same developer would have found 17.26% of all bugs, a 4.13x increase over random prioritization. Using RVPrio, a developer would have found 88.10% of all bugs by looking at only 25% of violations, saving a great deal of time. To be guaranteed to find all bugs, developers must still manually analyse all violations, but as we can expect the majority of developers to be satisfied with only 90% of bugs found, developers can focus their time on only a small portion of the list.

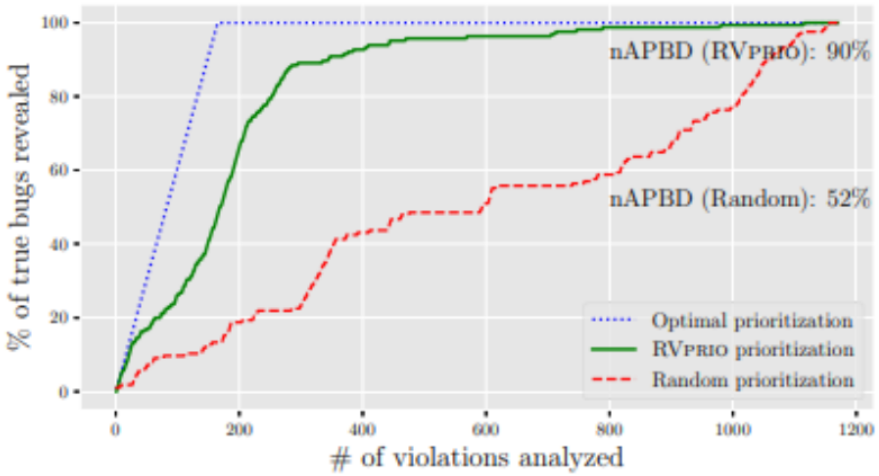

Figure 3.2: RVPrio prioritization performance [4]

% of violations	5%	10%	15%	25%	50%	75%
RVPrio-GBC	17.26%	32.74%	60.71%	88.10%	96.43%	97.62%
Random	4.17%	5.36%	14.29%	24.40%	56.55%	64.29%
Improv.	4.13x	6.11x	4.25x	3.61x	1.71x	1.51x

Table 3.2: RVPrio percentage of true bugs revealed at different fractions of the violation list [4]

4

Data Processing

Contents

4.1 Data Preparation Pipeline 25
4.2 Datasets 25
4.3 From SARIF to bug reports CSV 27
4.4 From metadata to solutions CSV 28
4.5 Final CSV 33

Following the steps of the related work presented above, we have also chosen to implement a machine learning model to help us classify bug reports as either true or false positives. This classification will later be used to prioritize our reports, achieving the goals stated in Chapter 1.

While the details of the machine learning model will be discussed during the next chapter, this chapter discusses the data collection and processing that was used for the development of this thesis. This is the preparation of the data that will be fed to the machine learning model. In addition we also discuss some practices that we believe could be useful as standards for bug datasets and that could facilitate their use for a wider range of works.

From the next section onwards we will present and discuss every step in the data preparation pipeline, including our reasoning behind each decision taken.

4.1 Data Preparation Pipeline

The data preparation pipeline followed in this project is the following:

1. Acquire datasets
2. Run SmartBugs on acquired datasets
3. Parse output SARIF file from SmartBugs' execution to a bug reports CSV file
4. Parse the dataset's metadata (information about the actual bugs in the dataset) to a solutions CSV file
5. Join the bug reports CSV file and a solutions CSV file, into a final CSV file containing all bug reports as well their solution (i.e. if they represent a true or false positive bug report).

The next sections will discuss each step of this pipeline in detail.

4.2 Datasets

For this project, we have chosen to implement supervised machine learning techniques. This is, the machine learning model is trained with already classified data. In our case it means that the bug reports provided to the machine learning model will already have information that classifies them as either true or false positives. Supervised machine learning techniques allow developers to train more accurate models when compared unsupervised machine learning techniques (techniques that allow us to train models without previously classified data), but it requires that we prepare data that already has said classification.

In order to achieve this, our bug datasets must already have metadata information about which bugs exist in the datasets so that we can later match that information with the reports provided by SmartBugs' execution. It is a consequence of this method however, that we trust the classification provided by the dataset itself. Should the classification that is used to train the machine learning model be proven to be untrustworthy, then we can also assume that the trained machine learning model will also be as inaccurate as the data it is trained on.

Taking all this into consideration, the datasets used in this project are:

1. SmartBugs' $sb^{curated}$ dataset, including 143 faulty smart contracts manually analysed by experts
2. SolidiFI's bug dataset [29], containing 550 faulty smart contracts created through bug injection
3. HuangGai's manual dataset [30], containing 964 faulty smart contracts manually analysed by experts

The datasets chosen can be roughly separated into two categories:

1. Datasets that have been manually verified by experts
2. Datasets that have been automatically generated through bug injection, but not manually verified by experts

Datasets that have been manually verified have the advantage of being more likely to be correctly classified. As an expert familiar with the bugs in question has verified its content, the probability of it being incorrectly classified is smaller when compared to a purely automatic procedure. The downside of this approach is its high cost of development. After all, an expert must spend a significantly high amount of time to manually evaluate all the bugs in the dataset. As a result, these datasets are often much smaller than their automatic counterparts.

Datasets that have been automatically generated through bug injection but not manually verified by experts are the opposite. The completely autonomous process through which they are created requires very little of the developers time if we consider that the tool that provides the injection has already been developed. As a result, these datasets are usually much larger than those that have a higher human component in their development cycle. The downside is that it leaves more room for error in classification, as no expert will verify the results provided by the bug injection tool.

For this project we chose a mixture of the two, using manually verified datasets when possible, and then bolstering the numbers of our dataset through bug injection.

It is also important to notice that no further datasets could added to this work, due to exponential time it takes to run all contracts of said datasets with SmartBugs. As mentioned above, SmartBugs has 11 different analysis tools, so each contract is run 11 different times, making the analysis process very lengthy. Further datasets would have taken too long to run, making them unviable for our project.

After acquiring the datasets, the next step is to run the SmartBugs framework with all tools enable and save the output SARIF file. As mention before, this is an extremely time consuming process. Having the results in hand, the next section explains the start of the data processing proper.

4.3 From SARIF to bug reports CSV

The first step of data preparation is to join the multiple reports given in SARIF format in such a way that we can see which reports belong to the same bugs. Do note that if all 11 tools present in SmartBugs report a bug, there would exist 11 different reports concerning the same bug. The goal is to join all reports of the same bug together in the same line of a CSV file, thus allowing the machine learning model to grasp information on consensus.

Each CSV line has a bug_id, followed by the report of each tool. Each tool's report is further divided into the rule_id of the bug that was found by the tool, as well as the level of the bug the tool characterized it as (error or warning). Should a tool not have found the bug, than the columns related to it will be left empty. The final result is a CSV file where each line represents a bug, and each column has the reports of each tool on that same bug. These columns will be used as our machine learning model's features, this is the information though which it learns.

Take for example the line:

```
1 buggy_40.sol:620,UncheckedLowCalls_7,warning,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,AccessControl_19,error
```

The line expresses that bug id buggy_40.sol:620, was reported by two tools. First it was reported as a warning by SmartCheck [20] with a rule_id of UncheckedLowCalls_7 and then as an error by Slither [19] with a rule_id ofAccessControl_19.

This approach, however, raises a technical difficulty: How to determine if the two bug reports concern the same bug? When a large file is analysed, multiple bugs can be present, and consequently multiple bug reports may appear on the analysis tools. The issue of how to organize these bug reports appears.

To resolve this issue, we need a systematic way to join our reports together. The solution implemented in this paper is as follows:

1. Two reports concern the same bug if they are reported on the same line and source file
2. Two reports concern the same bug if the bug in question is of the same category

If both of the rules above apply we consider these reports to be about the same bug, if not we consider them to relate to different bugs.

The first item is easy to verify, but the second proved more complicated. When it comes to category, we have mapped each of the rule_id's of each tool to a category and use said category in the rules

described above. This is, if a tool reports an arithmetic bug, while a different tool reports an access control bug on the same line, then we consider them to be referring to different bugs.

The categories we have chosen to use for this project follow the DASP10 (Decentralized Applications Security Project Top 10) regulation [31]. DASP10 divides solidity bugs into 10 categories:

1. **Reentrancy** Occurs when external contract calls are allowed to make new calls to the calling contract before the initial execution is complete
2. **Access Control** Vulnerabilities that give attackers straightforward ways to access a contract's private values or logic
3. **Arithmetic** This category contains issues such as integer overflow and underflow
4. **Unchecked Return Values For Low Level Calls** Vulnerabilities created by low levels calls such as call() and send() failing silently without the developer noticing and while the program continues executing
5. **Denial of Service** Vulnerabilities that stop the service
6. **Bad Randomness** Vulnerabilities created by variables that are not truly random
7. **Front-Running** Refers to vulnerabilities that occur when users spend more gas to have their transaction run first
8. **Time Manipulation** Vulnerabilities created when miners purposely manipulate the exact time their blocks are mined (a variable that can be used inside smart contracts)
9. **Short Address Attack** Vulnerabilities created when poorly coded clients encode arguments incorrectly before including them in transaction
10. **Unknown Unknowns** Everything else

The mapping utilized between the tools in SmartBugs and the categories shown above is shown in figure A.1, available in appendix A of this paper. Using these categories we can follow the two rules presented above and join our bug reports together.

4.4 From metadata to solutions CSV

The next step in the data processing pipeline is to prepare a CSV file containing the solutions to our experiment. Since our machine learning model uses supervised learning methods, it is important to prepare data where we know the classification results beforehand. This way we can train a better model.

To obtain said data, we need to look at each dataset used, and parse the location and category of each bug to a solutions CSV file that will later be joined with the bug reports CSV file prepared in the previous section. In this section we will discuss the steps taken to prepare said solutions CSV file, while also suggesting a standard that could be used to declare bugs in a bug dataset.

Since each dataset used in this project had its own unique way to declare the bugs it carries, a unique parser had to be developed for each dataset to parse their bug metadata to our desired CSV file.

Another issue concerns categories. To match up with the DASP10 categories described in the last section, we have also mapped a DASP10 category to each bug of each dataset. This is possible since every bug on the datasets was created to purposely illustrate a certain bug type. The details of each mapping will be described in each dataset's subsection.

In short, there are two pieces of information we need to extract from every bug's metadata:

1. The location (file and line of code) where said bug appears
2. The category of the bug in question

The following subsections will describe how this extraction is done for each of the datasets used.

4.4.1 SmartBugs sb^{curated} dataset

The SmartBugs dataset was fairly easy to parse. Since the bugs had already been organized by DASP10 category, there was no need to map between SmartBugs category division and DASP10 category division. After all, this dataset was created by the developers of the SmartBugs framework this work extends, having already been prepared with DASP10 categories in mind.

As for the location of the bug, this information had been left in the form of comments in the source code of the files from which the bugs originated.

There was also an additional JSON file containing all bugs in the dataset, including their location and category. In practice it was this file that was read. To have a standard from which to judge all datasets, this information was parsed to our solutions CSV file. The format this file has, as well as the standard we wish to start, will be explained in the last subsection of this section.

4.4.2 SolidiFI

As for the SolidiFI dataset, its metadata had been left in the form of CSV files (one for each source code program). Said metadata contains only the location of the bug in the source file. To extract the location of the bug, it sufficed to simply read it from the corresponding file's CSV auxiliary file.

To get the bug's category, we had to map the folder category of this dataset (which separated each bug by it's own category) to the DASP10 category system. This is necessary for the future step of

SolidiFI Category	DASP10 Category
Overflow-Underflow	Arithmetic
Re-entrancy	Reentrancy
Timestamp-Dependency	Time Manipulation
TOD (Time Order Dependency)	Front-Running
tx.origin	Access Control
Unchecked-Send	Unchecked Return Values For Low Level Calls
Unhanded-Exceptions	Unchecked Return Values For Low Level Calls

Table 4.1: Vulnerabilities mapping between SolidiFI and DASP10

joining the bug reports CSV file and solutions CSV file. As mentioned in the previous section, we consider two bug reports to concern the same bug if they are reported on the same line, and are of the same category. This second requirement would be impossible to fulfill if there was no common ground to compare categories from each dataset.

The mapping we followed is presented in table 4.1.

4.4.3 HuangGai

Lastly we have the HuangGai dataset. In this case a CSV file accompanied the source file (one CSV file for each source file), which provided the lines of code where the bugs occurs.

While the idea is similar to the SolidiFI's dataset described above, the way the CSV file itself is presented is different. For this reason, a separate parser had to be developed to extract HuangGai's metadata from their CSV files.

As for the category, it was mapped from the folder category of the dataset to the DASP10 category system presented above, in a similar matter to the SolidiFI dataset.

The mapping utilized for HuangGai's categories is as presented in table 4.2.

4.4.4 Solutions CSV file organization and standards

After extracting bug information from all datasets, we can now populate the solutions CSV file described at the beginning of the section. To fully utilise the advantages of consensus a new format was developed. This format represents a CSV file where each row has the following columns:

1. Contract name: The contract name of the contract in question, including its relative path from the main folder. This way we can differentiate between multiple contracts with the same name, but on different folders.

2. Lines: The next piece of information is the lines where the bug is located on. To allow more versatility in expressing bugs that can be accepted to be reported on multiple lines, there exist

HuangGai Category	DASP10 Category
contractAffectedByMiners	Bad Randomness
DosByComplexFallback	Denial of Service
forcedToReceiveEthers	Arithmetic
hashWithMulVarLenArg	Unchecked Return Values For Low Level Calls
integerOverflow	Arithmetic
lockedEther	Denial of Service
nonStandarNaming	Unchecked Return Values For Low Level Calls
NonpublicVarAccessdByPublicFunc	Access Control
preSentEther	Arithmetic
publicFuncToExternal	Access Control
reentrancy	Reentrancy
shortAddressAttack	Short Address Attack
specifyFuncVarAnyType	Unchecked Return Values For Low Level Calls
suicideContract	Access Control
transactionOrderDependancy	Front-Running
txOriginForAuthentication	Access Control
unhandledException	Unchecked Return Values For Low Level Calls
uninitializedLocalVariables	Unchecked Return Values For Low Level Calls
unlimitedCompilerVersions	Unchecked Return Values For Low Level Calls
wastefulContracts	Access Control

Table 4.2: Vulnerabilities mapping between HuangGai and DASP10

multiple ways to declare in what lines a bug is located. First, a simple number can be used to express a single line, i.e. the row ”contract.sol,100,access_control” describes that there is a bug in contract contract.sol, on line 100 whose category is access_control. Another way to express lines is using a ”-” operator. A lines value of 100-120 implies that a bug can exists between line 100 and line 120. Lastly, our format also accepts the ”/” operator. A line value of 100/105 means that a bug exists on either line 100 or line 105. It is also important to notice that multiple operators can be used simultaneously. For example a lines value of 100-120/132 means that a bug exists either in between lines 100 and 120 or in line 132. This way of expressing lines can be used to better describe bugs that can be reported on multiple lines

3. Category: The last piece of information described in each row of our solutions CSV file is the category of the bug in question. As mentioned above this, category must be one of the previously described DASP10 categories. Similarly to lines, there exist multiple different ways to express which categories to accept a bug report as. The simplest way is to simply write a single category in the category field. The row ”contract.sol,100,access_control”, as mentioned above, describes that there is a bug in contract contract.sol, on line 100 whose category is access_control. Another way would be to use the ”/” operator. A category value of arithmetic/denial_service would mean that a certain bug is either of the arithmetic or of the denial_service category. Since there is no continuity between different categories as exists between numbers, there is no ”-” operator for the

category value. There is however an ”ANY” value that matches to any category. As for the lines value, this way of expressing the category of a bug is more expressive, allowing us to accept bug reports of bugs whose categories are ambiguous.

In most cases, when parsing from the datasets used to our CSV format, there is a single line and single category reported, making little use of the ”-” and ”/” operators. That isn't to say however, that there is no use to them. The datasets we used simply did not have that information available, so it would have required manual analysis of all 1,657 faulty smart contracts in our final dataset to add this information to their metadata. There are plenty of use cases for the proposed format.

Take this bad randomness vulnerability for example:

```
1 function deal(address player) internal returns (uint8) {
2     uint b = block.number;
3     uint timestamp = block.timestamp;
4     return uint8(uint256(keccak256(block.blockhash(b), player, timestamp)) / 52);
5 }
```

Variables with bad randomness were declared in lines 2 and 3. They were however, only used in line 4, which is where the vulnerability applies. Where should this bug be reported? In lines 2 and 3, or line 4? Both approaches seem reasonable and it is up to the dataset's author to decide. If the author marks lines 2 and 3, he is marking 2 different bugs, and a report on line 4 would be considered false. If the author marks line 4, then only one bug is declared and any reports on lines 2 or 3 will be considered false. Using our format however, the author could write 2-3/4 on the lines value and accept both cases.

It is also important to note that different analysis tools take different approaches and may report the bug in any of the two present cases. Only using our format could both reports be counted as true positives.

Because we do believe our format to be more expressive than any other we have come across, we would like to suggest to our readers to use it as well in their bug datasets. This level of flexibility will allow for better descriptions, and therefore analysis, of bugs.

Based on the study of multiple bug datasets available, table 4.3 describes what methods are currently in use to present the metadata related to bugs in nine datasets.

As seen in table 4.3, most datasets already use CSV files for their metadata (usually one per source file), so a transition to a different format shouldn't be overly difficult. Furthermore, we would suggest for it to be a CSV file for each source code file (as it is already common practice) as well a single CSV file containing all bugs in the dataset to facilitate works such as ours that look at the dataset in its entirety.

Lastly, it is important to notice that multiple ways to select metadata can be used simultaneously, such as the case for the SolidiFI dataset, where both CSV files and comments are used. So there is no need to exclusively use the standard suggested while disregarding previous approaches.

Dataset	Bug Language	Method
SmartBugs [1]	solidity	JSON and Comments
SolidiFI [29]	solidity	CSV and Comments
Jiuzhou [32]	solidity	Json
Huang Gai [30]	solidity	Txt and Comments
Defects4JS [33]	java	CSV
BugsJS [34]	java	CSV
The Bug Prediction Dataset [35]	java	CSV
Bug-Fix Dataset [36]	java	CSV
Unified Bug Dataset [37]	java	CSV

Table 4.3: Metadata method per bug dataset

4.5 Final CSV

Returning to our own data processing pipeline, after preparing both a bug reports CSV file and a solutions CSV file, the last step is to join them both, obtaining a single CSV file with all bugs and all bug reports. The objective is to classify each bug report in the bug reports CSV file as either a true or a false positive based on the bug information available in the solutions CSV file.

The final CSV file will have all the columns of the bug reports CSV (the rule_id and level found by each tool) as well as a final column describing if this report refers to a true or a false positive.

A report from the bug reports CSV file is considered a true positive when:

1. The bug report was reported on the same file as a bug in the solutions CSV file
2. The bug report's lines matches, or are at least a subset of, the same bug's line in the solutions CSV file. Do note that the solutions CSV file can accept multiple lines as the location of the bug as per described in the previous section.
3. The report's category matches at least one of the categories of the bug in the previous line

Should all these conditions prove true, then the bug report is considered to be a true bug report, otherwise it is considered to be a false positive bug report. At the end of file we put all the true positives not caught by the tools, making sure all bugs and bugs reports are properly represented in the final CSV file.

This final CSV file contains all bug reports and all bugs properly connected, so the machine learning model can learn which reports represent true positives, and which reports represent false positives.

5

Machine Learning Model

Contents

5.1 Features 37

5.2 Data preparation 37

5.3 Machine learning models used 39

This chapter describes the machine learning models utilized in this project. The goal of said machine learning models is to classify bug reports provided by SmartBugs as either true or false positives. We will then re-organize the list of bug reports, putting first those classified as true positives, and then those classified as false positives. This way we can achieve a prioritized list of reports, allowing developers to focus their time at the top of the list, where bug reports are more likely to be correctly identified.

The following sections describe the important aspects of the machine learning models.

5.1 Features

The data provided to the machine learning model is the final CSV file described at the end of the previous chapter. To re-iterate, it is a CSV file where each row represents a bug, and columns represent the analysis provided by each tool on said bug. Each tool is represented in two columns, one for the rule_id of the rule broken that originated the bug report, and one for level the tool classified this bug as (either warning or error). Should a tool not have reported this bug, than these columns are left empty. At the end there is a final column classifying the bug report as either true or false.

The last column will be used to help train the model with already classified data, so it is not counted as one of the features. The other columns (excluding the id column), are the features provided to our machine learning model. This is, the features provided to the model are the results of the analysis of each tool for the same bug, as per determined in the previous chapter. The model will have access to the rule_id and level that results from the analysis of each tool on the same bug.

The consequence of this feature selection is information about consensus. Our machine learning model has information about the same bug, from all tools' points of view and can use that information to better classify bug reports as either true or false positives.

Should a specific tool (or conjunction of tools) prove to be more reliable than others at classifying a certain type of bug report, then our machine learning model can learn this information and rely more on said tools. Should a specific tool (or conjunction of tools) prove to be less reliable than others at classifying a certain type of bug report, the opposite can apply.

This is the benefit of consensus between multiple analysis tools and this is what we expect our machine learning model to learn with.

5.2 Data preparation

The data presented to machine learning model is that which is presented in the previous chapter, but besides the data preparation there mentioned, there is still two matters that must still be prepared, further described in the following subsections.

5.2.1 Data division

The first is related to data division between training and testing sets. Here, a balance must be struck between increasing the size of the training set and increasing the size of the testing set.

On one hand, increasing the size of the training dataset will give more information for the machine learning model to train with. Having more information could allow for better trained models, that can better predict the classification of our bug reports. Increasing our model's accuracy is of crucial importance, so the bigger the training set, the better.

On the other hand, increasing the size of the testing dataset allows for a better evaluation of the machine learning model. A bigger testing set means there is more data to test the model with, and will consequently allow us to test the model under a wider range of scenarios. Just like the data training set, the bigger the testing set, the better.

To strike a balance between these two conditions we have chosen to use 80% of our total data as the training set data, and the remaining 20% as the testing set data. This way we can spend most of our data on training, while retaining a significant amount for testing. The advantage of separating our data as such, is that we can test the model with data it has never seen before, avoiding any biases in the model itself for the data it is tested with.

Lastly, it is important to note that the data for each set is randomized. This is, 20% randomly chosen data will be used for testing, and the remaining 80% will be used for training. The order of each dataset is also randomized. This way we can avoid any unwanted correlation between training and testing sets.

5.2.2 Data balancing

The second matter that must be addressed in the context of data preparation is data balancing. Due to the nature of the data in this thesis, we can expect the majority of bug reports used as data to refer to false positive bug reports. From the data used, 166,348 out of 192,073 reports concern false positives, while only 25,725 out of 192,073 reports concern true positives.

The implication of this imbalance is great. As the vast majority of bug reports concern false positives, a machine learning model that always predicts the bug report as false will be correct the vast majority of the time. As 86% of our reports concern false positives, a machine learning model that always classifies reports as false positives will have an accuracy of 86%, which might seem high but would in fact be useless. To avoid this issue, developers can look at more metrics besides accuracy that will be described and discussed in the next chapter, and developers can also balance their data to minimize or avoid the issue altogether.

There exist two major approaches to achieve data balancing:

- Undersampling techniques

- Oversampling techniques

Undersampling techniques reduce the majority class in the imbalanced dataset until the dataset is balanced. In our case, since we have 166,348 false positive bug reports and 25,725 true positive bug reports, we would need to eliminated 166,348 - 25,725 = 140,623 false positive bug reports to reach a balance. There exist multiple different algorithms to achieve this, but the common point between all undersampling techniques is that data will be lost to achieve a balance. The disadvantage of these techniques is that we must effectively ignore the vast majority of the data we have acquired.

Oversampling techniques are the opposite, they increase the minority class in the imbalanced dataset until the dataset is balanced. In our case, since we have 166,348 false positive bug reports and 25,725 true positive bug reports, we would need to create 166,348 - 25,725 = 140,623 true positive bug reports to reach a balance. There exist multiple different algorithms to achieve this, but the common point between all oversampling techniques is that data will be created to achieve a balance. The disadvantage of creating data is that the model will be trained with data that was algorithmically created to be similar to previously classified data, having the potential of being incorrectly classified, or too being similar to the original data, creating biases in the model.

For this work we have chosen to use oversampling techniques, as we have concluded that the cost of throwing away 73% (140,623 eliminated false positive reports out of 192,073 total reports) of our dataset via undersampling techniques to be too high.

The specific algorithm chosen was the SMOTE (Synthetic Minority Oversampling Technique) algorithm. SMOTE creates synthetic data points of the minority class by declaring data points on the vector line between an original data point of the minority class and their nearest neighbors of the same class. This way it creates data points that are similar, but not equal, to the data points previously existing, increasing the number of data points in the minority class while maintaining high probability of a correct classification.

5.3 Machine learning models used

In this thesis we have chosen to utilize three types of machine learning models:

- Classification models
- Regression models
- Ensemble models

The following subsections will describe the difference between the three types, as well as some details of their implementation, namely how classification is set and how prioritization is done.

5.3.1 Classification models

Classification models are in one sense, the simplest. They simply classify each report as either a true positive or a false positive. There is no need for any further work when it comes to classification. Examples of this models are decision tree classifiers and KNeighboors classifiers.

As for prioritization, since our results are classified into true or false positives, we can only sort our bug reports list by presenting those with a true positive value first, and then those with a false positive value after. As there is no distinction between multiple reports with the same value, there is no way to further organize the list after the previous step has been completed. This is, there is no way to sort the list between two reports considered as true positives. The same applies for false positives.

As both the classification and the prioritized list will be used to calculate the evaluation metrics described in the next chapter, there is an inherited level of randomness to the metrics related to the order of the prioritization list. As sorting between multiple reports with the same classification cannot be done, their order will depend on the order that the reports were presented in. As such, depending of the random sorting of the data preparation section 5.2, the order of the prioritized list can be slightly different, and therefore slightly affect the results on each run. As machine learning models already have inherent differences between multiple iterations on the same data, this small randomness did not prove to be noticeable.

5.3.2 Regression models

Regression models such Logistic Regression and KNeighboors Regressor work differently from classifier models. Instead of providing a final result of either true or false positive, they instead provide a probability value of the report being a true bug.

For metrics that require binary classification of either true or false positives, we consider a prediction to refer to a true positive when it's probability is above 50% and consider it to refer to a false positive when it's probability is below 50%.

As for prioritization, regression models allow for a more flexible approach when compared to classification models. Regression models give continuous (instead of discrete) values. As such, a much more detailed sorting can be achieved by sorting the reports list by the probability value provided by the machine learning model. Using probabilities, not all true positives are the same, which allows for better prioritization.

5.3.3 Ensemble models

Ensemble models are machine learning models created using a combination of other machine learning models. They can be either classifiers or regressors, taking classification models or regression models

for their creation accordingly.

Some of this models, such as the Extra Trees Classifier, have predetermined models as part of their ensemble algorithms (in this case decision trees), while other models, such as the Voting Regressor, have the user choose which estimator models to use as part of their learning mechanism. For models where this is the case the better performing models of each category (including other ensemble models that do not require the user to choose models by hand) where chosen as the input for the ensemble models.

The estimator models we used to train our ensemble models were the following:

- Decision Tree
- K Neighbors
- Ada Boost
- Extra Trees
- Gradient Boosting
- Random Forest
- Hist Gradient Boosting

All of these models have both classifier and regressor versions, where classifier versions were used for ensemble classifiers, and regressor versions were used for ensemble regressors.

For the ensemble classifiers we processed their data as per the classification models section, and for the ensemble regressors we processed their data as per the regression models section.

6

Evaluation

Contents

6.1 Evaluation metrics 45

6.2 Classification models 48

6.3 Regression models 51

6.4 Ensemble models 53

This chapter discusses the evaluation of the machine learning models presented in the previous chapter. The first section of this chapter presents the metrics utilized for the evaluation procedure, while the following sections provide detailed information on the performance of each model.

6.1 Evaluation metrics

This section of this chapter describes the metrics utilized to evaluate the varied machine learning models. It is further divided into three subsections, discussing the different types of metrics.

6.1.1 Classification Metrics

The first set of metrics are the classification metrics. They require discrete classifications to be calculated, and where those are not provided (such as regression models that provide continuous output) they are treated as presented in the previous chapter. These metrics are:

1. Accuracy
2. Precision
3. Recall
4. F1-Score

Accuracy is a metric that quantifies the amount of correctly identified samples, divided by the total number of samples. In simpler terms, it measures the accuracy of the model, this is the probability of a sample being correctly classified. As mentioned in the previous chapter, highly imbalanced datasets can easily achieve a high accuracy by using simple blind classification to the majority class, so further metrics are also needed. The worst value for this metric is 0, and the best value is 1. The formula for this metric is:

$$Accuracy = \frac{True\ positives + True\ negatives}{True\ positives + False\ positives + True\ negatives + False\ negatives}$$

Please do note that true and false positives in this section refer to the final classification of the machine learning model. This is, true positives are bug reports considered as positive by the model and that actually correspond to bugs according to the dataset's classification. False positives refer to bug reports the machine learning model considered as positives but that in fact do not correspond to bugs according to the dataset's classification. Likewise true negatives are reports the model considers as false and that do not correspond to bugs, and false negatives are reports the model considers as false but that do in fact correspond to actual bugs.

The next metric used is recall, used to evaluate the machine learning model's ability to find all positive samples. This is a great counter-metric to accuracy, as a model that uses blind classification to have good accuracy in an imbalance dataset will instead have the lowest value of recall available. The worst value for this metric is 0, and the best value is 1. The formula for this metric is as follows:

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

The next metric utilized is precision. Precision measures the model's ability to not misclassify a positive sample as negative. Same as recall, it is also a great counter-metric to accuracy for the same reasons. The worst value for this metric is 0, and the best value is 1. The formula for this metric is as follows:

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

The last metric in this section is F1-Score. F1-score measures the harmony between precision and recall. This is a useful metric, both precision and recall must be high in order to have an effective machine learning model. The worst value for this metric is 0, and the best value is 1. The formula for this metric is as follows:

$$F1 - Score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

All these metrics will be applied to each machine learning model, both in its training and testing datasets. The reason these metrics are applied to both the training and testing sets is to check if the model suffers from overfitting. Overfitting occurs when a statistical model fits exactly against its training data, but then performs poorly on never seen before testing data. By applying these metrics to both training and testing sets, we can see if there is a big difference between the two. Should that be the case, than the machine learning model shows signs of overfitting. After checking these metrics, we have concluded that our machine learning model does not suffer from overfitting, as the results of these metrics are very similar in both training and testing sets. For this reason, we have abbreviated the training metrics from the tables presented in the next subsections, but full tables are presented in appendix B of this paper.

6.1.2 APBD

APBD stands for Average Percentage of Bugs Detected. It is a metric presented in RVPrio's report [4], discussed in Chapter 3. This metric takes a prioritized list of bug reports (how this is list is calculated is further discussed in the previous chapter) and computes the weighted average of the percentage of true

bugs revealed over the course of said list. The worst value for this metric is 0, and the best value is 1. The definition of APBD for a set of n bug reports R with m bugs B existing in R is:

$$APBD = 1 - \frac{RB_1 + RB_2 + ... + RB_m}{nm} + \frac{1}{2n}$$

RB1 represents the first true bug present in the list of reports R, RB2 the second bug, and RBm the last bug present in set R.

The higher the APBD value, the more true positive bug reports are presented at the beginning of the list, which is good. In counterpart, the lower the APBD value the fewer true positive bug reports are presented at the beginning of the list, which is bad. This is a useful metric, because it measure the prioritization of the sorted list, rather than the model itself, which is our final output.

In practice, APBD is our most valued metric, as it is the only metric presented so far the measures the prioritized list presented to the user directly. It was to achieve this prioritized list that the machine learning model was developed to begin with.

6.1.3 Prioritization graph

The last important metric is the prioritization graph. The same as APBD, this metric works directly over the prioritized list that results from our machine learning model's predictions.

It fundamentally measures the same thing, but presents it in form of a graph instead of a number. Figure 6.1 shows an example of a prioritization graph.

The horizontal axis in this graph represents the number of reports analysed, starting from zero and ending at the end of the prioritized list. The vertical axis represents the percentage of true bugs found until that point in the horizontal axis, and goes from 0 to 100%.

The graph starts at 0,0 as no true bugs (0% on vertical axis) have been found after analysing 0 reports (0 on the horizontal axis). The graph must also end at end_of_list,100% as all true bugs (100% on vertical axis) have been found after analysing all bug reports in the prioritized list (end_of_list on the horizontal axis).

The green line in the example represents the perfect prioritized list, where all true bugs are at the beginning of the list. Since every report at the beginning of the list is a true positive, the vertical axis quickly grows to 100% until the all true bugs are found, and then remains constant at 100% as only false positives are left. The green line represents the ideal prioritization, so the closer our models follows the green line the better.

The red line represents a non-prioritized list, where true positives bug reports have been equally spaced throughout the list. This is, to find 50% of all true positives bug reports we would have to look at 50% of all reports, and to find 75% of all true positive bug reports we would also have to look at 75% of

Figure 6.1: Example of a prioritization graph

all available reports. Basically, the red line represents a list that has not been prioritized at all, and what would happen if no prioritization is applied. The closer our model follows the red line the worst. If the model performs even worst than the red line (that is to say it is constantly bellow the red line) then the model has failed as its prioritization results are even worse than not prioritizing at all.

Lastly, the blue line represents the prioritization list resulting from the analysis of our machine learning model. As mention above, the closer this line follows the green line the better, the closer it follows the red line the worst. In the example shown, the line closely, but not perfectly, follows the green line showing a rather good performance. Further discussion of the result of these metrics will be presented in the following sections.

6.2 Classification models

The classification models tested, as well as their results in the various metrics, can be seen in table 6.1

An analysis of the table justifies the selection of metrics presented above. Taking the Passive Aggressive Classifier model as an example, it has a high accuracy of 0.833, but a low recall of just 0.161. This likely means that this model is heavily biased towards marking any report as false, only achieving a high accuracy due to the imbalanced dataset and failing to find most true positive samples in the dataset. This phenomenon is further explained above and shows the need to look at all metrics as a whole. Since the objective of this paper is prioritization, it is important to take all the metrics into consideration, and

Model	Accuracy	F1-Score	Recall	Precision	APBD
Decision Tree Classifier	0.985	0.947	0.987	0.909	0.921
Perceptron	0.836	0.606	0.941	0.447	0.830
Passive Aggressive Classifier	0.833	0.206	0.161	0.285	0.541
Ridge Classifier Cross Validation	0.822	0.594	0.971	0.428	0.835
SGD Classifier	0.837	0.613	0.961	0.450	0.838
KNeighbors Classifier	0.967	0.891	0.999	0.803	0.916

Table 6.1: Testing metrics for each classification model tested

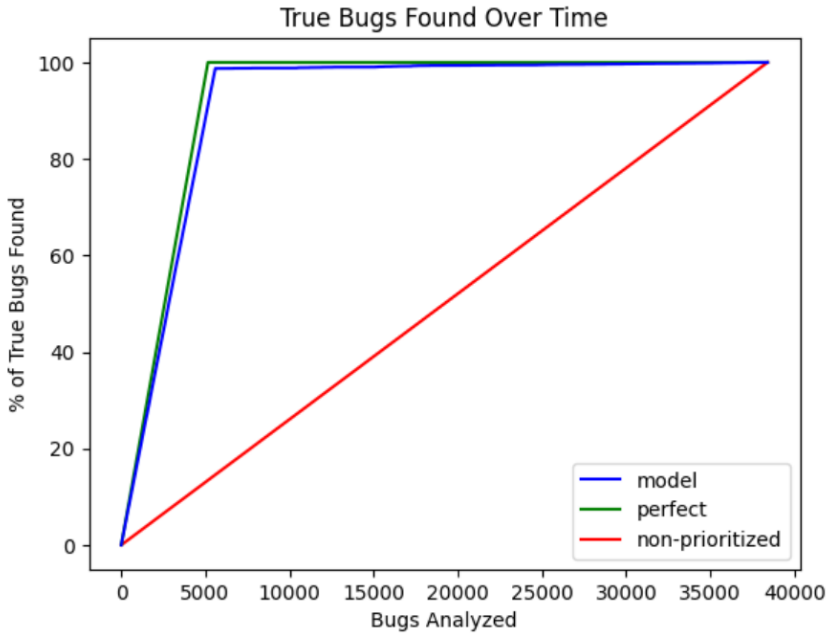

Figure 6.2: Decision tree classifier model's prioritization graph

we can thus consider the Passive Aggressive Classifier model to be a poor fit for our context.

Taking this into consideration, our results point to the Decision Tree Classifier and the KNeighbors Classifier as the most suitable machine learning classification models for our environment. This is because these are the models that have consistently higher metric scores, namely the APBD metric which we consider to be the most important. Figure 6.2 presents Decision Tree Classifier's prioritization graph, while figure 6.3 presents the same graph for the KNeighbors Classifier model.

Both graphs are similar as these two models have similar metrics. As can be seen, and in both cases, the blue line representing the model closely resembles the perfect green line of a perfect system where all bugs are correctly prioritized. As was explained in the previous section, this means our models can almost perfectly prioritize the vast majority of true bugs towards the beginning of the prioritization list, with only a few being incorrectly put at the end of the list.

The difference between the two graphs can be explained by the metrics of their respective models. The Decision Tree Classifier has better accuracy and precision and therefore more samples are correctly

Figure 6.3: KNeighbors classifier model's prioritization graph

Approach	10%	25%	50%	75%
Non-prioritized Decision Tree Classifier	10% 67%	25% 98%	50% 99%	75% 99%
Increase	6.7x	3.92x	1.98x	1.32x

Table 6.2: Time table for the Decision Tree Classifier model

classified when compared to the KNeighbors Classifier model. This can be seen by how closely its blue line resembles the green line in figure 6.2. Alternatively, the KNeighbors Classifier model has a near perfect recall and therefore very few true positives samples are incorrectly classified by this model. This can be in figure 6.3 by how the blue line overlaps with the green line after a while. This means that the Decision Tree Classifier provides better prioritization towards the beginning of the list, while the KNeighbors classifier model can find all true positive samples sooner. Prioritizing different metrics results in different advantages for each model.

Perhaps the best way to express the usefulness of this prioritization is though table 6.2. The table expresses that after analyzing 10% of bug reports in a non-prioritized list we would have found only 10% of true bugs, but analysing the same 10% of reports of a list prioritized by a Decision Tree Classifier, we would instead find 67 % of all true bugs, a 6.7x increase over a non-prioritized list. This represents a major increase in utility from the point of view of a developer that has too look through so many bug reports. By simply analysing 25% of bug reports a developer would have already found 98% of all bugs in the dataset.

Model	Accuracy	F1-Score	Recall	Precision	APBD
Decision Tree Regressor	0.985	0.947	0.987	0.909	0.932
Logistic Regressor Cross-Validation	0.868	0.655	0.935	0.503	0.884
KNeighbors Regressor	0.967	0.891	0.999	0.803	0.930
Linear Regressor	0.822	0.594	0.971	0.428	0.882
SGD Regressor	0.753	0.516	0.980	0.350	0.899
Ridge Cross-Validation	0.882	0.594	0.971	0.428	0.882
ARD Regressor	0.822	0.594	0.971	0.428	0.901
Bayesian Ridge	0.822	0.594	0.971	0.428	0.882

Table 6.3: Testing metrics for each regression model tested

Returning to the graphs, a clear point of transition between two different line gradients can be seen in figure 6.2 on the blue line, at around the 95% mark on the vertical axis. This is a consequence of prioritizing reports based only on the binary classification of true and false. As explained in the previous chapter, classification models cannot differentiate between two different reports that have been both marked as false bug reports. As a consequence, it cannot make any further prioritization besides presenting the bug reports classified as true first, and those classified as false second. Since this model shows high values in both accuracy and precision, it is very effective at classifying bug reports. This can be seen in the graph by the almost perfectly stable incline of the blue line until said 95% mark. However, by fault of being a classifier and not a regressor, it cannot minimize the errors of any missed classifications and therefore have an equally stable, but worse, incline after the mark. The regression models presented bellow will have a different behaviour.

6.3 Regression models

The regression models tested, as well as their results in the various metrics, are presented on table 6.3.

Similarly to the classification models discussed in the previous section, it is once again the Decision Tree Regressor and KNeighbors Regressor that perform the best in our environment. Also similarly to the previous sections there were many models, such as the Linear Regressor model, that perform well on a specific category (namely recall) but them perform poorly in others, leading to an overall bad performance. As expressed above, we are interested most in models that have consistently high scores in all presented metrics, so the Linear Regressor is a poor fit for our solution.

One thing to notice is that, when comparing this table to the previous one, APBD values are overall higher with regression models over classification ones. This is due to the higher versatility in prioritization that comes from the continuous output of regression models. This point is further shown in the prioritization graphs for these two models presented in figure 6.4 and 6.5 respectively.

When looking at figure 6.4, one can see a much smother curve around the 95% mark that was

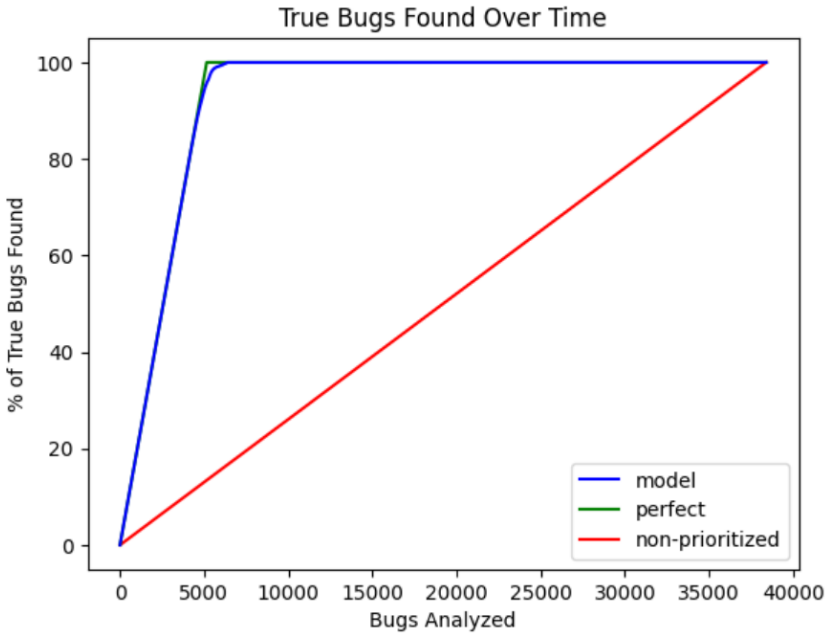

Figure 6.4: Decision tree Regressor model's prioritization graph

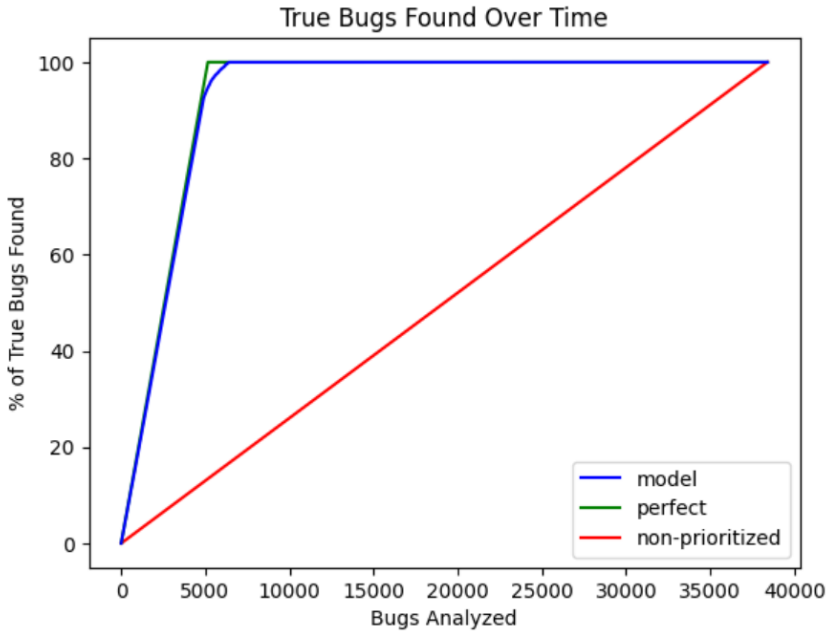

Figure 6.5: KNeighbors Regressor model's prioritization graph

Approach	10%	25%	50%	75%
Non-prioritized	10%	25%	50%	75%
Decision Tree Regressor	75%	99%	100%	100%
Increase	7.5x	3.96x	2x	1.33x

Table 6.4: Time table for the Decision Tree Regressor model

discussed in the previous section. This is because the Decision Tree Regressor has the potential to recover quickly from any missed classifications. Even when it makes an incorrect classification, due to the continuous output, it can still differentiate from the multiple reports considered to be false positives, allowing the prioritization algorithm to take this into consideration. The result is a much more consistent prioritization list. Table 6.4 shows the same time table discussed in the previous section for the Decision Tree Regressor model.

This table shows even better results than the last. By simply analysing 10% of bug reports with a list prioritized by a Decision Tree Regressor model, a developer could find 75% of all bugs in the dataset. This represents a 7.5x increase over a non-prioritized list, where the same developer could only hope to find 10% of true bugs. Likewise, by looking at only the first 25% of bug reports, the developer would be able to find 99% of all bugs in the dataset by using the prioritize list, and even the full 100% before the 50% mark of analysed reports. This is the great benefit of prioritized bug report lists over their non-prioritized counterparts.

6.4 Ensemble models

Lastly, the ensemble models tested, as well as their results in the various metrics, can be seen on table 6.5.

The table presents better results for both ensemble classifiers and ensemble regressors when compared their simpler versions. It is also noticeable that results are consistent throughout most models, whereas the previous sections had a considerable amount of models with poor results. Results are so consistent in fact, that some of the models have exactly similar results in many metrics (at least as far as third decimal case precision). This seems to support the related-work's suspicion that ensemble models provide better results than simpler models in the context of bug prioritization.

Our overall best models are the Extra Trees models and the Random Forest models. These two models are extremely similar, being in fact two different variants of the same algorithm. Since they have exactly similar metrics, we have chosen to select the Random Forest models as the example of choice for the rest of this section. The reason we chose the Random Forest models over the Extra Trees models is because the Random Forest algorithm chooses deterministic split points where the Extra

Model	Accuracy	F1-Score	Recall	Precision	APBD
Ada Boost Classifier	0.952	0.874	0.991	0.741	0.906
Extra Trees Classifier	0.985	0.947	0.987	0.909	0.921
Gradient Boosting Classifier	0.964	0.882	0.999	0.789	0.915
Random Forest Classifier	0.985	0.947	0.987	0.909	0.921
Hist Gradient Boosting Classifier	0.985	0.946	0.987	0.909	0.921
Stacking Classifier	0.985	0.947	0.987	0.909	0.921
Voting Classifier	0.985	0.947	0.987	0.909	0.921
Ada Boost Regressor	0.897	0.710	0.942	0.570	0.884
Extra Trees Regressor	0.985	0.947	0.987	0.909	0.932
Gradient Boosting Regressor	0.964	0.883	0.999	0.791	0.931
Random Forest Regressor	0.985	0.947	0.987	0.909	0.932
Hist Gradient Boosting Regressor	0.984	0.946	0.987	0.908	0.932
Stacking Regressor	0.967	0.891	0.998	0.803	0.929
Voting Regressor	0.967	0.891	0.999	0.804	0.932

Table 6.5: Testing metrics for each ensemble model tested

Approach	10%	25%	50%	75%
Non-prioritized	10%	25%	50%	75%
Random Forest Regressor	75%	100%	100%	100%
Increase	7.5x	4x	2x	1.33x

Table 6.6: Time table for the Random Forest Regressor model

Trees algorithms chooses the best split point between multiple randomly tested splits. This results in the Extra Trees algorithm running faster when presented with a large number of features, but at the cost of potentially growing more inaccurate. While that inaccuracy has not made itself felt in this project, it could perhaps start to show as more and more tools are added to SmartBugs which results in more and more feature being available to the machine learning model. Taking this into consideration, we have chosen to follow the slower but safer algorithm, and thus select the Random Forest algorithm as the most suitable model for this project.

Figure 6.6 shows the prioritization graph for the Random Forest Classifier model, and figure 6.7 shows the same graph but for the Random Forest Regressor model instead.

As per the findings of the previous sections, we once again conclude that regressor models are more suitable for our environment when compared to classification models. This makes us wonder why has the related work instead chosen the latter, choosing to classify bug reports into the very low, low, medium, high and very high probability categories. The comparison between figure 6.6 and figure 6.7 implies that regression models leave better results in this context.

Table 6.6 shows the time table for the Random Forest Regressor model.

As can be seen in the table, merely analysing 10% of results would lead to a developer finding 75% of all true bugs in the dataset, having actually found all 100% of true bugs before the 25% mark on

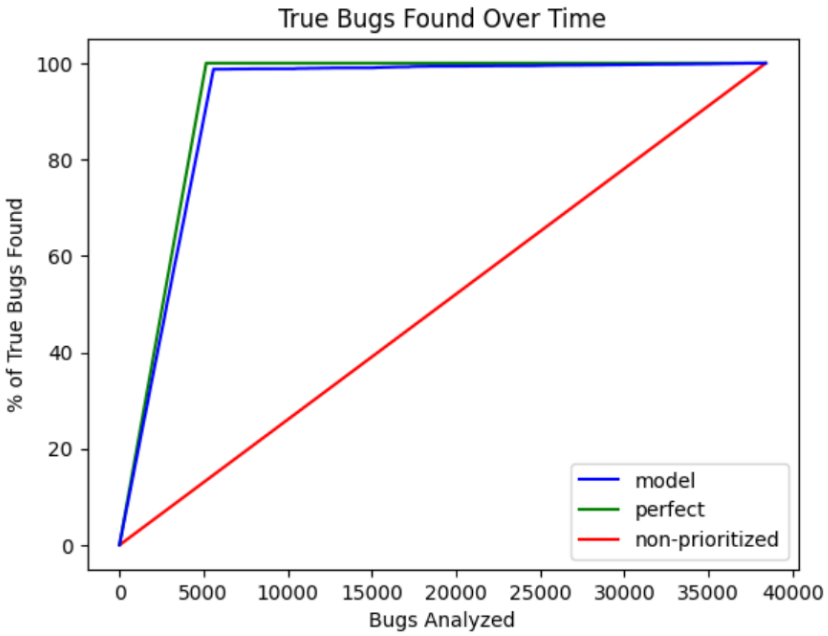

Figure 6.6: Random Forest Classifier model's prioritization graph

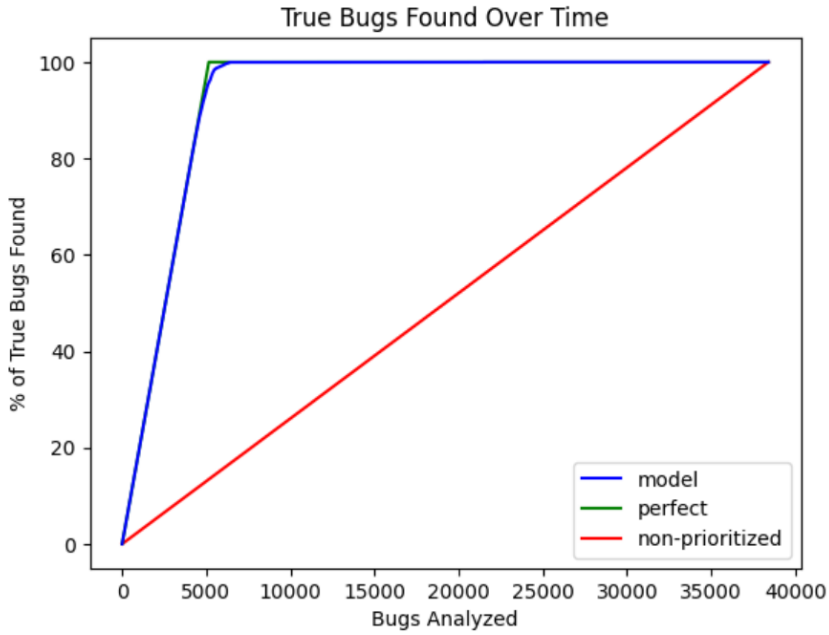

Figure 6.7: Random Forest Regressor model's prioritization graph

the horizontal axis. In fact, all true bugs of the dataset would have been found after analysing only 21% of bug reports, saving almost 80% of the developer's time should he had analysed the whole list instead. This once again proves that analysing a prioritized list of bugs is multiple times more efficient than analysing the same list without prioritization.

7

Conclusion

Contents

7.1 Contributions 59
7.2 Answer to research questions 59
7.3 Threats to validity 60
7.4 Future work 61

This project extends the SmartBugs framework with a bug prioritization mechanism capable of ranking true positive bug reports over false positive bug reports. To implement this mechanism we first start by preparing and processing data. We prepare a new bug dataset composed of multiple other bug datasets in a newly proposed format, we join multiple SARIF reports resulting from SmartBugs' executions together, we extract bug information from our prepared dataset and we finally join the reports extracted from SmartBugs' execution and the dataset's metadata together into a final CSV data file.

We then feed this information to a machine learning model, taking extra care to discuss data division and balancing before describing the implementation details of various types of machine learning models.

After the models are implemented we then evaluate them based on a variety of discussed metrics and compare their results to see which one is the most suitable for this environment.

7.1 Contributions

The contributions of this thesis start with the new bug dataset created for the purposes of this project as well as the format it is in. The proposed format is more flexible than any other we have come across and it is most suitable for the purpose bug prioritization through consensus of multiple analysis tools. We hope to see it implemented in more bug datasets in the future.

The implementation of a machine learning model based on consensus from SARIF only features resulting from the analysis of multiple analysis tools on the same contract, contributes to field to bug prioritization. To our knowledge there are no other extensive works on this field, and our project shows that consensus can be a valuable feature for the purpose of bug prioritization.

The evaluation of multiple machine learning models is also a contribution. The best results from our experiments correspond to the Random Forest Regressor model. We thus conclude that regression models are more suitable than classifiers models for prioritization problems, unlike what was used by the related-work. We also conclude that ensemble models are the most suitable models for our prioritization environment, this time as suggested by the related-work.

As for final results, our best model proved to able to shorten the extensive work of bug analysis by almost 80%. A developer analysing bug reports prioritized by this model would have encountered all true bugs after analysing only 21% of all reports in the dataset, an almost 5x increase in efficiency when compared to non-prioritization.

7.2 Answer to research questions

After discussing the whole project we can now answer the research question posed at the beginning of this thesis.

RQ1: Does the SARIF format provide enough information on consensus to achieve reasonable results in bug prioritization?

A:Yes, using SARIF only features our machine learning models were capable of achieving high values in all presented metrics. Our best model could even make use of SARIF to reach an APBD level of 0.932. As explored in Chapter 4 we have also used the SARIF format to join multiple reports about the same bug together, so it clearly has enough information to do that. Considering that the format did not hinder our development cycle and that it allowed for good results, we can thus conclude that the SARIF format does provide enough information on consensus to achieve reasonable results in bug prioritization

RQ2: Can consensus of bug reports between multiple tools be used to better predict true and false positives?

A: Yes, consensus was used in this thesis to develop multiple worthwhile machine learning models capable of prioritizing bug reports. The best model was even capable of putting all true bug in the dataset in the first 21% of the prioritized list of reports. This would have saved the developer tasked with analysing said list almost 80% of its time. As this result was achieved through a model that learns on consensus alone, we can safely conclude that consensus of bug reports between multiple tools can be used to better predict true and false positives.

RQ3: What are some effective machine learning algorithm to prioritize bug reports from multiple tools in the context of SmartBugs?

A: Based on our experiments, the most effective machine learning algorithm to prioritize bug reports from multiple tools in the context of SmartBugs is the Random Forest Regressor model. Regressor models proved to be more effective than classifiers and ensembles more effective than their simpler counterparts. Out of all ensemble regressors tested, the Random Forest Regressor model had the better results, with an APBD of 0.932.

7.3 Threats to validity

Throughout this thesis we have presented a conjunction of assumptions, that if false, could prove a danger to the findings presented. First amongst them is the data used in the datasets. As manually checking the thousands of bugs in our datasets is unviable, we have trusted the classification provided by the datasets themselves. Should this classification prove largely incorrect, than it is highly likely that our model is also inaccurate. As these classifications are used for training and testing both, all evaluation metrics become untrustworthy if the datasets used prove to have a high amount of missed classifications. This is to say that not only will the model prove to be inaccurate, the evaluation will be equally unusable. This is a common danger between all machine learning models.

The second danger is the mapping utilized in Chapter 4. As described in the same chapter, said

mapping is used to join multiple reports of multiple analysis tools together, as well as deciding their classification. If the mapping proves to be massively incorrect, than the data fed to machine learning model will also be incorrect. Although its consequences would likely be smaller than that of the first danger presented, it danger itself is similar.

The third danger refers to all the libraries imported during the development of this work. Many of the machine learning models and metrics were implemented with help from the scikit-learn library for python [38], so if the library itself has bugs, so will the function implemented with it. The probability of this being the case in incredibly small, as this is a massive open-source library that has been peered-reviewed multiple times.

7.4 Future work

Future work on this topic should look to further increase the amount of tools available to the SmartBugs framework, therefore increasing the potential upside for consensus between multiple tools. This work has shown that consensus can be a useful metric for bug prioritization, so future work should seek to further capitalize on this knowledge.

Second, it might be useful to develop a machine learning model that not only depends on SARIF features and consensus although they have proven to be useful features. It could prove effective to not only use consensus but to also to look at the source code directly in some way, shape or form. This is, to develop our own bug analysis tool that would work in tandem with the solution presented in this paper. This would be in agreement with Koc et al [3], that concluded for their context that machine learning models that use the source code as features have better results than models that do not. While the context of our work only looked at the results of tools available to SmartBugs, it could perhaps show even greater results by merging both aproaches.

Bibliography

[1] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "Smartbugs: A framework to analyze solidity smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1349–1352. [Online]. Available: https://doi.org/10.1145/3324884.3415298

[2] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 530–541. [Online]. Available: https://doi.org/10.1145/3377811.3380364

[3] U. Koc, S. Wei, J. S. Foster, M. Carpuat, and A. A. Porter, "An empirical assessment of machine learning approaches for triaging reports of a java static analysis tool," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 288–299.

[4] B. Miranda, I. Lima, O. Legunsen, and M. d'Amorim, "Prioritizing runtime verification violations," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 297–308.

[5] T. Kremenek and D. R. Engler, "Z-ranking: Using statistical analysis to counter the impact of static analysis approximations," in *SAS*, 2003.

[6] Ethereum. (2021) Solidity documentation. [Online]. Available: https://docs.soliditylang.org/en/v0.8.10/

[7] The dao hack. [Online]. Available: https://www.coindesk.com/learn/2016/06/25/understanding-the-dao-attack

[8] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. T. Vechev, "Securify: Practical security analysis of smart contracts," *CoRR*, vol. abs/1806.01143, 2018. [Online]. Available: http://arxiv.org/abs/1806.01143

[9] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 653–663. [Online]. Available: https://doi.org/10.1145/3274694.3274743

[10] C. F. Torres, M. Steichen, and R. State, "The art of the scam: Demystifying honeypots in ethereum smart contracts," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1591–1607. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/ferreira

[11] Oasis. (2021) Sarif documentation. [Online]. Available: https://docs.oasis-open.org/sarif/sarif/v2.0/sarif-v2.0.html/

[12] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, *EThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts*. New York, NY, USA: Association for Computing Machinery, 2020, p. 621–640. [Online]. Available: https://doi.org/10.1145/3372297.3417250

[13] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov, "How good are the specs? a study of the bug-finding effectiveness of existing java api specifications," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 602–613.

[14] Docker. (2021) Docker documentation. [Online]. Available: https://www.docker.com/

[15] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," *CoRR*, vol. abs/1907.03890, 2019. [Online]. Available: http://arxiv.org/abs/1907.03890

[16] B. Muelle, "Smashing ethereum smart contracts for fun and real-profit," 2018. [Online]. Available: https://conference.hitb.org/hitbsecconf2018ams/sessions/smashing-ethereum-smart-contracts-for-fun-and-actual-profit/

[17] C. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," 12 2018.

[18] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 254–269. [Online]. Available: https://doi.org/10.1145/2976749.2978309

[19] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," *CoRR*, vol. abs/1908.09878, 2019. [Online]. Available: http://arxiv.org/abs/1908.09878

[20] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2018, pp. 9–16.

[21] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," 08 2019.

[22] N. Velose. (2018) Conkas. [Online]. Available: https://github.com/nveloso/conkas

[23] M. Alenezi and S. Banitaan, "Bug reports prioritization: Which features and classifier to use?" in *2013 12th International Conference on Machine Learning and Applications*, vol. 2, 2013, pp. 112–116.

[24] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin, "Aletheia: Improving the usability of static security analysis," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 762–774. [Online]. Available: https://doi.org/10.1145/2660267.2660339

[25] Joana (java object-sensitive analysis) - information flow control framework for java. [Online]. Available: https://pp.ipd.kit.edu/projects/joana

[26] Find security bugs. [Online]. Available: http://find-sec-bugs.github.io

[27] The owasp benchmark for security automation. [Online]. Available: https://www.owasp.org/index.php/Benchmark

[28] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov, "How good are the specs? a study of the bug-finding effectiveness of existing java api specifications," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 602–613.

[29] SolidiFI. (2018) Solidifi. [Online]. Available: https://github.com/DependableSystemsLab/SolidiFI-benchmark

[30] HuangGai. (2018) Huanggai. [Online]. Available: https://github.com/xf97/HuangGai

[31] Dasp. (2018) Dasp. [Online]. Available: https://dasp.co/#item-1

[32] JiuZhou. (2018) Jiuzhou. [Online]. Available: https://github.com/xf97/JiuZhou

[33] rjust. (2018) defects4j. [Online]. Available: https://github.com/rjust/defects4j

[34] bugsjs. (2018) bugsjs. [Online]. Available: https://bugsjs.github.io/

[35] B. prediction dataset. (2018) Bug prediction dataset. [Online]. Available: https://bug.inf.usi.ch/index.php

[36] B.-F. Dataset. (2018) Bug-fix dataset. [Online]. Available: https://figshare.com/articles/dataset/Replication_Package_-_PROMISE_19/8852084

[37] U. B. Dataset. (2018) Unified bug dataset. [Online]. Available: http://www.inf.u-szeged.hu/~ferenc/papers/UnifiedBugDataSet

[38] scikit learn. (2018) scikit-learn. [Online]. Available: https://scikit-learn.org/

A

Mapping Tables

This appendix section has the mapping tables presented in chapter 4

A.1 Mapping between SmartBugs tools' vulnerabilities and DASP10 vulnerability types

Tools	Vulnerability name	Vulnerability Type
Maian	is_lock_vulnerable	Other
Maian	is_prodigal_vulnerable	Access Control
Maian	is_suicidal_vulnerable	Access Control
Manticore	Delegatecall to user controlled address	Access Control
Manticore	Delegatecall to user controlled function	Access Control
Manticore	INVALID instruction	Other
Manticore	Potential reentrancy vulnerability	Denial of Service
Manticore	Potentially reading uninitialized memory at instruction	Other
Manticore	Potentially reading uninitialized storage	Other
Manticore	Reachable ether leak to sender	Access Control
Manticore	Reachable ether leak to sender via argument	Access Control
Manticore	Reachable external call to sender	Access Control
Manticore	Reachable external call to sender via argument	Access Control
Manticore	Reachable SELFDESTRUCT	Access Control
Manticore	Reentrancy multi-million ether bug	Denial of Service
Manticore	Returned value at CALL instruction is not used	Unchecked Return Values For Low Level Calls
Manticore	Returned value at CALL instruction is not used	Unchecked Return Values For Low Level Calls
Manticore	Unsigned integer overflow at ADD instruction	Arithmetic
Manticore	Unsigned integer overflow at MUL instruction	Arithmetic
Manticore	Unsigned integer overflow at SUB instruction	Arithmetic
Manticore	Warning BLOCKHASH instruction used	Other
Manticore	Warning NUMBER instruction used	Other
Manticore	Warning ORIGIN instruction used	Access Control
Manticore	Warning TIMESTAMP instruction used	Time Manipulation
Mythril	Call data forwarded with delegatecall()	Access Control
Mythril	DELEGATECALL to a user-supplied address	Access Control
Mythril	Dependence on predictable environment variable	Other
Mythril	Dependence on predictable variable	Other
Mythril	Ether send	Access Control
Mythril	Exception state	Other
Mythril	Integer Overflow	Arithmetic
Mythril	Integer Underflow	Arithmetic
Mythril	Message call to external contract	Denial of Service
Mythril	Multiple Calls	Ignore
Mythril	State change after external call	Denial of Service
Mythril	Transaction order dependence	Front-Running
Mythril	Unchecked CALL return value	Unchecked Return Values For Low Level Calls
Mythril	Unchecked SUICIDE	Access Control
Mythril	Use of tx.origin	Access Control
Osiris	callstack_bug	Denial of Service
Osiris	concurrency_bug	Ignore
Osiris	division_bugs	Arithmetic
Osiris	overflow_bugs	Arithmetic
Osiris	reentrancy_bug	Denial of Service
Osiris	signedness_bugs	Arithmetic
Osiris	time_dependency_bug	Time Manipulation
Osiris	truncation_bugs	Arithmetic
Osiris	underflow_bugs	Arithmetic
Oyente	Callstack Depth Attack Vulnerability.	Denial of Service
Oyente	Integer Overflow.	Arithmetic
Oyente	Integer Underflow.	Arithmetic
Oyente	Parity Multisig Bug 2.	Access Control
Oyente	Re-Entrancy Vulnerability.	Denial of Service

Oyente	Timestamp Dependency.	Time Manipulation
Oyente	Transaction-Ordering Dependency.	Front-Running
Securify	DAO	Denial of Service
Securify	DAOConstantGas	Denial of Service
Securify	LockedEther	Other
Securify	MissingInputValidation	Ignore
Securify	RepeatedCall	Ignore
Securify	TODAmount	Front-Running
Securify	TODReceiver	Front-Running
Securify	TODTransfer	Front-Running
Securify	UnhandledException	Unchecked Return Values For Low Level Calls
Securify	UnrestrictedEtherFlow	Access Control
Securify	UnrestrictedWrite	Access Control
Slither	arbitrary-send	Access Control
Slither	assembly	Ignore
Slither	calls-loop	Denial of Service
Slither	constable-states	Ignore
Slither	constant-function	Ignore
Slither	controlled-delegatecall	Access Control
Slither	deprecated-standards	Ignore
Slither	erc20-indexed	Ignore
Slither	erc20-interface	Ignore
Slither	external-function	Ignore
Slither	incorrect-equality	Other
Slither	locked-ether	Other
Slither	low-level-calls	Unchecked Return Values For Low Level Calls
Slither	naming-convention	Ignore
Slither	reentrancy-benign	Denial of Service
Slither	reentrancy-eth	Denial of Service
Slither	reentrancy-no-eth	Denial of Service
Slither	shadowing-abstract	Ignore
Slither	shadowing-builtin	Ignore
Slither	shadowing-local	Ignore
Slither	shadowing-state	Ignore
Slither	solc-version	Ignore
Slither	suicidal	Access Control
Slither	timestamp	Time Manipulation
Slither	tx-origin	Access Control
Slither	uninitialized-local	Other
Slither	uninitialized-state	Other
Slither	uninitialized-storage	Other
Slither	unused-return	Unchecked Return Values For Low Level Calls
Slither	unused-state	Ignore
Smartcheck	SOLIDITY_ADDRESS_HARDCODED	Ignore
Smartcheck	SOLIDITY_ARRAY_LENGTH_MANIPULATION	Arithmetic
Smartcheck	SOLIDITY_BALANCE_EQUALITY	Other
Smartcheck	SOLIDITY_BYTE_ARRAY_INSTEAD_BYTES	Ignore
Smartcheck	SOLIDITY_CALL_WITHOUT_DATA	Denial of Service
Smartcheck	SOLIDITY_DEPRECATED_CONSTRUCTIONS	Ignore
Smartcheck	SOLIDITY_DIV_MUL	Arithmetic
Smartcheck	SOLIDITY_ERC20_APPROVE	Ignore
Smartcheck	SOLIDITY_ERC20_FUNCTIONS_ALWAYS_RETURN_FALSE	Ignore
Smartcheck	SOLIDITY_ERC20_TRANSFER_SHOULD_THROW	Ignore
Smartcheck	SOLIDITY_EXACT_TIME	Time Manipulation

Smartcheck	SOLIDITY_EXTRA_GAS_IN_LOOPS	Ignore
Smartcheck	SOLIDITY_FUNCTIONS_RETURNS_TYPE_AND_NO_RETURN	Other
Smartcheck	SOLIDITY_GAS_LIMIT_IN_LOOPS	Denial of Service
Smartcheck	SOLIDITY_INCORRECT_BLOCKHASH	Other
Smartcheck	SOLIDITY_LOCKED_MONEY	Other
Smartcheck	SOLIDITY_MSGVALUE_EQUALS_ZERO	Ignore
Smartcheck	SOLIDITY_OVERPOWERED_ROLE	Ignore
Smartcheck	SOLIDITY_PRAGMAS_VERSION	Ignore
Smartcheck	SOLIDITY_PRIVATE_MODIFIER_DONT_HIDE_DATA	Ignore
Smartcheck	SOLIDITY_REDUNDANT_FALLBACK_REJECT	Ignore
Smartcheck	SOLIDITY_REVERT_REQUIRE	Ignore
Smartcheck	SOLIDITY_SAFEMATH	Ignore
Smartcheck	SOLIDITY_SEND	Unchecked Return Values For Low Level Calls
Smartcheck	SOLIDITY_SHOULD_NOT_BE_PURE	Ignore
Smartcheck	SOLIDITY_SHOULD_NOT_BE_VIEW	Ignore
Smartcheck	SOLIDITY_SHOULD_RETURN_STRUCT	Ignore
Smartcheck	SOLIDITY_TRANSFER_IN_LOOP	Denial of Service
Smartcheck	SOLIDITY_TX_ORIGIN	Access Control
Smartcheck	SOLIDITY_UINT_CANT_BE_NEGATIVE	Arithmetic
Smartcheck	SOLIDITY_UNCHECKED_CALL	Unchecked Return Values For Low Level Calls
Smartcheck	SOLIDITY_UPGRADE_TO_050	Ignore
Smartcheck	SOLIDITY_USING_INLINE_ASSEMBLY	Ignore
Smartcheck	SOLIDITY_VAR	Arithmetic
Smartcheck	SOLIDITY_VAR_IN_LOOP_FOR	Arithmetic
Smartcheck	SOLIDITY_VISIBILITY	Ignore
Smartcheck	SOLIDITY_WRONG_SIGNATURE	Ignore
Solhint	indent	Ignore
Solhint	max-line-length	Ignore
Honeybadger	hidden_state_update	Ignore
Honeybadger	uninitialised_struct	Other
Honeybadger	inheritance_disorder	Ignore
Honeybadger	straw_man_contract	Denial of Service
Honeybadger	hidden_transfer	Other
Honeybadger	balance_disorder	Ignore
Honeybadger	type_overflow	Arithmetic
Conkas	Reentrancy	Denial of Service
Conkas	Integer_Overflow	Arithmetic
Conkas	Integer_Underflow	Arithmetic
Conkas	Unchecked Low Level Call	Unchecked Return Values For Low Level Calls
Conkas	Transaction Ordering Dependence	Front-Running
Conkas	Time Manipulation	Time Manipulation

B

Full results table

This table contains the full list of results presented in chapter 6

B.1 Full results table, including training and testing metrics for all tested models.

Model	TR Accuracy	TR F1-Score	TR Recall	TR Precision	TE Accuracy	TE F1-Score	TE Recall	TE Precision	APBD
Decision Tree Classifier	0.984	0.984	0.984	0.983	0.985	0.947	0.987	0.909	0.932
Perceptron	0.882	0.889	0.943	0.840	0.836	0.606	0.941	0.447	0.830
Passive Aggressive Classifier	0.548	0.285	0.157	0.719	0.833	0.206	0.161	0.285	0.541
Ridge Classifier Cross Validation	0.886	0.895	0.970	0.830	0.822	0.594	0.971	0.428	0.835
SGD Classifier	0.890	0.898	0.961	0.842	0.837	0.613	0.961	0.450	0.838
KNeighbors Classifier	0.980	0.980	1.0	0.962	0.967	0.891	0.998	0.803	0.916
Decision Tree Regressor	0.984	0.984	0.984	0.938	0.985	0.947	0.987	0.909	0.932
Logistic Regressor Cross-Validation	0.897	0.900	0.934	0.869	0.868	0.655	0.935	0.503	0.884
KNeighbors Regressor	0.980	0.980	1.0	0.962	0.967	0.891	0.999	0.803	0.930
Linear Regressor	0.980	0.980	1.0	0.962	0.822	0.594	0.971	0.428	0.882
SGD Regressor	0.980	0.980	1.0	0.962	0.753	0.516	0.980	0.350	0.889
Ridge Cross-Validation	0.980	0.980	1.0	0.962	0.822	0.594	0.971	0.428	0.882
ARD Regressor	0.980	0.980	1.0	0.962	0.822	0.594	0.971	0.428	0.901
Bayesian Ridge	0.980	0.980	1.0	0.962	0.822	0.594	0.971	0.428	0.882
Ada Boost Classifier	0.968	0.969	0.991	0.947	0.952	0.847	0.991	0.740	0.906
Extra Trees Classifier	0.984	0.984	0.984	0.983	0.985	0.947	0.987	0.909	0.921
Gradient Boosting Classifier	0.979	0.979	1.0	0.960	0.964	0.882	0.998	0.789	0.915
Random Forest Classifier	0.984	0.984	0.984	0.983	0.985	0.947	0.967	0.909	0.921
Hist Gradient Boosting Classifier	0.984	0.984	0.984	0.983	0.985	0.946	0.987	0.909	0.921
Stacking Classifier	0.984	0.984	0.984	0.983	0.985	0.947	0.987	0.909	0.921
Voting Classifier	0.984	0.984	0.984	0.983	0.985	0.947	0.987	0.909	0.921
Ada Boost Regressor	0.917	0.918	0.940	0.898	0.897	0.710	0.942	0.570	0.884
Extra Trees Regressor	0.984	0.984	0.984	0.983	0.985	0.947	0.987	0.909	0.932
Gradient Boosting Regressor	0.979	0.979	1.0	0.960	0.964	0.883	0.999	0.791	0.931
Random Forest Regressor	0.984	0.984	0.984	0.983	0.985	0.947	0.987	0.909	0.932
Hist Gradient Boosting Regressor	0.983	0.983	0.984	0.983	0.984	0.946	0.987	0.908	0.932
Stacking Regressor	0.980	0.980	1.0	0.962	0.967	0.891	0.999	0.803	0.929
Voting Regressor	0.980	0.981	1.0	0.962	0.967	0.891	0.999	0.804	0.932

Table B.1: Full metrics for all models tested. TE stands for testing metrics and TR stands for training metrics