# Automatic Bug Prioritization of SmartBugs Reports using Machine Learning

João Tiago Sousa Dinis
joao.tiago.dinis@tecnico.ulisboa.pt
Instituto Superior Técnico
Lisbon, Portugal

## 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 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 efficiency in the process of analysing bug reports.

## CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

## KEYWORDS

SmartBugs, Bug Prioritization, Machine Learning

**ACM Reference Format:**
João Tiago Sousa Dinis. 2022. Automatic Bug Prioritization of SmartBugs Reports using Machine Learning. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

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 [16] 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 [11] 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 [8] 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 [2] 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 [7] 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 [31]), some are dynamic analysis tools (e.g. Maian [22]) and some, like HoneyBadger [30], 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) [23] 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.


*ISSTA 2023, 17-21 July, 2023, Seattle, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Alternatively, bugs reported by a single tool might prove to be less trustworthy reports. There also exist other tools such as eThor [26], 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([19], [15]), 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 [23] 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 is the most effective machine learning algorithm to prioritize bug reports from multiple tools in the context of SmartBugs?

## 2 BACKGROUND

This section describes some background information necessary to understand the rest of this paper.

### 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. [17] 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 [19], 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 [16] 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 SmartBugs

SmartBugs [11] 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 section, 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 [6]. Where images were available said images were used, otherwise they were created and publicized. The 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 [30], Maian [22], Manticore [20], Mythril [21], Osiris [29], Oyente [18], Securify [31], Slither [9], Smartcheck [28], Solhint [10] and Conkas [32]. 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.

This paper extends the SmartBugs framework by adding a prioritization mechanism capable of ranking each bug report by the likelihood of being a true bug. Reports from all 11 different tools will be merged together and fed to a machine learning model so it can learn to prioritize them based on the consensus between the various tools. The machine learning in question is further explained in the next sections.

## 2.3 Bug Prioritization with Machine Learning

While we have not come across any other works on the subject of bug prioritization of reports provided by multiple analysis tools, bug prioritization itself is a well studied-field. This subsection looks at some of these works.

*2.3.1 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" [15] 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. The static analysis tool selected for the study was FindSecBugs (version 1.4.6) [12], a popular security checker for Java web applications.

The authors tested multiple machine learning models with a variety of features to understand which one performed the best. The results point to Long Short Term Memory Recurrent Neural Networks and Gated Graph Neural Networks being the best models among the ones tested. This, the authors theorize, is because they are the only two tested models that retain information about the order/relation of the code in the initial program. This would suggest that models that keep more information about the original program have better results than models that keep less. Additionally, the authors also test multiple versions of data preparation for each model. Their conclusions on this aspect lead to the idea that more data preparation leads to "cleaner" features and better results.

*2.3.2 Prioritizing Runtime Verification Violations.* This study by Miranda et al. [19] 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 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. From the tested models, it was the Gradient Boosting Classifier, an ensemble model, that performed the best. 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 when compared to simple classifiers, and RVPrio confirms this to be the case.

This work also presents the APBD metric to evaluate the prioritized list of reports, as well as a prioritization graph and time table. We have also chosen to implement these metrics and they are further discussed in future sections.

# 3 SMARTBUGS REPORT PRIORITIZATION

This next section focuses on the bug prioritization mechanism implemented into the SmartBugs framework.

## 3.1 Data Preparation

The first step of our solution is to prepare the data that will later be fed to the machine learning model. This includes the collection of datasets, merging of bug reports, and even a proposed standard for bug repositories.

*3.1.1 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, than 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 [27], containing 550 faulty smart contracts created through bug injection
(3) HuangGai's manual dataset [13], 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 developer's 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.

After acquiring the datasets, the next step is to run the SmartBugs framework with all tools enable and save the output SARIF file.

*3.1.2 Merging Bug Reports Together.* The next 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 through which it learns.

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 a bug of the arithmetic category, while a different tool reports a bug of the access control category 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 [3]. 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

Each rule id of each tool was mapped to one of these DASP10 categories that we will continue to use throughout this project. Using these categories we can follow the two rules presented above and join our bug reports together.

*3.1.3 Retrieving Information from Datasets.* 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 sub section.

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.

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

Each dataset had its own way to store information, from JSON files to CSV files and even comments. The location of the bug was retrieved from these files and the category was inferred from the datasets folder structure. It was however necessary to map these categories (each dataset had its own), to the DASP10 categories presented above. This is necessary for the future step of matching bug reports resulting from SmartBugs execution to the actual bugs of the dataset. At the end, we are left with the location and DASP10 category for each bug in the dataset that we store in a specific format.

*3.1.4 Proposed Format.* The format we use to store our bug's metadata is as 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 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.

```
14
15  function deal (uint8 cardNumber) returns (uint8) {
16    uint b = block.number;
17    uint timestamp = block.timestamp;
18    return uint8(b * timestamp % 52);
19  }
20
```

**Figure 1: bad randomness vulnerability example**

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 the bad randomness vulnerability presented in figure 1 for example.

Variables with bad randomness were declared in lines 16 and 17. They were however, only used in line 18, which is where the vulnerability applies. Where should this bug be reported? In lines 16 and 17, or line 18? Both approaches seem reasonable and it is up to the dataset's author to decide. If the author marks lines 16 and

**Table 1: Metadata method per bug dataset**

| Dataset | Language | Method |
|---|---|---|
| SmartBugs [11] | solidity | JSON & Comments |
| SolidiFI [27] | solidity | CSV & Comments |
| Jiuzhou [14] | solidity | Json |
| Huang Gai [13] | solidity | Txt & Comments |
| Defects4JS [25] | java | CSV |
| BugsJS [1] | java | CSV |
| The Bug Prediction Dataset [24] | java | CSV |
| Bug-Fix Dataset [4] | java | CSV |
| Unified Bug Dataset [5] | java | CSV |

17, he is marking 2 different bugs, and a report on line 18 would be considered false. If the author marks line 18, then only one bug is declared and any reports on lines 16 or 17 will be considered false. Using our format however, the author could write 16-17/18 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 1 describes what methods are currently in use to present the metadata related to bugs in nine datasets.

As seen in table 1, 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.

*3.1.5 Final Data.* 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.

## 3.2 Machine Learning Model

This sub section describes the details about the machine learning model implemented. First it describes some additional data preparation that had to implemented and then discusses the features available to the model.

*3.2.1 Data Division and Balancing.* Two matters had to be addressed to finish preparing the data that is fed to the machine learning model.

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.

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 section, 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.

*3.2.2 Features and Consensus.* The final data to reach the machine learning model is a balanced dataset where 80% of the dataset will be used for training and the remaining 20% for testing. The features available to the model are the columns of the CSV file presented in the previous section, this is the rule_id and level resulting from the analysis of each tool for each bug report. Should a tool not have reported this bug, then these columns are left empty.

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, than 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.

# 4 EXPERIMENTAL SETUP

The next section describes the experimental setup used in this project.

## 4.1 Metrics

The first thing to consider are the metrics chosen to evaluate the solution. We have classification metrics that evaluate the machine learning model and prioritization metrics that evaluate the final prioritized list of bug reports.

*4.1.1 Classification Metrics.* They require discrete classifications to be calculated. For regression models, we consider reports with over 50% probability of being true as true for the context of these metrics. The 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 section, 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 metrics is:

$$Accuracy = \frac{True\ positives + True\ negatives}{All\ Samples}$$

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, a 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 section.

*4.1.2 Prioritization Metrics.* These metrics evaluate the prioritized list of bug reports directly. This list is produced based on the results of the machine learning model. After the model classifies the reports they are then put in a list, where those considered most likely to be true are presented first, and those considered less likely to be true are presented last. This list is then evaluated through various metrics.

APBD stands for Average Percentage of Bugs Detected. It is a metric presented in RVPrio's report [19], discussed in Section 2. This metric takes a prioritized list of bug reports 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.


**Figure 2: Example of a prioritization graph**

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.

The next 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 2 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 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

**Table 2: Testing metrics for each ensemble model tested**

| Model | Accuracy | F1-Score | Recall | Precision | APBD |
|---|---|---|---|---|---|
| Decision Tree Class | 0.985 | 0.947 | 0.987 | 0.909 | 0.921 |
| Perceptron | 0.836 | 0.606 | 0.941 | 0.447 | 0.830 |
| Passive Aggressive Class | 0.833 | 0.206 | 0.161 | 0.285 | 0.541 |
| Ridge Class | 0.822 | 0.594 | 0.971 | 0.428 | 0.835 |
| SGD Class | 0.837 | 0.613 | 0.961 | 0.450 | 0.838 |
| KNeighbors Class | 0.967 | 0.891 | 0.999 | 0.803 | 0.916 |
| Decision Tree Regr | 0.985 | 0.947 | 0.987 | 0.909 | 0.932 |
| Logistic Regr | 0.868 | 0.655 | 0.935 | 0.503 | 0.884 |
| KNeighbors Regr | 0.967 | 0.891 | 0.999 | 0.803 | 0.930 |
| Linear Regr | 0.822 | 0.594 | 0.971 | 0.428 | 0.882 |
| SGD Regr | 0.753 | 0.516 | 0.980 | 0.350 | 0.899 |
| Ridge | 0.882 | 0.594 | 0.971 | 0.428 | 0.882 |
| ARD Regr | 0.822 | 0.594 | 0.971 | 0.428 | 0.901 |
| Bayesian Ridge | 0.822 | 0.594 | 0.971 | 0.428 | 0.882 |
| Ada Boost Class | 0.952 | 0.874 | 0.991 | 0.741 | 0.906 |
| Extra Trees Class | 0.985 | 0.947 | 0.987 | 0.909 | 0.921 |
| Gradient Boosting Class | 0.964 | 0.882 | 0.999 | 0.789 | 0.915 |
| Random Forest Class | 0.985 | 0.947 | 0.987 | 0.909 | 0.921 |
| Hist Gradient Boosting Class | 0.985 | 0.946 | 0.987 | 0.909 | 0.921 |
| Stacking Class | 0.985 | 0.947 | 0.987 | 0.909 | 0.921 |
| Voting Class | 0.985 | 0.947 | 0.987 | 0.909 | 0.921 |
| Ada Boost Regr | 0.897 | 0.710 | 0.942 | 0.570 | 0.884 |
| Extra Trees Regr | 0.985 | 0.947 | 0.987 | 0.909 | 0.932 |
| Gradient Boosting Regr | 0.964 | 0.883 | 0.999 | 0.791 | 0.931 |
| Random Forest Regr | 0.985 | 0.947 | 0.987 | 0.909 | 0.932 |
| Hist Gradient Boosting Regr | 0.984 | 0.946 | 0.987 | 0.908 | 0.932 |
| Stacking Regr | 0.967 | 0.891 | 0.998 | 0.803 | 0.929 |
| Voting Regr | 0.967 | 0.891 | 0.999 | 0.804 | 0.932 |

line) than 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 section.

## 5 EVALUATION

This section discusses the results of our experiments with a variety of machine learning models.

Table 2 shows the results for all tested machine learning models. Classifier models have been abbreviated to 'Class' and regressor models to 'Reg'. The first block presents pure classifier models, the second presents pure regressor models, the third shows ensemble classifier models and the fourth block shows ensemble regressor models.

The table presents favorable results for most models. While some models such as the Passive Aggressive Classifier have high values in some metrics (namely its high accuracy of 0.833) but low results in others (namely its low recall of just 0.161), many models have consistently high results across the board. This phenomenon of high variance between the metrics likely means that these models are 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 shows the need to look at all metrics as a whole.

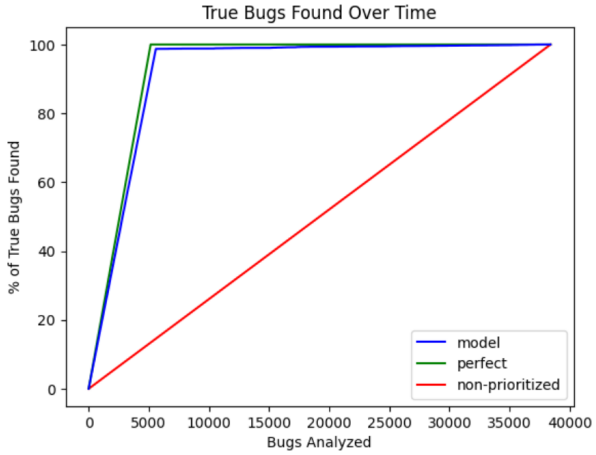


**Figure 3: Random Forest Classifier model's prioritization graph**

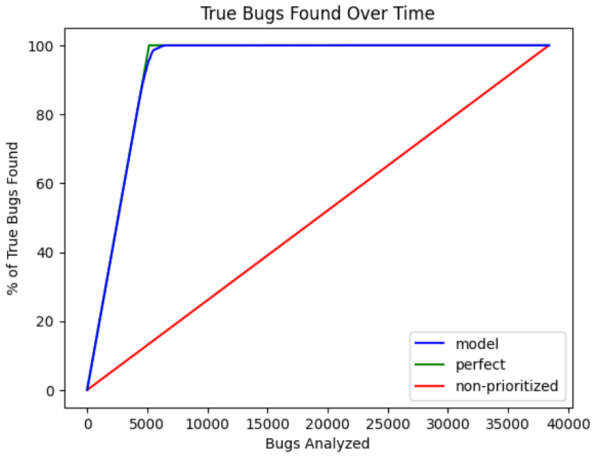


**Figure 4: Random Forest Regressor model's prioritization graph**

Dividing the table into blocks we can see that regressors have overall better performance than classifiers and that ensembles have more consistent metrics than either pure classifiers or regressors. This seems to suggest that regressor ensembles are the most suitable type of model for our environment. This makes us wonder why the related-work instead choose to use classifiers over regressors.

This point is further illustrated in figure 3 and figure 4 that show the prioritization graph for one of our best performing ensemble models, the Random Forest model. Figure 3 shows the graph for the classifier version and 4 for the regressor version.

As can be seen by comparing these graphs, the regressor version of this model has a closer resemblance to the green line when compared to the classifier version. This is particularly noticeable at around the 95% mark on the vertical axis. This is because classifiers are limited in their prioritization due to their binary output. As the model simply classifies the report as true or false, we can only ever

**Table 3: Time table for the Random Forest Regressor model**

| 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 |

prioritize the list of reports by putting those classified as true first and those classified as false second. There is no way to distinguish between two true or two false bug reports so the model cannot recuperate well from mistakes in its classification.

The regressor model, however, gives a continuous output that is more useful for prioritization. Reports can be organized by their continuous probability value and so the model can more easily recover from any missed classifications. This results in a much smoother curve when transitioning from reports that the model considers as true to reports the model considers as false.

Perhaps the best way to express the advantages of this prioritization is through table 3 shows a time table for the Random Forest Regressor model.

The table expresses that after analyzing 10% of bug reports in a non-prioritized list a developer would have found only 10% of true bugs, but analysing the same 10% of reports in a list prioritized by a Random Forest Regressor, the developer would instead find 75% of all true bugs, a 7.5x 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. 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.

# 6 CONCLUSIONS

The contributions of this paper 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 of this project. 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 the most suitable 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.

## 6.1 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 is the most 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.

## 6.2 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

[15], 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.

## REFERENCES


[1] bugsjs. 2018. *bugsjs.* https://bugsjs.github.io/

[2] coindesk. 2016. *The DAO Hack.* https://www.coindesk.com/learn/2016/06/25/understanding-the-dao-attack

[3] Dasp. 2018. *Dasp.* https://dasp.co/#item-1

[4] Bug-Fix Dataset. 2018. *Bug-Fix Dataset.* https://figshare.com/articles/dataset/Replication_Package_-_PROMISE_19/8852084

[5] Unified Bug Dataset. 2018. *Unified Bug Dataset.* http://www.inf.u-szeged.hu/~ferenc/papers/UnifiedBugDataSet

[6] Docker. 2021. *Docker Documentation.* https://www.docker.com/

[7] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 530–541. https://doi.org/10.1145/3377811.3380364

[8] Ethereum. 2021. *Solidity Documentation.* https://docs.soliditylang.org/en/v0.8.10/

[9] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework For Smart Contracts. *CoRR* abs/1908.09878 (2019). arXiv:1908.09878 http://arxiv.org/abs/1908.09878

[10] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework For Smart Contracts. (08 2019).

[11] João F. Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2020. SmartBugs: A Framework to Analyze Solidity Smart Contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1349–1352. https://doi.org/10.1145/3324884.3415298

[12] FindSecBugs. 2022. *Find Security Bugs.* http://find-sec-bugs.github.io

[13] HuangGai. 2018. *HuangGai.* https://github.com/xf97/HuangGai

[14] JiuZhou. 2018. *JiuZhou.* https://github.com/xf97/JiuZhou

[15] Ugur Koc, Shiyi Wei, Jeffrey S. Foster, Marine Carpuat, and Adam A. Porter. 2019. 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)*. 288–299. https://doi.org/10.1109/ICST.2019.00036

[16] Theodore Kremenek and Dawson R. Engler. 2003. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *SAS*.

[17] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. 2016. 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)*. 602–613.

[18] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

[19] Breno Miranda, Igor Lima, Owolabi Legunsen, and Marcelo d'Amorim. 2020. Prioritizing Runtime Verification Violations. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 297–308. https://doi.org/10.1109/ICST46399.2020.00038

[20] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. *CoRR* abs/1907.03890 (2019). arXiv:1907.03890 http://arxiv.org/abs/1907.03890

[21] Bernhard Muelle. 2018. Smashing ethereum smart contracts for fun and real-profit. https://conference.hitb.org/hitbsecconf2018ams/sessions/smashing-ethereum-smart-contracts-for-fun-and-actual-profit/

[22] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 653–663. https://doi.org/10.1145/3274694.3274743

[23] Oasis. 2021. *SARIF Documentation.* https://docs.oasis-open.org/sarif/sarif/v2.0/sarif-v2.0.html/

[24] Bug prediction dataset. 2018. *Bug prediction dataset.* https://bug.inf.usi.ch/index.php

[25] rjust. 2018. *defects4j.* https://github.com/rjust/defects4j

[26] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. *EThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts.* Association for Computing Machinery, New York, NY, USA, 621–640. https://doi.org/10.1145/3372297.3417250

[27] SolidiFI. 2018. *SolidiFI.* https://github.com/DependableSystemsLab/SolidiFI-benchmark

[28] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 9–16.

[29] Christof Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts.

[30] Christof Ferreira Torres, Mathis Steichen, and Radu State. 2019. The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1591–1607. https://www.usenix.org/conference/usenixsecurity19/presentation/ferreira

[31] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. *CoRR* abs/1806.01143 (2018). arXiv:1806.01143 http://arxiv.org/abs/1806.01143

[32] Nuno Velose. 2018. *Conkas.* https://github.com/nveloso/conkas


Received 31 October 2022