Automatically Assessing Vulnerabilities Discovered by Compositional Analysis

Saahil Ognawala
Technical University of Munich
Germany
saahil.ognawala@tum.de

Ricardo Nales Amato
Technical University of Munich
Germany
ricardo.nales@tum.de

Alexander Pretschner
Technical University of Munich
Germany
alexander.pretschner@tum.de

Pooja Kulkarni
Technical University of Munich
Germany
pooja.kulkarni@tum.de

ABSTRACT

Testing is the most widely employed method to find vulnerabilities in real-world software programs. Compositional analysis, based on symbolic execution, is an automated testing method to find vulnerabilities in medium- to large-scale programs consisting of many interacting components. However, existing compositional analysis frameworks do not assess the severity of reported vulnerabilities. In this paper, we present a framework to analyze vulnerabilities discovered by an existing compositional analysis tool and assign CVSS3 (Common Vulnerability Scoring System v3.0) scores to them, based on various heuristics such as interaction with related components, ease of reachability, complexity of design and likelihood of accepting unsanitized input. By analyzing vulnerabilities reported with CVSS3 scores in the past, we train simple machine learning models. By presenting our interactive framework to developers of popular open-source software and other security experts, we gather feedback on our trained models and further improve the features to increase the accuracy of our predictions. By providing qualitative (based on community feedback) and quantitative (based on prediction accuracy) evidence from 21 open-source programs, we show that our severity prediction framework can effectively assist developers with assessing vulnerabilities.

CCS CONCEPTS

• Security and privacy → Vulnerability management; **• Software and its engineering → Software testing and debugging**;

KEYWORDS

vulnerability assessment, software testing, symbolic execution, compositional analysis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MASES '18, September 3, 2018, Montpellier, France
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5972-6/18/09…$15.00
https://doi.org/10.1145/3243127.3243130

ACM Reference Format:
Saahil Ognawala, Ricardo Nales Amato, Alexander Pretschner, and Pooja Kulkarni. 2018. Automatically Assessing Vulnerabilities Discovered by Compositional Analysis. In *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis (MASES '18), September 3, 2018, Montpellier, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3243127.3243130

1 INTRODUCTION

Due to an increase in the size, features, and complexity of software applications coupled with increased automation in the hands of professional bug hunters, we have seen an explosion in the number of vulnerabilities exposed in popular software. Automated software testing is the preferred way for early detection of bugs in programs leading to vulnerabilities. Popular vulnerability scanners, however, report many, so-called, *false positives* [37] that may never materialize in a real-world usage of the program or associated components. In addition to expert knowledge, this calls for vulnerability assessment techniques for reported vulnerabilities that take into account the context of development, usage, underlying assets and the likelihood of exploitation [10].

One such automated testing and vulnerability discovery tool is Macke [29]. Macke is a compositional analysis tool based on symbolic execution, that achieves higher instruction coverage and discovers more potential vulnerabilities in many open-source programs than forward (simple and without compositional analysis) symbolic execution tools, such as KLEE [7]. The basic idea behind Macke is symbolic execution of isolated components in a program, summarizing vulnerabilities found in them and, performing a reachability analysis for the discovered vulnerabilities to generate exploits. However, when exploits cannot be generated (potential false-positives), Macke reports vulnerabilities without providing any contextual information to help developers prioritize the fixing process for reported vulnerabilities.

Fortunately, the Common Vulnerability Scoring System (CVSS) [33], a standard that is being rapidly adopted by the IT industry, is an existing system to rate the severity of a vulnerability and, hence, prioritize them. CVSS scores vulnerabilities by combining some properties of a vulnerability through empirically derived parameters.

In this paper, we will present a data mining and machine learning based technique for correlating features of vulnerable components

discovered by compositional analysis in C programs and predicting CVSS3 base-score values.

Problem: The state-of-the-art vulnerability scanners do not satisfactorily report found vulnerabilities. Static analysis tools, such as Splint, report too many false-positives. Compositional analysis tools, such as Macke, report fewer vulnerabilities (and, hence, possibly fewer false-positives) than static analysis tools but, in the absence of contextual information, it is difficult to triage bugs and prioritize reported vulnerabilities for which no exploit could be generated.

Solution: We collect a set of vulnerabilities reported in the past, with CVSS3 scores, and the corresponding versions of the programs affected by them. We compositionally analyze these programs with Macke and process the results to extract some features from them. Then, using scores of the collected set of past vulnerabilities as ground truth, we apply basic machine learning techniques to learn a model that can predict CVSS3 scores for vulnerable components from the features. We present the vulnerabilities with predicted severities to security and software testing experts and gather feedback on our prediction framework. Using this feedback, we add more features and obtain new models that predict severities for reported vulnerabilities with higher accuracy.

Contribution: Closing a gap in this field, we present a novel strategy to utilize the contextual information provided by compositional analysis that affects the severity of discovered vulnerable functions in C programs. Building on the empirical evidence that compositional symbolic execution reveals more vulnerabilities than forward symbolic execution [29], we state and evaluate the claim that it also results in relevant information for prioritizing vulnerabilities, such as ease of exploitation and the extent of damage that a vulnerability might cause if exploited. To the best of our knowledge, ours is the first framework to automatically assess severity of vulnerabilities reported by compositional symbolic execution, and evaluate it based on the accuracy of assessment and feedback from the open-source community.

This paper is structured as follows – section 2 gives an overview of the background to our work. In section 3, we describe our research methods and implementation of our prediction framework. In section 4, we list and describe the evaluation criteria for our methods, results and their interpretation and threats to validity of our experiments. Section 5 lists some past works related to ours and in section 6 we conclude the paper.

2 BACKGROUND

2.1 Symbolic Execution

Symbolic execution was introduced as a technique for software testing by King [19]. It is a deterministic method that uses instrumentation to dynamically collect constraints representing branching conditions in a program and solves these path constraints to generate inputs that execute the corresponding paths in the program. A path constraint (or *path condition*) may be defined as an ordered sequence of conditional branches that a program's execution takes to reach from an entry point (e.g. `main` function) to an exit point (e.g. `return` statement) of the program. Some of these paths may lead to unhandled exceptional behaviour in the program (such as buffer-overflows). Symbolic execution, and its practical approaches such as *concolic execution* [34] and *whitebox fuzzing* [15], have been shown [15] to be capable of extracting complex path-conditions for edge-cases and exceptions where other methods, like random testing, have failed to find potential vulnerabilities.

However, symbolic execution and its variants suffer from bottlenecks of underlying constraint solvers [14] and path-explosion [8], which leads to a major degradation of its performance in terms of both, path-coverage and vulnerability discovery.

2.2 Compositional Analysis with Macke

Compositional analysis or compositional symbolic execution has been proposed [11, 29, 32] as a mitigation strategy for the path-explosion problem in simple symbolic execution. The basic idea of compositional analysis is as follows – instead of symbolically executing the full program, we symbolically execute all components, such as functions, that can be executed in isolation. Then, we only focus on the inter-compositional interactions of these components, by means of directed symbolic execution [23]. In this way, the symbolic execution engine would not have to deal with all those paths that do not constitute any communication-related instructions between the components. Macke [29] is such a compositional analysis tool for C language programs, where the components are functions. We will, briefly, formally describe the working of Macke.

All the following description applies to all function, f, in the program, P, that are *executable* and we assume that there is at least one such function in P. An executable function is defined as a function that can be called *with arguments* from another function in the program. Let, the set of all executable functions in P be F_P. An executable function, f, may be represented as an *execution tree* that characterizes the possible paths followed during a symbolic execution of the function, as described by King [19]. If the list of arguments to the function, f, is $I = \{\alpha_1, \alpha_2, \ldots \alpha_n\}$, then a path-condition, pc, is a Boolean expression over α_i's. For every symbolic execution of a function

$$pc_{initial} = True \tag{1}$$

Whenever a branching statement, e.g. `If-else`, with branching condition, q, is encountered pc is extended as follows

$$pc = pc \wedge r \tag{2}$$

where, $r = q$ or $\neg q$.

Let the set of all pc's symbolically executed in an isolated function, f, be PC_f [1]. By the end of every execution, a pc will represent a path in f that ends due to (1) `return` statement, or (2) program crash, or (3) exceptional condition, such as an assertion failure. Macke considers case (2) as a segmentation fault occurring due to *buffer-overflow* vulnerability and case (3) as a special case of *assertion-failure* vulnerability. In this paper, we will only focus on buffer-overflows. Let the subset of PC that corresponds to all buffer-overflow vulnerabilities in f be PC^{vuln}. The corresponding arguments that execute the paths in PC^{vuln} can be considered exploits for these vulnerabilities.

[1]Note that PC_f does not contain a set of all *possible* paths in f, but only those that were symbolically executed.

After symbolically executing all functions, f, in isolation Macke performs directed symbolic execution to confirm the reachability of vulnerable paths via function calls.

For describing directed symbolic execution and reachability let us first define a function, $parents$. For any two isolated functions, $f_1, f_2 \in F_P$

$$\forall f_1, f_2 \in F_P : f_1 \in parents(f_2) \iff f_1 \textbf{calls} f_2 \tag{3}$$

I.e. The $parents(f)$ function lists all functions that may *potentially* call f.

Now, let $f_1 \in parents(f_2)$ and the set of vulnerabilities discovered in f_2 be $PC_{f_2}^{vuln}$. Then, a vulnerability in f_2 is said to have *infected* function, f_1 if

$$\exists pc_{f_1} \in PC_{f_1}, \exists pc_{f_2} \in PC_{f_2}^{vuln} : pc_{f_1} \wedge pc_{f_2} \tag{4}$$

Using a directed search strategy [23, 24], Macke reduces the set of paths by removing those paths in PC_{f_1} that do not satisfy the relation in equation (4), thereby reducing path-explosion. Additionally, it reports those vulnerabilities in isolated functions that cannot be found by forward symbolic execution of the *main* function but, nonetheless, *might* be reproducible through some functions in $parents(f)$. The above properties of Macke will be used for machine learning, as described in section 3.

2.3 CVSS3

The first version of Common Vulnerability Scoring System (CVSS) [25], introduced in 2003, proposed a way to capture the principal characteristics of a vulnerability and assign a numerical score reflecting its severity. The latest version to be used is version 3.0, or simply CVSS3. CVSS3 [33] consists of a base-score and optional temporal and environmental scores. In this paper, we will only focus on predicting the base-score. CVSS3 base-score is made up of the following metrics, which can take one of the corresponding values

- *Attack Vector (*`AV`*)* – The context in which exploiting the vulnerability is possible. **Allowed values:** *network*, *adjacent*, *local* and *physical*.
- *Attack Complexity (*`AC`*)* – The complexity of the attack process, if possible. **Allowed values:** *low* and *high*.
- *Privileges Required (*`PR`*)* – The level of privileges an attacker must have to carry out an exploit. **Allowed values:** *none*, *low* and *high*.
- *User Interaction (*`UI`*)* – The amount of direct user interaction required for the attacker to carry out an exploit. **Allowed values:** *none* and *required*.
- *Scope (*`S`*)* – Whether or not other components (changed scope) than the vulnerable one can be affected if the vulnerability is exploited. **Allowed values:** *unchanged* and *changed*.
- *Confidentiality (*`C`*)* – The amount of confidential data that will be exposed if the vulnerability is exploited. **Allowed values:** *none*, *low* and *high*.
- *Integrity (*`I`*)* – How much information can the attacker modify in the exploited component. **Allowed values:** *none*, *low* and *high*.
- *Availability (*`A`*)* – Can the attacker deny access to the exploited component and whether that component is critical. **Allowed values:** *none*, *low* and *high*.

In section 4, we will see how effectively we can predict all of the above CVSS3 base-score metrics.

Comparison with Bugzilla severity. The popular bug-reporting platforms such as [1] use *Bugzilla*'s nominal categories for prioritizing bug fixes. However, in past works [5] and from our own experience, we note that the context-free ranking system of Bugzilla results in an order that doesn't truly represent the severities of bugs. The first reason for this is that, for most of the bugs analyzed by us (section 4), the development community ignored the "priority" field of the reports, and used only the "severity" field as a proxy for both, priority and severity. Unlike Bugzilla, for calculating CVSS and CVSS3 scores, evaluation *all* base-score values is mandatory. Secondly, we also found many instances in the analyzed programs where the severity values in Bugzilla were changed by the developers when, either, the underlying assets were not considered important enough, or the bug would not be fixed because it was too complex to exploit it. However, CVSS base-scores explicitly take into account underlying assets and attack complexity, thereby eliminating the need for further arbitrary adjustment.

3 METHODOLOGY

In this section, we will give a high-level overview of the implemented methods, including data collection, programs' compositional analysis, initial feature extraction, the first stage of machine learning, interactive feedback gathering and machine learning with combined features.

In the following subsections, we will use the running example of *Autotrace 0.31.1*, a UNIX command-line-based program to convert bitmap image formats to a vector graphics format.

3.1 Data Collection

We, firstly, require a set of known vulnerabilities with CVSS3 base-scores to learn a predictive model. For this study, we only consider *buffer-overflow vulnerabilities*. Because of reasons listed in section 2.3, Bugzilla repositories, such as [1], were not suitable for our study. The National Vulnerability Database (NVD) [2] contains analyses of Common Vulnerabilities and Exposures (CVEs) by aggregating program and vulnerability description, vulnerability type enumeration, applicability statements, impact metrics (in CVSS or CVSS3 format) and other relevant references.

After selecting NVD as our preferred database, we needed to filter the data for reports

1. that follow CVSS3 notation, instead of CVSS 1.0,
2. about C programs,
3. about buffer-overflows, and
4. that state the name of the vulnerable function.

Setting the filters on the data collected from NVD, as described above, we were left with a set of vulnerability reports that were thorough in their description and had CVSS3 scores attached to them. The next step is to, manually or automatically, determine the name and version of the affected program and download the

source-code of it. In section 4.1, we will present the size of dataset and ground-truth for machine learning.

3.2 Compositional Symbolic Execution

As described in section 2.2, Macke performs compositional symbolic execution by symbolically executing isolated functions in a program and, then, performing a reachability analysis for the reported vulnerabilities from parent (calling) functions. In this step of our methodology, we perform compositional analysis on the programs collected in section 3.1, using Macke.

The output of Macke includes a JSON file that lists

(1) all discovered vulnerabilities,
(2) source-code location of the vulnerable instruction, and
(3) functions through which an identical vulnerability may be exploited.

The results of the analysis with Macke are presented in section 4.2. In addition to the above list, Macke also outputs a *call-graph* of the program. The call-graph for Autotrace 0.31.1 is shown in figure 1. The vulnerable functions, as discovered by Macke, in this graph are highlighted.

3.3 Feature Extraction

The output of running Macke on the candidate programs are processed in this step to extract some features related to the vulnerabilities and vulnerable functions. We will now describe these features and our intuition behind including them as possibly correlating factors for predicting the severity of vulnerabilities. Please note that in the following list the terms "nodes" and "functions" are used interchangeably. Also, we assume that each function in the call-graph has an equal likelihood of sanitizing an input (argument).

(1) *Node degree (*`d_in`*,* `d_out`*)*, defined as the number of callers or callees (called, henceworth, also as *neighbours*) for a function in the call-graph. As explained by El Emam et al. [13], and Nagappan et al. [28], node degree is an important feature because the higher the node's degree, the more likely that a vulnerability in it may infect other functions, thereby leading to a failure in the program. For the function `rle_fread` (figure 1) the values of incoming node degree, `d_in`, and outgoing node degree, `d_out`, are 1 and 3, respectively. For function, `std_fread`, `d_in` and `d_out` are 1 and 1, respectively.
(2) *Distance to interface (*`di`*)*, defined as the length of the shortest path from an interface (such as `main` function) to the function. The shorter the distance from an interface, the less likely it is that a pointer argument was sanitized before being accessed. The value of `di` for `std_fread`, as seen from figure 1 is 3, while it is 2 for `ReadImage`.
(3) *Clustering coefficient (*`cc`*)*, defined as the ratio of neighbouring functions of a node that are also mutually connected (as caller-callee pair). The intuition behind this feature is that the bigger a *cluster* in which a vulnerable function is, the more likely it is that this vulnerability may be exploited by another function in the cluster. From figure 1, we can see that `rle_fread` has 4 immediate neighbouring functions (callers or callees). Out of 6 possible pairs of neighbours of `rle_fread`, 3 are also connected to each other in a caller-callee relationship. Therefore, the value of `cc` for `rle_fread` is 0.5.
(4) *Node path length (*`nl`*)*, defined as the average number of steps that need to be taken to reach any reachable node from the function being analyzed. If the node path length for a node is low, it denotes a higher likelihood that an argument leading to buffer-overflow is *not* being sanitized before being passed between functions, assuming every function sanitizes input with the same likelihood. From figure 1, we can see that there are three functions that are reachable from `rle_fread` and their distances from `rle_fread` are 1, 1 and 1, respectively. Hence, the value of `nl` for `rle_fread` is 1 (average of 1, 1 and 1). In our implementation, we detect loops in the call-graph and stop counting nodes when we encounter a loop (e.g. in recursion).
(5) *Vulnerabilities discovered (*`nv`*)* depends on the output of Macke, and it denotes the number of unique vulnerable instructions discovered by Macke. A higher number of buffer-overflow vulnerabilities indicate [27] that an argument might be assumed to be sanitized by a calling function, but such a sanitization never took place in reality. In the functions, `rle_fread` and `std_fread`, Macke discovered 1 vulnerable instruction each, i.e. `nv` is 1. However, for `ReadImage` (figure 1) Macke found the instructions in, both, `rle_fread` and `std_fread`, to be exploitable. Therefore, `nv` is 2 for `ReadImage`.
(6) *Maximum length of infection (*`li`*)* depends on the results of compositional analysis. It is the longest chain of *caller-callee* pair through which the same vulnerability was discovered by Macke. The intuition behind this feature [29] is that if the same vulnerability can be exploited in a long chain, then it is more likely that a sanitization on passed arguments was not performed. We can see from the call-graph in figure 1 that the underlying "causes", or unsafe operation, for the vulnerabilities in `ReadImage` function are in `std_fread` and `rle_fread`. In both cases, the maximum length of infection, `li`, is 2 ($ReadImage \rightarrow std_fread$ or $ReadImage \rightarrow rle_fread$).

3.4 Prediction of CVSS3 Base Scores

3.4.1 Preparation of data for prediction models. With the ground truth and features related to functions and vulnerabilities, the next step is to train machine learning models to learn the correlation between these features and CVSS3 base-scores and use a linear or non-linear combination of the features to predict these base-scores.

We consider each CVSS3 base-score value as individual targets for prediction and generate the final CVSS3 severity score, $severity$, based on the formula in the original specification document of CVSS3 [33], $calc_cvss3$. i.e.

$$severity = calc_cvss3(y_{AV}, y_{AC}, y_{PR}, y_{UI}, y_S, y_C, y_I, y_A) \quad (5)$$

where the subscripts, $base$, are the CVSS3 base-scores described in section 2.3, Concretely,

$$y_{base} = f^L_{base}(V) \quad (6)$$

where,

$$V = [d_in, d_out, di, cc, nl, nv, li] \quad (7)$$

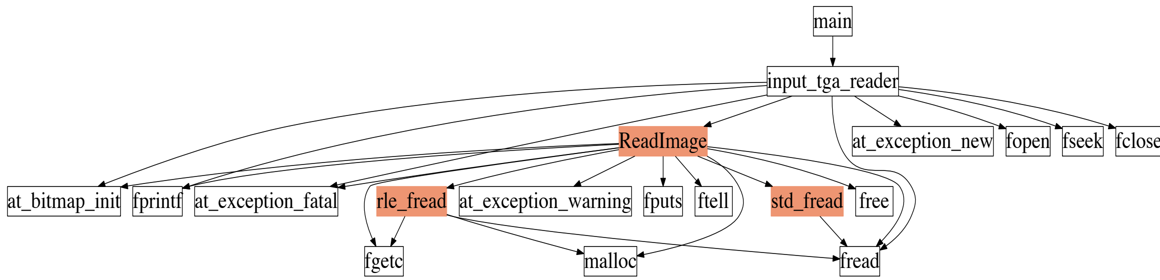

Figure 1: Call-graph of Autotrace 0.31.1 program to convert a TGA bitmap to vector graphics format

The superscript, L, in equation (6) stands for "learned", denoting the learned model, f^L_{base}. The $[\dots]$ in equation (7) represents a vector of feature values.

3.4.2 Machine learning models. We apply two standard machine learning algorithms to learn functions, f^L, viz.

(1) Random-forest classifier,
(2) Naive Bayes classifier,

We use scikit-learn [31], a widely used machine learning toolkit in Python. For all models, we apply K-fold cross-validation on the training dataset. We, then, apply the model with best validation score on the test dataset to calculate the test scores, which are reported in section 4.

For presentation to the experts for gathering feedback, we apply the best of all machine learning models (based on test scores) to predict scores for those vulnerabilities that were discovered by Macke but were previously unreported.

3.5 Presentation and Feedback Gathering

3.5.1 Interactive reporting of vulnerabilities. After learning predictive models for CVSS3 base-score values we present the predictions to security and software development experts in an interactive medium to obtain feedback from them. The requirements for such a presentation medium are

(1) Ability to interact with the call-graph, including zooming in on functions and viewing the source-code.
(2) Ability to view CVSS3 base-scores and aggregate values, by clicking on the function node.
(3) Ability for the user to change base-score values, using a numerical text-box. The aggregate CVSS3 value must be updated automatically.
(4) Tracking all the above interaction, including when a function node is clicked, source-code is expanded and a base-score value is updated.
(5) Ability for the user to send textual feedback, using a multi-line text-box.

We created a web-application on a server running NodeJS, with a ReactJS frontend. The resultant interface, satisfying all the requirements, for Autotrace program is shown in figure 2.

3.5.2 Feedback from experts. The goal of this step is to learn from our target audience how effective a severity prediction tool

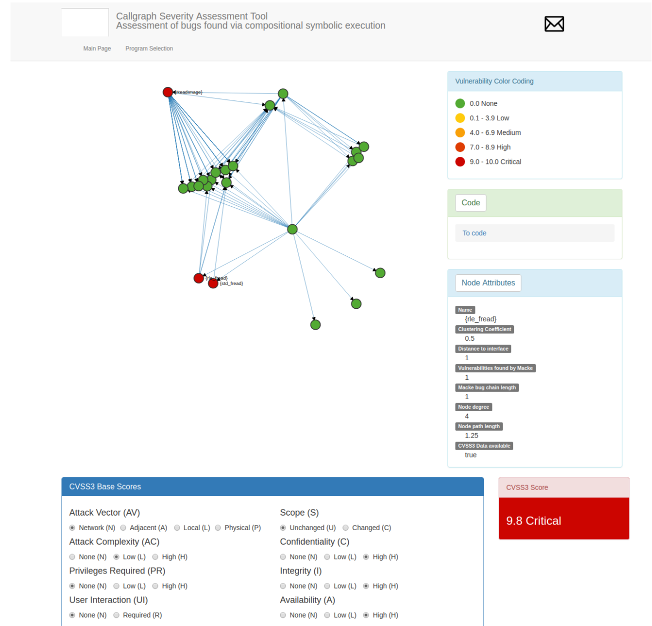

Figure 2: Severity assessment interface for Autotrace 0.31.1 with interactive call-graph

would be in improving the manual task of prioritizing vulnerabilities. We provide a link to the online tool to selected experts listed in section 4.4 and show the predicted CVSS3 base-score values and the aggregate score, for previously reported and unreported vulnerabilities. However, the distinction between previously reported and unreported vulnerabilities is invisible to the experts, so that their feedback may be validated. For previously unreported vulnerabilities, if the experts disagree with the predicted values, we ask for their reasons for disagreement, so that we can improve our prediction framework. From the experts that choose to participate in our survey, we collect the following information

(1) Which function node was expanded (by clicking on it, as shown in figure 2).
(2) Function nodes for which any CVSS3 base-score value was changed (if an expert disagreed with the predicted value), along with the old and new values,
(3) Function for which the expert referred to the source-code,

(4) Textual feedback, when the expert wanted to provide feedback or clarifications,
(5) (optional) expert's full name and email if they agreed to be contacted by us.

Above information is stored in a MySQL backend and processed manually by us at the end. The qualitative analysis of received feedback will constitute our first measure of effectiveness.

3.6 Feature Addition

Based on the feedback received from experts, we add more features to extend the predictive power of our machine learning models. The new features added and the intuition behind including them are as follows

(1) *Function size* (`s`), a simple count of LLVM [21] instructions in the function. The intuition behind this feature is that a higher number of instructions might indicate that, instead of splitting the functionality across various single-purpose function, it exists in a singular function. Vulnerability inside such a function that is not interacting with, and hence doesn't depend for the sanitization of its inputs on, other functions of the program should be fixed with a high priority.
(2) *Approximate function complexity* (`fx`), a count of LLVM *basic-blocks* in the function. A basic-block [16] is defined as a straight-line of instruction-sequence that contains no branches inside it, other than the entry or exit points of the block. Nagappan et al. [28] claimed and showed with case studies that the number of basic blocks and arcs in CFG, which may be an approximation of McCabe's complexity measure, correlate well with the possibility of a module's failure in many projects. In our case, a higher function complexity (in terms of basic-blocks) may indicate a higher likelihood that a vulnerable instruction was left unhandled unintentionally.
(3) *Pointer parameters* (`pt`), the number of parameters that a function accepts that are of pointer type. Pointers are important for a vulnerable function because we are dealing with buffer-overflow vulnerabilities only. If more pointer parameters are specified, then there is a higher likelihood that access to at least one of them might be in an unsafe manner.

After adding these features to the existing features, we learn the new predictive functions, f^L, for all CVSS3 base-score values, using the same machine learning algorithms as listed in section 3.4. The accuracy of the final learned model, on previously reported vulnerabilities, constitutes the second effectiveness measure of our tool.

4 EVALUATION

Based on the methodology of our study, as described in the previous section, we will now discuss the experiments carried out for evaluating the effectiveness of our prediction framework. The raw-data related to all stages of evaluation has been made public by us at [3].

Table 1: Programs analyzed and vulnerabilities in them

Program and Version	LOC	Connected functions	Vulnerable functions from NVD, (with CVSS) $\|\|N\|\|$	Vulnerable functions CVSS scored manually, $\|\|M\|\|$	Vulnerable functions found by Macke, $\|\|X\|\|$
BlueZ 5.42	286,206	49	3	2	6
AutoTrace 0.31.1	18,581	23	3	0	3
GraphicsMagick 1.3	324,422	22	4	4	10
Icoutils 0.31.1	40,093	45	2	3	5
ImageMagick 6.0.4-8	476,747	51	1	3	8
Jasper 1.900.27	46,578	33	3	4	19
Jasper 2.0.10	46,622	33	2	3	6
Libarchive 3.2.1	204,993	62	1	4	15
Libass 0.13.3	18,745	46	1	3	29
Libmad 0.15.1	12,866	22	1	1	4
Libplist 1.12	6,075	69	1	5	27
Libsndfile 1.0.28	85,189	153	1	3	40
Libxml2 2.9.4	334,796	36	2	3	22
Lrzip 0.631	18,622	115	1	1	7
Openslp 2.0.0	55,545	27	1	3	17
Potrace 1.12	12,928	28	1	3	13
Rzip 2.1	2,651	34	1	3	19
Tcpdump 4.9.0	103,152	13	1	1	7
Tiff 4.7.0	82,725	125	3	2	6
Virglrenderer 0.5.0	57,213	70	2	0	21
Ytnef 1.9.2	4,818	70	6	1	12
		Total	**41**	**52**	**296**

4.1 Vulnerability Reports and Affected Programs

The *21 open-source programs* analyzed in this study are listed in table 1. The second column in table 1 lists the lines of source-code in the programs. The analyzed programs range from, approximately, 2,000 to 470,000 lines of code, with an average of, approximately, 100,000 lines of code

The third column in table 1 lists the number of *connected* functions in the analyzed programs. We define connected functions as the functions that are (potentially) reachable, as determined by a simple syntactic analysis, from an entry point in the program. Many of the analyzed programs have several direct entry points (such as `main`) or public APIs. In this study, for every program, we only choose the connected component that contains a function which is present in one of the collected vulnerability reports (from NVD).

The fourth column in table 1 lists the number of vulnerable functions (functions containing at least one reported vulnerability) reported in the respective program in NVD. These vulnerability reports are obtained after applying all the filters described in section 3.1. The number of CVSS3 scored vulnerable functions, as we can see from table 1, is not large enough to effectively train any machine learning algorithm. Therefore, to augment this list obtained from NVD, we manually score some more vulnerable functions, listed in the fifth column of table 1, that are discovered by Macke.

4.2 Analysis with Macke

After collecting the vulnerability reports and the respective affected versions of the programs, the next step is to apply Macke on these programs to find the vulnerabilities including ones listed in section 4.1.

The sixth column in table 1 lists the number of vulnerable functions discovered by Macke in approximately *30 minutes*. We can see that the number of unique vulnerable functions found by Macke, 296, is more than the reported vulnerable functions in NVD. After carefully analyzing the results, we found that *all vulnerabilities reported in NVD* were also found by Macke within the given time-limit. The extra vulnerabilities and vulnerable functions may or may not be true positives.

4.3 Feature Extraction and Machine Learning

After running Macke on the programs to be analyzed, the next step is to extract features from the programs and their analyses results, as described in section 3.3, and learn our prediction models using them. The *ground truth* to be used for machine learning is $G = N \cup M$, where N and M are as shown in table 1.

As described in section 3.4, we trained two machine learning models using features listed in section 3.3. The dataset for learning, G, is split into training (75%) and testing (25%) sets. The training set is split for 4-fold cross-validation, i.e. 4 machine-learning models are trained by holding out each fold one-by-one, and the best model chosen. Because of the small training set, we chose only 4 folds for cross-validation, instead of, say, 10 folds that is more common in many papers. To remove the effect of random initial states, for all iterations, we train 10 models generated with different seeds and perform majority voting for predicting base-score values. We calculate the *accuracy measure* for all base-scores values, which is the ratio of correct predictions of the predictions made. The results of training machine learning models are shown in table 2. The accuracy metric listed is on the testing dataset and the best accuracy scores for every base-score value is listed in bold-face text.

Table 2: Prediction results on test dataset – with original features (section 3.3) only

	Random Forest	Naive Bayes
AV	**0.59**	0.27
AC	0.55	**0.59**
PR	0.91	**0.95**
UI	**0.73**	0.45
S	**0.91**	0.91
C	**0.64**	0.45
I	**0.55**	0.27
A	**0.82**	0.55

We can see from these results that, except for *attack complexity* and *privileges required*, random-forest classifier performs the best in terms of test accuracy scores. Even though some of the accuracy scores, such as for *attack vector*, *confidentiality impact* and *integrity impact* seem to be low (0.59, 0.64 and 0.55 respectively), we don't consider them too bad because these base-score values may be in one of 4, 3 and 3 classes respectively.

Table 3: Summary of feedback received by experts

Expert ID	Programs analyzed	Functions expanded	Comments left
1	1	1	1
2	1	4	1
3	1	1	1
4	1	1	1
5	1	2	2
6	1	8	4
7	3	6	3
Unique	**5**	**20**	**13**

4.4 Feedback from Experts

The best models obtained from the learning phase are used to predict CVSS3 base-scores for the vulnerabilities in the set $(X - G)$ (previously unreported vulnerabilities) and the final CVSS3 scores for them calculated using the equations presented in [33]. For getting feedback, we contacted (1) developer mailing-lists of the programs analyzed, (2) students of "Security Engineering" lecture course, who had sufficient background in secure software development principles and symbolic execution, and (3) two members of technical staff at our organization, one of whom has a doctoral degree in a security-related field. The call-graphs, respective CVSS3 base-scores, and final scores are, then, shown to the experts using the interface described in section 3.5.

In table 3, we have summarized the feedback received from the experts that we contacted. The second column in table 3 shows the number of programs that an expert analyzed. The third column shows the number of *unique* functions in the programs for which an expert clicked to either, expand to view the source code or, only view the assigned CVSS3 base-scores. The last column in table 3 shows the number of comments or feedback items left by the respective expert during the entire exercise. The last row of this table lists the number of *unique* analyzed programs, functions and feedback items received by us. In listing 1, we have presented verbatim some of these feedback items. We have omitted some comments from this paper because they were either, identical to the feedback items shown, or were unrelated to bug triage, e.g. "you must examine the latest version of this program because some vulnerabilities were fixed later."

Listing 1: Some feedback from experts

```
Program: Jasper 2.0.10
Functions selected: jpc_dec_decodepkt
Comment:
Without pinpointing the vulnerable instruction, the
function is very hard to analyze manually. Looking
at the size of this function (250 lines), it might be a
good idea to keep the score high, because it's likely
to be reused somewhere. I also think the signature
of the function suggests the incoming parameters are
very varied and, hence, might be prone to being unsanitized

Functions selected: jpc_dec_decodepkt, main, jpc_dec_lookahead
Comment:
The tool looks great, but it would be really useful if
for each function you would also indicate the number
of the LOC where the buffer overflow vulnerability
occurs. Otherwise, for large functions, it is difficult
to pinpoint the vulnerability manually. Of course, it is
also easier to analyze the code of commented functions
in comparison to functions without any comments.
```

```
Program: Rzip 2.1
Functions selected: read_buf, write_u16,
      BZ2_bzBuffToBuffCompress, write_buf
Comment:
All OK.

Functions selected: read_u8
Comment:
All is OK but for this file I'm not sure the result of
      confidentiality

Functions selected: read_stream, write_stream, write_u32
Comment:
I think the file is used locally.

Program: Libass 0.13.3
Functions selected: ass_pre_blur1_vert_c
Comment:
To me this function does not seem to be exploitable via the
      network.

Program: ImageMagick 6.0.4-8
Functions selected: ReadRLEImage
Comment:
Here it looks to me like those code will be exploitable via the
       network as imagemagic is often used to parse network data
```

Synthesizing Feedback. As a qualitative measure of *effectiveness*, we wanted to know if our framework successfully helped the experts assign severity to vulnerabilities. We could distill the following main points from the feedback received from experts who used our prediction framework

(1) Most experts found the tool to be useful.
(2) Triage process is significantly affected by the size of the source-code being analyzed.
(3) In the absence of relevant comments, the perceived severity of functions is affected by how "complex" it is.
(4) The perceived severity of a vulnerable function, somehow, depends on what parameters are passed to it.
(5) Experts would prefer pinpointing of the vulnerable instructions, rather than only the affected function.
(6) At least one expert was suspicious of the confidentiality impact score assigned to a function.

4.5 Machine Learning from Improved Features

Based on the feedback received from experts, we add more features, expecting to increase the accuracy scores for all base-score values. The added features, as listed in section 3.6, are – number of LLVM instructions in the function, number of basic-blocks in the functions and the number of function parameters that are of pointer type. Using these three additional features, we, now, train the same machine learning models as in section 4.3, i.e. 4-fold cross-validated random-forest classifier and naive Bayes classifier. The accuracy scores on the test set after adding these three features to the existing features (total *ten* features) are shown in table 4. The results show that, after including three more features the random-forest classifier and naive Bayes classifier were, both, able to more accurately predict *most of* the CVSS3 base-score values of the ground truth. The best scores, as we can see from table 4, were all found with random-forest classifier. Especially notable is the results that, with new features based on the feedback received from security experts and developers, *privileges-required* (`pr`) and *scope change* (`sc`) could be predicted with 100% accuracy in the test dataset. *User-interface* (`ui`) could also be predicted with an almost-perfect accuracy.

Table 4: Prediction results on test dataset – with original and added features (section 3.6)

	Random Forest	Naive Bayes
AV	**0.64**	0.50
AC	**0.82**	0.55
PR	**1.00**	0.95
UI	**0.95**	0.95
S	**1.00**	0.95
C	**0.91**	0.91
I	**0.73**	0.50
A	**0.91**	0.82

4.6 Interpretation of the Results

Based on the various experiments conducted by us, we will now take a big-picture view of the results obtained.

4.6.1 Effectiveness of Prediction. From tables 2 and 4, we can see that, while some CVSS3 base-score values could be predicted by our framework with high accuracy, there are others for which the framework does not perform reasonably well. We want to stress in this work that we don't claim that *all features* of extracted functions may correlate with *all base-score values*.

It is, perhaps, not surprising that the accuracy in predicting *attack vector* and *integrity impact* was not as high as, say, *availability impact*. This is because attack vector of a reported vulnerability describes the means through which a system may be attacked by an outsider. This base-score value may only be predicted based on more detailed information about the deployment, and usage, if at all. Similarly, integrity impact describes the effect on the overall integrity of the data managed, manipulated or protected by the system and it is difficult to predict it based only on features of a function. However, intuitively, the high accuracy in predicting base-score values of *attack complexity* or *availability impact* can be explained, respectively, by taking into account complexity of a function (section 3.6) [13] and the effect that a buffer-overflow vulnerability would have, if exploited, i.e. program crash. This is clearly reflected in our results.

Therefore, we can claim by looking at our results that our chosen features can be used to assign correctly *most* of the base-score values with a high accuracy for previously reported bugs. However, for other base-score values, where the accuracy of prediction is not as high, we should use other sources, such as function or requirements specifications, or even manual intervention for increasing effectiveness assessment.

4.6.2 Effectiveness of Overall Framework. The comments, as listed in listing 1, indicate that most experts who used our tool were satisfied by the format of the tool and agreed with the predicted values for base-scores of previously unseen vulnerabilities. Some of the feedback, as seen in listing 1, was not concerned with the features of the analyzed programs and functions but instead with the presentation of the results. Even though these comments could not be used to enrich the list of features, we used them to improve our interactive tool. For example, we extended it to include and highlight the precise lines of code where a potential buffer-overflow may occur.

Therefore, by qualitatively analyzing feedback received from the experts, we can claim that such a tool for predicting CVSS3 severities can effectively aid vulnerability assessment.

4.6.3 Adaptability of Features. Our framework depends on effective discovery of vulnerabilities by a reliable tool. We chose Macke for this, due to the following reasons. Firstly, Macke can find *more* vulnerabilities [29] in isolated functions, than forward symbolic execution. It also generates fewer false-positives than static-analyzers, such as *Splint*. Secondly, Macke outputs "maximum length of infection" (`li`), that we discussed in section 3.3. According to an ad-hoc analysis of features, this is one of the most highly correlated features to most CVSS3 base-score values and, therefore, is crucial in increasing the accuracy of predictions.

However, an important feature of our tool is that it does not depend on the technique used to discover the vulnerable functions. All the features used for machine learning, except `li`, can be just as easily extracted using any static analyzer or even a manual code review. A comparison of the effectiveness of severity assessment based on different vulnerability scanners is left as a future work.

4.7 Threats to Validity

We will now discuss some threats to validity of our results, especially in the context of generalization to other programs or programming languages.

4.7.1 Subjectivity of Assessment. The subjectivity of existing CVSS3 scores directly affected our methodology. The assignment of these base-scores may not be objective or reproducible when the same vulnerabilities are presented to two different experts. This would be detrimental because we have assumed that our ground truth always holds. We counter this threat by including 21 different programs in our analysis, which introduces variation in factors influencing the manual process.

4.7.2 Subjectivity of Feedback. Similar to the ground truth for machine learning, there is also subjectivity in feedback from the experts who participated in the study. Since none of the participants analyzed every program, we needed a way to balance their opinions based only on the programs that they analyzed. The measure that we took to counter this threat is to manually interpret the experts' feedback and use a distilled list of feedback items for feature addition.

4.7.3 Randomization. The machine learning results (accuracy) may not be reproducible with the same initial training and testing dataset, due to the randomization in the implementation of random-forest classifier and naive Bayes classifier in scikit-learn [31]. To counter this threat, we performed 10 runs, with different seeds, of every fold of 4-fold verification and obtain the results of prediction from a majority voting algorithm for every base-score value.

4.7.4 Criticism of CVSS. Another threat to validity is the scoring system of our choice, CVSS3. As discussed previously by Allodi and Massacci [4] and Holm and Afridi [17], CVSS scores are not always indicative of how bug fixing in real-world should be prioritized. However, as authors of both these papers admit, there aren't any competing vulnerability ranking measures that take into account as many factors for ranking severity of vulnerabilities as CVSS does and, hence, we chose it for our experiments.

4.7.5 External Threats to Validity. The last threat to validity is the external threat of variance in the *nature* of the analyzed programs. We have analyzed programs for which we could find reported vulnerabilities with CVSS3 scores in NVD. However, the common characteristics of these programs were that they were all open-source and maintained by members of a large and growing community with a different set of skills and expertise that we didn't account for. We claim here that the results obtained by us generalize only over the set of programs analyzed by us, but may not hold for other industrial software, embedded systems or proprietary real-world programs.

5 RELATED WORK

5.1 Automatic Bug Assessment

By observing correctly that most past works in severity prediction [9, 39, 40] have been concerned with text-mining from bug repositories, Bettenburg et al. [6] decided to investigate features that make a bug report *good*. They concluded in this paper that reports containing steps to reproduce bugs and stack traces are considered to be most useful. However, none of the related works analyzed by us make use of the stack traces (that can be automatically generated by executing the exploit) to predict severity. Lamkanfi et al. [20] compared text-mining algorithms and concluded that a naive Bayes classifier performs the best in terms of ROC measure when applied on the textual content of bugs reported in Eclipse and GNOME open-source projects. However, the vulnerabilities could be classified in one of two classes only, viz. *severe* or *not severe*. Research by Menzies and Marcus [26], and Chaturvedi and Singh [9] applied similar text-mining approaches on NASA's Project and Issue Tracking System (PITS), with promising results, where the severity scale ranges from 1 to 5, with 1 being most severe. Other severity prediction frameworks, such as [35, 36, 39, 40], have also focussed on the textual content of the bug reports. There have been only a few, to our knowledge, works that do severity prediction using source-code or any other intermediate representation (e.g. control-flow-graph and call-graph) of the system-under-test. Palomba et al. [30] presented a method for improving bug prediction in software components using code smells. El Emam et al. [13] presented a technique for early detection of components in the software that may be faulty, based on metrics derived from the object-oriented design. However, a discussion of the severity of predicted faults is not included in [13]. Some past works by Nagappan et al. [28], Nagappan and Ball [27], and El Emam et al. [13] have inspired the choice of function and call-graph features that we have used in our study to learn the severity of vulnerabilities in them.

5.2 CVSS Scale

Papers such as [18], by Houmb et al., use CVSS to predict the frequency and impact of failures from a risk management perspective. Dobrovoljc et al. [12] suggest an improvement over CVSS that explicitly includes the features of perceived attackers who may be able to exploit a vulnerability. Similarly, Wang et al. [38], and Liu and Zhang [22] also suggest new scales by pointing our certain

deficiencies in CVSS that restrict its usage in their contexts. Allodi and Massacci [4], and Holm and Afridi [17] present criticisms of CVSS's ability to quantify the severity of discovered vulnerabilities by pointing out that vulnerabilities that are reported with sample exploits can, generally, be a much better indicator of their severities.

5.3 Contribution

By analyzing the related works above our work tries to fill the gaps in research in the following ways –

(1) Our work treats the system-under-test as a set of interacting components and predicts the severity of vulnerabilities based on heuristics of the affected functions and their interactions with other functions.
(2) To the best of our knowledge, our framework is the first to correlate call-graph features and compositional analysis results to any severity measures of reported vulnerabilities, CVSS or otherwise.
(3) Our work also comments on whether certain CVSS base-scores are suitable to be predicted by only the syntactic properties of a program.

6 CONCLUSION

In this paper, we described a framework for automatically predicting the severity of vulnerable functions reported by a compositional symbolic execution tool. Using a systematic procedure, we collected data from NVD about vulnerabilities reported in the past with CVSS3 severity scores for C programs. For the collected vulnerabilities, we compiled and analyzed the same programs with Macke, a compositional analysis tool based on symbolic execution. From the results of programs' analyses, we extracted some features for training machine models to predict CVSS3 base-score values. The results from the best model were, then, presented to various experts in the field of secure software development to obtain their feedback on the tool and predicted base-score values. Based on the feedback received, we updated the list of features to be extracted from the programs and re-trained the machine learning models. Our evaluation showed that the accuracy of predictions with the updated list of features had an improvement for all CVSS3 base-score values. Using this empirical result and qualitative feedback from the community of experts, we have shown that our predictive framework can effectively help developers assess the severity of vulnerabilities reported by Macke.

REFERENCES

[1] [n. d.]. GNOME Bugzilla. https://bugzilla.gnome.org/.
[2] [n. d.]. National Vulnerability Database (NVD). https://nvd.nist.gov/.
[3] 2018. CVSS3 assessment using machine learning. https://osf.io/g87ny/?view_only=21bf55c48263432f9b00702f84b23711
[4] L. Allodi and F. Massacci. 2014. Comparing vulnerability severity and exploits using case-control studies. *ACM Transactions on Information and System Security (TISSEC)* (2014).
[5] P. Ayari, K.and Meshkinfam et al. 2007. Threats on building models from cvs and bugzilla repositories: the mozilla case study. In *Conference of the center for advanced studies on Collaborative research*. IBM Corp., 215–228.
[6] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. 2008. What makes a good bug report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*.
[7] C. Cadar, D. Dunbar, and D. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*.
[8] C. Cadar and K. Sen. 2013. Symbolic execution for software testing: three decades later. *ACM Communications* (2013).
[9] K. Chaturvedi and V. Singh. 2012. Determining bug severity using machine learning techniques. In *Software Engineering (CONSEG), 2012 CSI Sixth International Conference on*.
[10] M. Christakis and C. Bird. 2016. What developers want and need from program analysis: an empirical study. In *International Conference on Automated Software Engineering*.
[11] M. Christakis and P. Godefroid. 2015. IC-Cut: A compositional search strategy for dynamic test generation. In *Model Checking Software*.
[12] A. Dobrovoljc, D. Trček, and B. Likar. 2017. Predicting Exploitations of Information Systems Vulnerabilities Through Attackers Characteristics. *IEEE Access* (2017).
[13] K. El Emam, W. Melo, and J. Machado. 2001. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software* (2001).
[14] I. Erete and A. Orso. 2011. Optimizing constraint solving to better support symbolic execution. In *ICSTW*.
[15] P. Godefroid, M. Levin, et al. 2008. Automated Whitebox Fuzz Testing.. In *NDSS*.
[16] J. Hennessy and D. Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.
[17] H. Holm and K. Afridi. 2015. An expert-based investigation of the common vulnerability scoring system. *Computers & Security* (2015).
[18] S. Houmb, V. Franqueira, and E. Engum. 2010. Quantifying security risk level from CVSS estimates of frequency and impact. *Journal of Systems and Software* (2010).
[19] J. King. 1976. Symbolic execution and program testing. *ACM Communications* (1976).
[20] A. Lamkanfi, S. Demeyer, et al. 2011. Comparing mining algorithms for predicting the severity of a reported bug. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*.
[21] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO)*.
[22] Q. Liu and Y. Zhang. 2011. VRSS: A new system for rating and scoring vulnerabilities. *Computer Communications* (2011).
[23] K. Ma, K. Phang, et al. 2011. Directed symbolic execution. In *SAS*.
[24] R. Majumdar and R. Xu. 2009. Reducing test inputs using information partitions. In *CAV*.
[25] P. Mell, K. Scarfone, and S. Romanosky. 2007. A complete guide to the common vulnerability scoring system version 2.0. In *Published by FIRST-Forum of Incident Response and Security Teams*.
[26] T. Menzies and A. Marcus. 2008. Automated severity assessment of software defect reports. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*.
[27] N. Nagappan and T. Ball. 2005. Static analysis tools as early indicators of pre-release defect density. In *International conference on Software engineering*.
[28] N. Nagappan, T. Ball, and A. Zeller. 2006. Mining metrics to predict component failures. In *International conference on Software engineering*.
[29] S. Ognawala, M. Ochoa, A. Pretschner, and T. Limmer. 2016. MACKE: Compositional analysis of low-level vulnerabilities with symbolic execution. In *ASE*.
[30] F. Palomba, M. Zanoni, F. Fontana, A. De Lucia, and R. Oliveto. 2016. Smells like teen spirit: Improving bug prediction performance using the intensity of code smells. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*.
[31] F. Pedregosa, G. Varoquaux, et al. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* (2011).
[32] A. Pretschner. 2003. Compositional generation of MC/DC integration test suites. *Electronic Notes in Theoretical Computer Science* 82 (2003).
[33] institution = FIRST.Org, Inc Scarfone, K. and others. 2016. *Common Vulnerability Scoring System v3.0: Specification Document*. Technical Report.
[34] K. Sen, D. Marinov, and G. Agha. 2005. CUTE: a concolic unit testing engine for C. In *ACM SIGSOFT Software Engineering Notes*.
[35] H. Shen, J. Fang, and J. Zhao. 2011. Efindbugs: Effective error ranking for findbugs. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*.
[36] Y. Tian, D. Lo, and C. Sun. 2012. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*.
[37] M. Torchiano, M. Morisio, et al. 2010. Assessing the precision of findbugs by mining java projects developed at a university. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*.
[38] R. Wang, L. Gao, Q. Sun, and D. Sun. 2011. An improved CVSS-based vulnerability scoring mechanism. In *Multimedia Information Networking and Security (MINES), 2011 Third International Conference on*.
[39] T. Wen, Y. Zhang, Y. Dong, and G. Yang. 2015. A novel automatic severity vulnerability assessment framework. *Journal of Communications* (2015).
[40] G. Yang, T. Zhang, and B. Lee. 2014. Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports. In *Computer software and applications conference (COMPSAC), 2014 IEEE 38th annual*.