Revisiting the VCCFinder approach for the identification of vulnerability-contributing commits

Timothé Riom[1] · Arthur Sawadogo[2] · Kevin Allix[1] · Tegawendé F. Bissyandé[1] · Naouel Moha[2] · Jacques Klein[1]

Abstract

Detecting vulnerabilities in software is a constant race between development teams and potential attackers. While many static and dynamic approaches have focused on regularly analyzing the software in its entirety, a recent research direction has focused on the analysis of changes that are applied to the code. VCCFinder is a seminal approach in the literature that builds on machine learning to automatically detect whether an incoming commit will introduce some vulnerabilities. Given the influence of VCCFinder in the literature, we undertake an investigation into its performance as a state-of-the-art system. To that end, we propose to attempt a replication study on the VCCFinder supervised learning approach. The insights of our failure to replicate the results reported in the original publication informed the design of a new approach to identify vulnerability-contributing commits based on a semi-supervised learning technique with an alternate feature set. We provide all artefacts and a clear description of this approach as a **new reproducible baseline** for advancing research on machine learning-based identification of vulnerability-introducing commits.

Keywords Vulnerability detection · Machine learning · Replication · Software engineering

1 Introduction

Software development is a complex engineering activity. At any stage of the software lifecycle, developers will introduce bugs, some of which will lead to failures that violate security policies. Such bugs are commonly known as *software vulnerabilities* (Krsul 1998) and are one of the main concerns that our ever-increasingly digitalised world is facing. Detecting software vulnerabilities as early as possible has thus become a key endeavour for software engineering and security research communities (Zhu et al. 2019; Cadar et al. 2008; Livshits and Lam 2005; Larochelle and Evans 2001). Typically, software vulnerabilities are tracked

✉ Timothé Riom
  timothee.riom@uni.lu

1   SnT, University of Luxembourg, Luxembourg, Luxembourg

2   Université du Québec à Montréal, Montreal, Canada

during code reviews, often with the help of analysis tools that narrow down the focus scope by flagging potentially dangerous code. On the one hand, when such tools build on static analysis (either deciding based on code metrics or matching detection rules), the number of false positives can be a deterrent to their adoption. On the other hand, when the tools build on dynamic analysis (e.g., for pinpointing invalid memory address), they are operated on the entire software which may not scale to the frequent evolutions of software.

To address the aforementioned challenges that static and dynamic tools face in finding vulnerabilities, (Perl et al. 2015) have proposed the VCCFinder approach with two key innovations: (1) the focus is made on code commits, which are "the natural unit upon which to check whether new code is dangerous", allowing to implement early detection of vulnerabilities just when they are being introduced; (2) the wealth of metadata on the context of who wrote the code and how it is committed is leveraged together with the code analysis to refine the detection of vulnerabilities.

VCCFinder is a machine learning approach that trains a classification model, which can discriminate between safe commits and commits that lead to the code being vulnerable. The experimental assessment presented by the authors has shown great promise for wide adoption. Indeed, by training a classifier on vulnerable commits made in 2011 on open source projects, VCCFinder was demonstrated to be capable of precisely flagging a majority of vulnerable commits that were made between 2011 until 2014. VCCFinder further produced 99% less false positives than the tool the authors decided to compare their implementation to, namely FlawFinder (Wheeler 2001). Finally, the authors reported that VCCFinder flagged some 36 commits to which no CVE was attached, one of which has been indeed confirmed as a vulnerability introducing commit.

VCCFinder constitutes a literature milestone in the research direction of vulnerability detection at commit-time. Their overall detection performance, presented in the form of Recall-to-Precision curve, however indicates that the problem of vulnerability finding remains largely unsolved. Indeed, when precision is high (e.g., around 80%), recall is dramatically low (e.g., around 5%). This high precision is a promise that security experts' time will be spent on likely Vulnerability-Contributing Commits. This is how to make the best of their skills. Similarly, when aiming for high recall (e.g., at 80%), precision is virtually null.

Unfortunately, since the publication of VCCFinder, and despite the tremendous need and appeal of automatically detecting commits that introduce vulnerability, this field has not attracted as much interest, and therefore as much progress, as one could have imagined.

Thus, to date, it remains unclear (1) whether the ability of VCCFinder to detect Vulnerability-Contributing Commits can be replicated [1], (2) whether, given some variations in the datasets or in the algorithm implementation, the produced classification model is stable, and (3) whether some adaptations of the learning (e.g., to account for data imbalance) can improve the achievable detection performance.

This paper   We perform a study on the state of the art of vulnerability finding at commit-time in order to inform future research in this direction. To that end, we first report on a replication attempt of VCCFinder. Replication attempt for which we tried to stick as much as possible to the original work. Then, we present an exploratory study on alternative features from the literature as well as the implementation of a semi-supervised learning scenario. We contribute to the research domain in several axes:

---

[1]Throughout this paper, we use the words *reproduction* (different team, same experimental setup) and *replication* (different team, different experimental setup) as defined in the ACM Artifact Review and Badging Document. We further note that this terminology was updated in August 2020; We use the updated version. https://www.acm.org/publications/policies/artifact-review-and-badging-current

– We perform a replication study of VCCFinder, highlighting the different steps of the methodology and assessing to what extent our results conform with the authors published findings.
– We rebuild and share a clean, fully reproducible pipeline, including artefacts, for facilitating performance assessment and comparisons against the VCCFinder state-of-the-art approach. This new baseline might help unlock the field.
– We explore the feasibility of assembling a new state of the art in vulnerability-contributing commit identification, by assessing a new feature set.
– We identify one issue to be the lack of labelled data, and we explore the possibility to leverage a specialised technique, namely co-training, to mitigate that issue.

The main findings of this work are as follows:

– The VCCFinder publication lacks sufficient information and artefacts to enable replication.
– Despite our best experimental efforts, we were unable to replicate the results reported in the publication, suggesting some generalisation issues due to high sensitivity of the approach to dataset selection and learning process.
– A semi-supervised learning approach based on our new feature set (inspired by a recent work (Sawadogo et al. 2020) that is targeting the detection of vulnerability fix commits, rather than the detection of Vulnerability-Contributing Commits, or VCCs) does not achieve the same detection performance as reported in the state of the art. Nevertheless, our approach constitutes a **reproducible baseline** for this research direction.

> *While our work contains a replication study, it also acknowledges the limits of the replicated approach (i.e., VCCFinder) and, more importantly, it tries to unlock this important research field by providing a reproducible setup. Data, code and instructions are available. It also demonstrates that the artefacts we provide allow for new experiments to advance the state of the field.*

The rest of this paper is organised as follows:

– We first focus on describing the VCCFinder approach: what resources are available, what we had to guess, and how we reimplemented it (Section 2). We compare the achieved results with the originally presented ones.
– We then propose and evaluate in Section 3 a new approach, built with another feature set, and co-training.
– We finally contextualise our work with the existing related work (Section 4), and summarise our contributions in Section 5.

2 Replication Study of VCCFinder

The first objective of our work is to investigate to what extent the VCCFinder (Perl et al. 2015) state-of-the-art approach can be replicated (different team, different experimental setup) and/or reproduced (different team, same experimental setup). VCCFinder[2]

---
[2]VCCFinder means Vulnerability-Contributing Commit Finder

is a machine learning-based approach aiming at detecting commits which contribute to the introduction of vulnerabilities into a C/C++ code base.

As most machine learning-based approaches, VCCFinder relies on several building blocks:

1. A labelled dataset of commits which is used to train a supervised learning model;
2. A feature extraction engine that is used to extract relevant characteristics from commits;
3. A machine learning algorithm that leverages the extracted features to yield a binary classifier that discriminates vulnerability-contributing commits from other commits.

In the following, we present, for each of the aforementioned three building blocks, the descriptions of operations in the original paper. We then discuss to what extent we were able to replicate these operations. Subsequently, we present the results of our replication study.

2.1 Datasets

2.1.1 Datasets - VCCFinder Paper

A key contribution in the VCCFinder publication is the construction of two labelled datasets of C/C++ commits.

–   A dataset of commits that contribute vulnerabilities (VCCs) into a code base;
–   A dataset of commits that fix vulnerabilities that exist within a code base.

With the assumption that a commit that fixes a vulnerability does not introduce a new one, the authors consider the second dataset as a negative dataset (i.e., the corresponding dataset of non-vulnerability-contributing commits). To build both datasets, the paper reports that 66 open-source git repositories of C and C++ projects were considered. Overall, these repositories included some 170 860 commits. For the creation of the *vulnerability-fixing commits* data set, the authors gather all the CVEs[3] related to these repositories. They selected CVEs that are linked to a fixing commit. With this method, 718 vulnerability fixing commits were collected.

Collecting commits contributing to a vulnerability is less straightforward. Indeed, usually, commits introducing vulnerability are not tagged as such, and there are no direct information in the commit message that indicates the vulnerable nature of the commit.

To overcome this difficulty, the authors follow an approach defined by Śliwerski et al. (2005) and called SZZ. The principle is to start from vulnerable lines of code. Such vulnerable lines of code are identified thanks to the vulnerability fixing commits: indeed, it is reasonable to assume that the lines that have been fixed were previously vulnerable. Then the `git blame` command is used on these identified lines of code. The `git blame` command allows finding the last commit that modified a given line. The assumption here is that the last modification made on a vulnerable line of code is the modification that introduced the vulnerability.

Thanks to this method, 640 vulnerability-contributing commits (VCC) have been collected. Note that the numbers of vulnerability-contributing commits and vulnerability fixing commits are different simply because one commit can potentially contribute to more than one vulnerability.

---

[3]CVEs: Common Vulnerabilities and Exposures are publicly available cybersecurity vulnerabilities.

Table 1 Datasets comparisons

	VCCFinder Paper 66 repositories			Replication 38 repositories		
	Training	Test	Total	Training	Test	Total
Positive (vuln. contr. commit)[a]	421	219	640	470	253	723
Negative (vuln. fixing commit)	469	249	718	389	879	1268
Unlabelled	*90 282*	*79 220*	*169 502*	*229 381*	*119 489*	*348 870*
Total			170 860			350 861

[a]Vulnerability-Contributing Commit

In the VCCFinder paper, both datasets have been divided into a training set and a testing set (following a two-third, one-third ratio). All commits created before January, 1st 2011 are put in the training set, and the remaining in the test set. The numbers of commits of each dataset are presented in the left part of Table 1. Note that among the whole dataset of 170 860 commits, only 1258 (640+718) commits have been classified. The 468 (219+249) labelled commits in the test set is used as ground truth, notably to compute Precision and Recall performance metrics.

All other commits that are not categorised into the two first datasets (169 502) are put in a third dataset named *unlabelled* dataset. This dataset of unlabelled commits is also split into two datasets. All commits created after January, 1st 2011 are in a test set. In the original paper, this unlabelled test set is used to try to uncover yet-undisclosed vulnerabilities. The authors claim VCCFinder was able to flag 36 commits as VCCs. They detail one VCC for which they received confirmation from the development team that it was indeed a VCC. At the time they wrote the presentation of their work, they had not received confirmation for the others.

2.1.2 Datasets - Availability

The dataset of the original VCCFinder article is not directly accessible.

Online investigation may direct to a specific Github repository[4] that holds the name of the tool and the name of one of the authors. However, the original paper does not mention this repository. The code present in this repository is not fully documented, as was already mentioned by a prior work whose authors noted some major challenges to exploit its contents (Hogan et al. 2019). After carefully analysing this repository, we came to the conclusion that the artefacts in this repository would not allow us to re-construct the exact same dataset as the one used in the original VCCFinder. Moreover, it would not even allow to construct a *different* dataset, as parts of the features extraction process is missing (to the best of our knowledge).

---

[4]https://github.com/hperl/vccfinder

2.1.3 Datasets - Our Replication Study

At the time we reached a conclusion about the available Github repository, we had already contacted the authors of VCCFinder who offered to provide directly the output of their feature extraction pipeline. We accepted their offer, as it seemed that it was the only viable solution.

This dataset provided to us by VCCFinder's authors is a database export that contains three tables:

– A table listing 179 public repositories of C/C++ projects;
– A table listing 351 400 commits, each commit being linked to a repository thanks to the use of a repository id;
– A table listing the CVEs used to identify the vulnerability fixing commits.

Note that over those 179 repositories, all commits are related to an existing repository. However, only 50 repositories have at least one declared commit (i.e., 129 repositories have no related commit).

Furthermore, out of these 50 repositories, only 38 repositories contain at least one vulnerability fixing or vulnerability-contributing commit. Among these 38 repositories, only 27 are linked to both a vulnerability contributing commit and its relevant vulnerability fixing commit.

While no such process is mentioned by original authors, we opted to discard commits that do not modify any code file, as they are very unlikely to be involved in any vulnerability fixing or introducing. We used a simple heuristic that discards commits with no modification to a file whose extension is either `.h`, `.c`, `.cpp`, or `.cc`.

Table 1 presents a comparison between a) the number of commits that have been involved in our replication attempt, and b) the dataset described in VCCFinder original paper.

We note that the dataset provided to us is significantly different than the one described in the VCCFinder paper. We also note that we are unable to evaluate whether there is any overlap between the dataset we had access to and the original one.

> *As shown in Table 1, the datasets used in the VCCFinder paper and the ones used in our replication study are not identical. Even if the number of positive and negative samples in the training and test sets are close (same order of magnitude), we can notice significant differences regarding: (1) the number of repositories presenting a fixing commit (66 vs 38), (2) the number of negative samples (i.e. fix commits) in the Test sets (249 in the VCCFinder paper, 879 in our replication study).*
> *This fact alone guarantees that we will not be able to obtain exactly identical results. Given how much the datasets are different, we even expect our results to be potentially significantly different.*

Use of the data sets The aforementioned *ground truth* notion is important as VCCFinder's authors opted to both report performance metrics computed against this ground truth, and metrics computed on data they had no ground truth for (we do not know how they did this). Original authors were contacted but did not come back to us on the matter. As a result, we faced huge difficulty to clearly understand the notion of ground truth as used in the original VCCFinder paper.

Since our understanding of their notion of ground truth is based on deduction and guesswork, and not on a clear authoritative description from original authors, we now carefully detail on what we trained our classifiers on, and on what they were tested on. More specifically, we performed three different experiments:

1. What we think the original experiment was;
2. A less coherent setup;
3. A more traditional setup.

We note that we cannot definitely affirm which of the first or the second setup VCCFinder original paper used, as both are coherent with the figures reported. The repartition is presented in Table 2, and detailed in the following paragraphs:

Unlabelled Train Replication A classifier is trained on the whole training set, including the unlabelled commits created before 2011. This first one is the one we think to match the most with the description of the original experiment. The negative label (i.e., not VCC) is associated with those unlabelled commits before training. The resulting classifier is tested on the whole test set, including the unlabelled commits from 2011 and newer. Similarly, those unlabelled commits are associated with the negative label. The goal being to find VCCs, if the resulting classifier predicts one originally unlabelled commit to be a VCC, this will display as a *False Positive*.

Unlabelled Replication This setup is very similar to the previous one, with the exception that the unlabelled commits created before 2011 are not used in the training phase. Those related to after 2011 are used in the test set (and associated with the negative label). This scenario would enable to analyse the model's behaviour once facing security neutral commits. That is to say, commits that are neither VCCs nor fixing commits, the latter having to be written with a security mindset. Still, the model would train on the closest we have to a ground truth. This setup is less coherent in the sense that unlabelled commits are not treated similarly in the training than in the testing.

Ground Truth Replication In this more traditional setup, a classifier is trained on the train set for which we have a ground truth, i.e., excluding the unlabelled commits. Similarly, the resulting classifier is tested on the test set for which we have a ground truth, i.e., excluding the unlabelled commits.

Table 2 Dataset repartition scenarios

		Training	Test
Unlabelled Train	positive	470	253
Replication	negative	229 770 (389 + 229 381)	120 368 (879 + 119 489)
Unlabelled	positive	470	253
Replication	negative	389	120 368 (879 + 119 489)
Ground Truth	positive	470	253
Replication	negative	389	879

2.2 Features

2.2.1 Features - VCCFinder Paper

The second main step of the VCCFinder approach consists in extracting the relevant features that will feed the machine learning algorithm. Among the selected features, VCCFinder considers *code metrics* and *meta-data* related to both a particular commit and the whole repository.

Regarding the commit[5] itself, the patch code and the commit message are both considered. Note that a specific section of the original paper is dedicated to asserting the relevance of the features by comparing their frequency in vulnerability-contributing commits and other commits.

Regarding code metrics, for a given commit m from a repository R, VCCFinder extracts:

– The number of structural keywords of C/C++ programs (such as `if`, `int`, `struct`, `return`, `void`, `unsigned`, `goto`, or `sizeof`, etc) present in m. Overall, 62 keywords are referenced;
– The number of hunks[6] in m;
– The number of additions in m;
– The number of files changed in R.

Regarding metadata, for a given commit m from a repository R, VCCFinder considers:

– The total number of commits in R;
– The percentage of commits in R performed by the author of m;
– The number of changes performed on the files modified by m after m was applied;
– The number of changes performed on the files modified by m before m was applied;
– The number of authors altering the files impacted by m;
– The number of stargazers, forks, subscribers, open issues and others, including the commit message itself.

2.2.2 Features - Availability

The earlier mentioned git repository ends up registering commits in a database, though as already stated (Section 2.1.2), we are unsure whether the resulting database would have all the information needed, in particular, we have been unable to locate code that would compute all the features required. Furthermore, the original paper does not contain enough details to fully re-implement the full feature extraction ourselves.

Therefore, regarding the extraction of features, we have to rely on the fields present in the database given by the original authors.

2.2.3 Features - Our Replication Study

As already explained, the original paper does not precisely list all the features extracted leading to a situation where we were unable to re-implement a feature extraction engine, and thus unable to re-use their approach on another dataset.

---

[5]We remind that a commit is composed of a patch (i.e., the "diff" representing the code changes), and a commit message (explaining the modification performed by the patch)

[6]a hunk is a block of continuous added lines

However, the database that was shared with us already contains the features computed by VCCFinder authors themselves. We hence directly used those features.

> *Since the VCCFinder authors sent us datasets with the features already extracted, our replication study leveraged exactly the same features as the VCCFinder approach. However, since we did not obtain or re-implement the feature extraction engine, we are not able to extract features from other datasets of commits.*

2.3 Machine Learning Algorithm

2.3.1 Machine Learning Algorithm - VCCFinder Paper

The VCCFinder approach leverages an SVM algorithm (through its LibLinear (Fan et al. 2008) implementation) to learn discriminating vulnerability introducing commits from other commits. This algorithm builds a hyper-plan that would separate, in our case, vulnerability introducing commits from others. To classify a given commit, a distance is computed between the feature vector of this commit (i.e., a point in the hyper-space) and this hyper-plan. The sign of this distance determines whether this commit contributes to a vulnerability or not.

Given a commit and the extracted features, we describe now the generation of the feature vector of this commit that is used as input of the machine learning algorithm. This process follows a generalised bag-of-words approach that normalises the features' values into boolean vectors. Regarding the normalisation, for each feature, commits are categorised into bins based on the occurrences of the feature. Then a string is built by concatenating the name of the feature and the bin identifier. Finally, joining all these newly created strings together with the texts formed by the patch code and/or commit message, a considerable string is built and fed to a tool named SALLY (Rieck et al. 2012). SALLY is a binary tokenisation tool which generates a high-dimensional sparse vector of booleans from a string, computing a hash for each split-on-space sub-string. At the end of this process, each commit is represented now by, first, a boolean, indicating its class (vulnerability-contributing commit or not) and a succession of pairs (`feature_hash/binary value`) that represent a sparse vector of the features.

The VCCFinder authors mention they used a handicap value C of 1 and weight for this one-class problem of 100 as "*the best values*" (last sentence of their section 4.2).

Eventually, the authors present their results on the test set with a Recall-to-Precision curve for which the actual parameter is the threshold in Fig. 1. After computing the distance from the hyperplane for each commit in the test set and by incrementally lowering the threshold, the commits the closest to the hyperplane will be classified as VCCs. Lowering the threshold results in increasing the number of True Positives, but might also quickly bring more False Positives. The higher the Recall-to-Precision curve, the more precise, and the more horizontal, the more the model is not sacrificing precision for recall.

2.3.2 Machine Learning Algorithm - VCCFinder Availability

As already explained, VCCFinder authors did not release code that perform all the required steps of their approach. Even in the repository found on the Internet (but not mentioned in the VCCFinder paper), the code that orchestrates the training of the classifier and its usage is absent.
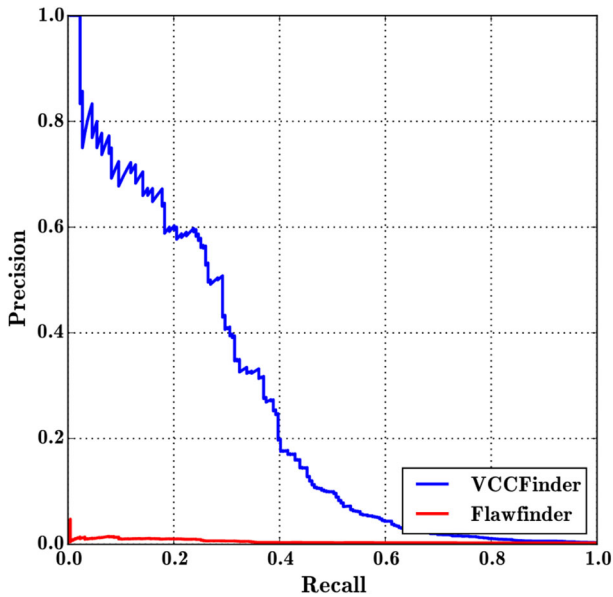
Fig. 1 Extracted from the VCCFinder paper: precision/recall performance profile of VCCFinders

However, as noted above, authors provide some of the parameters in the paper. We note that the embedding step (i.e., tokenisation and discretisation) is almost adequately described in the original paper, with the exception of the number of bins (cf. below).

2.3.3 Machine Learning Algorithm - Our Replication Study

The VCCFinder authors mentioned they used the LibLinear (Fan et al. 2008) library to run the SVM algorithm. However, several front-ends of LibLinear exist. We decided to use the `LinearSVC`[7] implementation included in the popular framework scikit-learn.

Regarding the construction of the feature vectors, and more specifically regarding the normalisation step, the authors do not specify the number of bins they use, nor on which features this step was performed. We decided to consider 10 bins per feature containing each, as much as possible, the same number of commits. This was done with scikit-learn's `preprocessing.QuantileTransformer` facility, assigning the value of 10 to `n_quantiles` parameter, and `'uniform'` to the `output_distribution` parameter.

We then apply LinearSVC classifier with C parameter equals to one, the weight of the class one to 100 over 200 000 iterations.

> With the exception of the exact usage of the unlabelled commits,
> we are rather confident that our own implementation of the machine learning algorithm building blocks mimics the VCCFinder one. However, we cannot evaluate if the differences have a significant impact on the results obtained.

---

[7]https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

2.4 Results

In this section, we detail the results yielded by VCCFinder in the original paper, as well as the results that we obtain when we replicate VCCFinder.

2.4.1 VCCFinder Paper

To assess the performance of their machine learning-based approach, the authors keep about two-thirds of their datasets for training, and use one-third of the datasets for testing. Table 1 presents the exact numbers. Note that, as explained in Section 2.1, we are not sure about what the training and testing sets are composed of.

The original results are presented in Fig. 1, which is directly extracted from the paper (Perl et al. 2015). The plot is obtained by measuring/computing precision and recall values when varying the threshold.

In the original paper, the authors compare VCCFinder against a then-state-of-the-art tool named flawfinder (in red in Fig. 1). Flawfinder is a static analyser tool that looks for dangerous calls to sensitive C/C++ APIs in the code as `strcpy` and flags them.

Figure 1 shows that VCCFinder greatly outperforms Flawfinder. The authors also set their tool to the same level of recall that Flawfinder is capable of for this dataset, 24%, and show that their approach presents then a precision of 60%. In comparison, Flawfinder can only achieve 1% in such conditions. For a recall of 84%, VCCFinder has a precision of 1%.

With precision and recall values extracted from Fig. 1, an F1-score can be computed thanks to the following formula:

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

We can notice that the maximal F1-score of VCCFinder seems to be lower than 0.4, with a maximum of either (Recall;Precision) =(0.25;0.6) or (Recall;Precision)=(0.3;0.5). Those lead to an F1-score of either 0.35 or 0.375.

Table 3 describes several metrics (extracted from the original paper) such as True Positive, False Positive, etc computed on the test set. VCCFinder flagged 53 commits that are, according to the ground truth, actually introducing a known vulnerability. Applying VCCFinder to the larger set of unclassified commits, 36 commits were flagged as suspicious. Among those 36 potential VCCs, one was described by authors as confirmed by the project maintainers, who had already patched this vulnerability. Authors opted not to comment on the other 35 commits, invoking "responsible disclosure". These 36 commits are presented as belonging all to the post-January 2011 unclassified set. Thus, on what they define themselves as the ground truth, no false positive is met.

2.4.2 Our Replication Study

The results presented in Fig. 2 show the precision per recall we obtain on the 3 different test sets while diminishing the threshold. One can understand the threshold as the minimum distance from the hyperplane for a commit to be considered as VCC. The grey curves represent the lines for a constant F1-score at 0.2, 0.4, 0.6 and 0.8. We now details the results for each of the 3 test sets presented in 2.1.3:

Ground Truth Replication The replication achieves a maximum F1-score of 0.63 for a recall of 0.76 and a precision of 0.54 (see line 2 of Table 3 and green dots in Fig. 2). We also

Table 3 Results of replication on updated test set

	True Positive(VCC[a])	False Positives	False Negatives	True Negatives[b]	Precision	Recall
VCCFinder	53	36	166	79 184	0.60	0.24
Ground_Truth Replication	61	5	192	885	0.92	0.24
Unlabelled Replication	61	3145	192	157 224	0.02	0.24
Unlabelled Trained Replication	61	695	192	159 674	0.08	0.24

[a]VCC: Vulnerability-Contributing Commit

[b]Vulnerability-Fixing Commit and post-2011 Unlabelled

set ourselves, for the purpose of comparison, to the reference recall used in VCCFinder's original paper of 0.24 to find a precision of then 0.92. In these conditions, the F1-score is of 0.38. It presents a progressive decline and correctly tags 61 commits as VCCs.

Unlabelled Replication This attempt trains on the ground truth but is tested on both ground truth and beyond 2011 unclassified is drawn in red in Fig. 2. We can see it perform very poorly, presenting more than three thousand false positives, once set to the same recall of 0.24. The precision is then barely of 2% and the F1 score of 0.037.

Unlabelled Train Replication It is after assessing how poorly the last experiments performed that we decided to include unclassified in the training, forcing them as non-VCCs. The results are illustrated thanks to the blue curve in Fig. 2 and the last row of Table 3. It improves sensibly the performances without reaching the level of the original. The precision for fixed recall is of 8%, leading to an F1-score of 0.12.
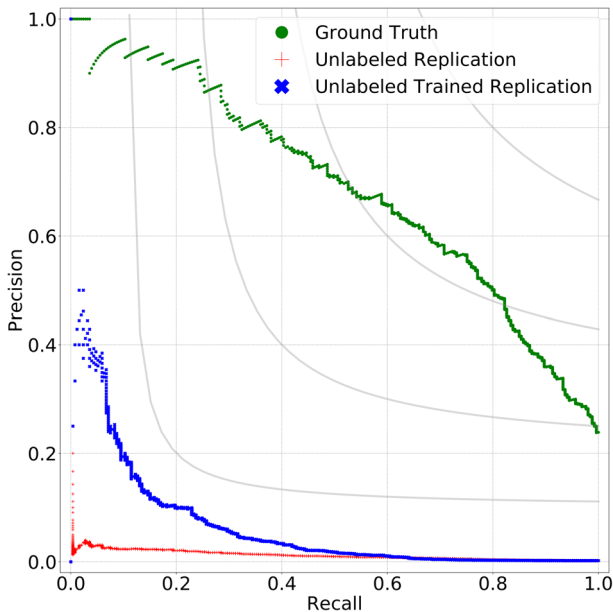
Fig. 2 Precision/recall performance profile of VCCFinder's Replication

2.4.3 Parameters Exploration

Besides the results on the 3 different test sets, we took the opportunity of this replication attempt of VCCFinder to investigate the impact of various parameters.

Exploration over parameter C In the original paper it is just stated that the optimal conditions are for a cost parameter C of 1. We experiment for different values of C on the basis of the Ground Truth Replication. We experiment for values from $C = 10^{-6}$ to 100, and obtain the values presented in Fig. 3.

It appears that the behaviour seems to tend toward an optimal behaviour starting at $C = 10^{-2}$ and higher. Thus, as advocated by the VCCFinder authors, using a value of C at 1 makes sense.

Exploration over class weight parameter Altering the weight of the positive class (VCCs) from 0.1 to 100, we saw no difference in the output using the same other settings. There is, thus, no reason to deviate from the original paper declared values.

Exploration with other algorithms We also experimented with a variety of different machine learning algorithms. Results are presented in Fig. 4. We note that SVM—that is used by the original VCCFinder paper—is among the algorithms that produce the best results.

2.5 Analysis

We discuss the experimental results of our replication attempt of the VCCFinder approach.
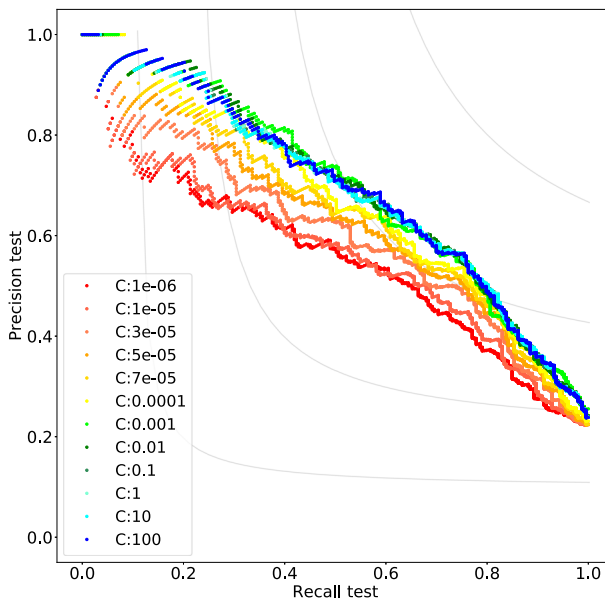***RQ 1: Is our reproduction of VCCFinder successful?***

Fig. 3 Precision/recall performance profile of VCCFinder's replication for varying values of C parameter
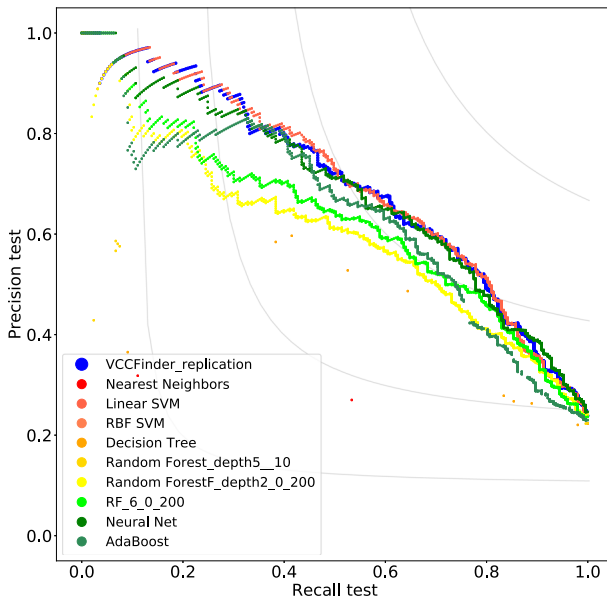
Fig. 4 Precision/recall performance profile for comparing classifying algorithms

According to the terminology used by ACM's *Artifact Review and Badging* guidelines, a *Reproduction* requires the same experimental setup (Association for Computer Machinery 2020). We recognise that some elements of our setup were different from the setup in VCCFinder publication. We have therefore documented the differences.

We note that the combination of a) an implementation of the approach, and b) the exact dataset used originally would have allowed us—and any other researcher—to positively validate the results reported by VCCFinder's authors.

> *We have been unable to Reproduce VCCFinder.*

RQ 2: Does the present work constitute a successful Replication of VCCFinder?

The ACM's terminology states that researchers conducted a successful *Replication* when they "obtain the same result using artifacts which they develop completely independently".[8]

We were unable to obtain the same results, mostly because we were unable to re-implement ourselves the code based on the paper. This is caused by the lack of details and/or of clarity of the original paper. As an example, even if we had had access to the software that collects the code repositories and built a database,[9] we would still miss the complete list of repositories that were involved in the original experiment.

> *We have been unable to Replicate the results in the VCCFinder publication.*

---

[8]https://www.acm.org/publications/policies/artifact-review-and-badging-current

[9]Note that the link provided in footnote 1 of page 3 in the original post-print publication raises a 404 error.

Given that the differences in experimental results between our replication study and the original VCCFinder publication may be due to the variations in the dataset or in the learning process, we propose to investigate an alternative approach, that we would make available to the research community, and that could yield similar performance to the promising one reported in the VCCFinder paper.

3 Research for Improvement

VCCFinder is an important milestone in the literature of vulnerability detection. Indeed, departing from approaches that regularly scanned source code to statically find vulnerabilities, VCCFinder initiated an innovative research direction that focuses on code changes to flag vulnerabilities while they are being introduced, i.e., at commit time. Unfortunately, its replicability challenges advances in this direction. By investing in an attempt to fully replicate VCCFinder and making all artefacts publicly available, we unlock the research direction of vulnerability detection at commit-time and provide the community with support to advance the state of the art.

Considering our released artefacts of **a new replicable baseline**, we propose to investigate some seemingly-appealing variations of the VCCFinder approach to offer insights to the community. Thus, in this section, we go beyond a traditional replication paper by :

(1) Studying the impact of leveraging a different feature set that was claimed to be relevant to vulnerabilities (Sawadogo et al. 2020), thus proposing a new approach to compare against VCCFinder (in Section 3.1);

(2) Trying to overcome the problem of unbalanced datasets, i.e., the fact that there are much more unlabelled samples than labelled ones (in Section 3.2).

3.1 Using an Alternate Feature Set

As described above, the feature set used in VCCFinder is not sufficiently documented to be re-implemented, and the VCCFinder authors did not release a tool that is able to extract features from a collection of commits.

In this section, we investigate the use of an alternate feature set, described in a recent publication (Sawadogo et al. 2020) that is targeting **the detection of vulnerability fix commits, rather than the detection of VCC**. To reduce ambiguity when needed, we refer to this alternate feature set as *New Features*, while the VCCFinder feature set is denoted *VCC Features*.

In this experiment, the settings of the machine learning stay the same as in the replication (LinearSVC with C=1 and the class weight set to 100).

RQ 3: How a less extensive but more security-focused feature set alters the VCCFinder approach?

3.1.1 New Feature Set

The *New Feature set* is made of three types of features: Text-based features, Security-Sensitive features and Code-Fix features. They are all shown in Table 4

–   Code metrics: A difference between the two feature sets concerning the code is that the new feature set focuses on 17 characteristics of the code, while VCCFinder collects

Table 4 Alternate set of features (adapted from Sawadogo et al. 2020)

ID	Code-fix	ID	Security-sensitive
F1	#commit files changed	S1	#sizeof added
F2	#loops added	S2	#sizeof removed
F3	#loops removed	S3	S1−S2
F4	F2−F3	S4	S1+S2
F5	F2+F3	S5-S6	Like S1-S2 for continue
F6-F9	Like F2-F5 for *if*	S7-S8	Like S1-S2 for break
F10-F13	Like F2-F5 for Lines	S9-S10	Like S1-S2 for INTMAX
F14-F17	Like F2-F5 for Parenthesized expression	S11-S12	Like S1-S2 for goto
F18-F21	Like F2-F5 for Boolean operators	S13-S14	Like S1-S2 for define
F22-F25	Like F2-F5 for Assignements	S15-S18	Like S1-S4 for struct
F26-F29	Like F2-F5 for Functions call	S19-S20	Like S1-S2 for offset
F30-F33	Like F2-F5 for Expressions	S21-S24	Like S1-S4 for void
ID	Text		
W1-W10	Most recurrent top 10 word		

62 keywords. Though, for each, it also computes whether they are added, removed, the difference of those two factors and their addition.

Taken individually, most of them are common to the two feature sets. Except for the count of elements under parenthesis, function calls, keywords: INTMAX, define and offset, VCCFinder's feature set includes them all and beyond.

– Commit message: In *New Features*, only the ten most significant words present in the commit message corpus, as obtained through a term-frequency inverse-document-frequency (TFIDF) analysis, are captured.

Note that we tried to normalise the features (as recommended in Hsu et al. (2003)). The results of detection along the test set were the same or slightly worse with this normalisation step. Thus we decided not to normalise the features.

3.1.2 Results

Figure 5 and Table 5 present the performances with the New Feature Set.

By considering the Ground Truth only (second line of Table 5 and green curve in Fig. 5), the New Features are less performant than VCC Features. For, still, a recall of 0.24, the precision is only 67% while it used to top at 92% in such a case.
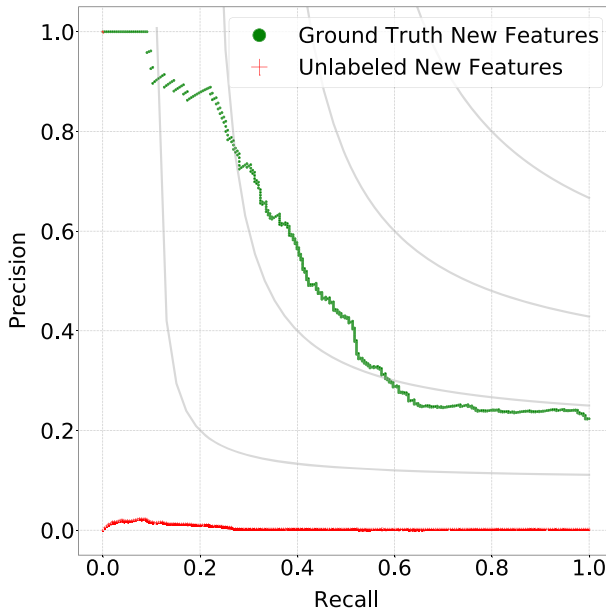
Fig. 5 Precision-recall performances using New Features

Here again, because of the doubt on what is the actual test set in the original paper (cf. Section 2.1.3), we also tested on both the ground truth and the unclassified commits post January, 1st 2011 (red curve in Fig. 5 and last row in Table 5).

> *Our feature set does not allow to outperform our VCCFinder replication.*

3.2 Adding Co-Training

A major issue with any VCC detection endeavour is the lack of labelled data, with less than one per cent of the data being labelled. While researchers can collect many hundreds of thousands commits, acquiring even a modest dataset of known VCCs requires a massive effort.

One field of machine learning focuses on the usability of the unlabelled data. The study by Castelli and Cover (1995) states that it is possible, in some case, to leverage unlabelled samples to improve a machine learning model. Zhang and Oles (2000) investigated the

Table 5 Confusion table for new features

	True Positive(VCC)	False Positives	False Negatives	True Negatives	Precision	Recall
VCCFinder	53	36	166	79 184	0.60	0.24
Ground Truth New Features	61	9	192	854	0.871	0.241
Unlabelled New Features	61	5672	192	120 346	0.010	0.241

potential for gaining information from unlabelled data. This last study concludes that so called active-methods have already proven theoretical efficiency.

In our case, depending on the interpretation of the use of the dataset as explained earlier, unlabelled commits for training (before 2011) are either discarded (*Ground Truth* experiment) or incorporated in the non-VCCs set (*Unlabelled Replication* and *Unlabelled Train Replication*).

RQ 4: Can semi-supervised sorting of unlabelled data improve the VCCFinder approach?

One semi-supervised learning approach, called co-training and introduced by Blum and Mitchell, could help answer this question. On a Web page classification problem, Blum and Mitchell (1998) used two classifiers in parallel to complete training sets with unlabelled data. They ended up with an error rate of just 5% based on both the page content and hyperlinks over a test set of 265 pages: only 12 pages labelled (3 as positives course-pages, 9 negatives) and around 800 unlabelled. They demonstrated that Co-Training achieved performances on this problem that was unmatched by standard, fully-supervised machine learning methods. It is a technique that has industrially proven a reduction of false positive by a factor 2 to 11 on specific element detection on a video (Levin et al. 2003), and for which conditions of maximum efficiency it induces were analysed (Balcan and Blum 2005).

3.2.1 Co-Training Principle

When trying to detect VCCs, an important point is that unlabelled commits are unlabelled not because they are not VCCs, but because it is unknown whether they are VCCs. Arguably, in any large-enough collection of commits, it is reasonable to assume at least some of them are actually VCCs.

The insight behind trying Co-Training with VCC detection is the following: By building two preliminary and independent VCC classifiers, the unlabelled commits predicted to be VCCs by both classifiers could be used to augment the training set. By repeating this step, it might be possible to leverage the vast space of unlabelled commits.

3.2.2 Description of the Algorithm

Blum and Mitchell (1998) showed that the co-training algorithm works well if the feature set division of dataset satisfies two assumptions: (1) each set of features is sufficient for classification, and (2) the two feature sets of each instance are conditionally independent given the class.

Both the VCC Features set and the alternate feature set can be split into two subsets of features: One based on code metrics, and one based on the commit message.

Previous work on security patches detection showed that, for the New Feature set, the two resulting feature subsets are independent, and thus satisfy the two main assumptions for Co-training (Sawadogo et al. 2020).

Once these two assumptions are satisfied, the Co-training algorithm considers these two feature sets as two different, but complementary *views*. Each of them is used as an input of one of two classifiers used in Co-training: One focused on code metrics, and the other on commit messages. The algorithm is given three sets: a positive set, a negative set, and a set of unlabelled.

As described in Algorithm 1, and shown in Fig. 6, the training process is an iterative process in which each classifier (**h1** and **h2** on Fig. 6) is initialised being just given the

labelled inputs **LP**, that is used as the ground truth. From the whole set of unlabelled, a subset **U'** is randomly selected. At every round, each classifier is trained on a labelled set (**LP** for the first round). Then a number of unlabelled commits from **U'** are classified with those two classifiers. When both classifiers agree on a commit, this commit is added to the ground truth, i.e., it will be used to augment the training set in the next round. The process keeps going until we reach a predetermined size of the labelled set.

Algorithm 1 Steps for each Co-Training iteration. (extracted from Sawadogo et al. 2020).

---

input : training set (LP), unlabelled data (UP)
input : pool U'

output: U': updated pool
output: LP: updated training set

Function $getView(x, classifier)$
   | **if** $classifier = C_1$ **then**
   | | return $Text\_features(x)$
   | return $Code\_features(x)$
Function $buildClassifier(first)$
   | $vectors = \emptyset$;
   | **if** $first = True$ **then**
   | | **foreach** $x \in LP$ **do**
   | | | $vectors = vectors \cup getView(x, C_1)$;
   | **else**
   | | **foreach** $x \in LP$ **do**
   | | | $vectors = vectors \cup getView(x, C_2)$;
   | $classifier \leftarrow train\_model(\text{SVM}, vectors)$;
   | return $classifier$;
$h_1 \leftarrow buildClassifier(True)$; $h_2 \leftarrow buildClassifier(False)$;
$(P_1, N_1) \leftarrow classify(h_1, U')$; $(P_2, N_2) \leftarrow classify(h_2, U')$;
$LP \leftarrow LP \cup random\_subset(\#p, P_1) \cup random\_subset(\#p, P_2)$;
$LP \leftarrow LP \cup random\_subset(\#n, N_1) \cup random\_subset(\#n, N_2)$;
$U' \leftarrow U' \cup random\_subset(\#2 * (p + n), UP)$;

3.2.3 Implementation

For the implementation of the Co-training, we select two Support Vector Machines (SVM) (Vapnik 2013) as classification algorithms. We also perform experiments using three different size limits of the training set: by 1000, 5000 and 10 000 unlabelled commits added.

    This variation enables us to compare the effect of this variable in prediction performance. To respect temporality, the unlabelled commits were all taken before January, 1st 2011, as was for the original unaltered training set. For both sets of features, the co-training occurs after the extraction of features. One classifier trains on the code metrics and the other on the metadata. We finally use, as for the replication, a LibLinear model to classify the commits of the test set. For the latter values of C is 1 and, still, the weight of the class to 100.
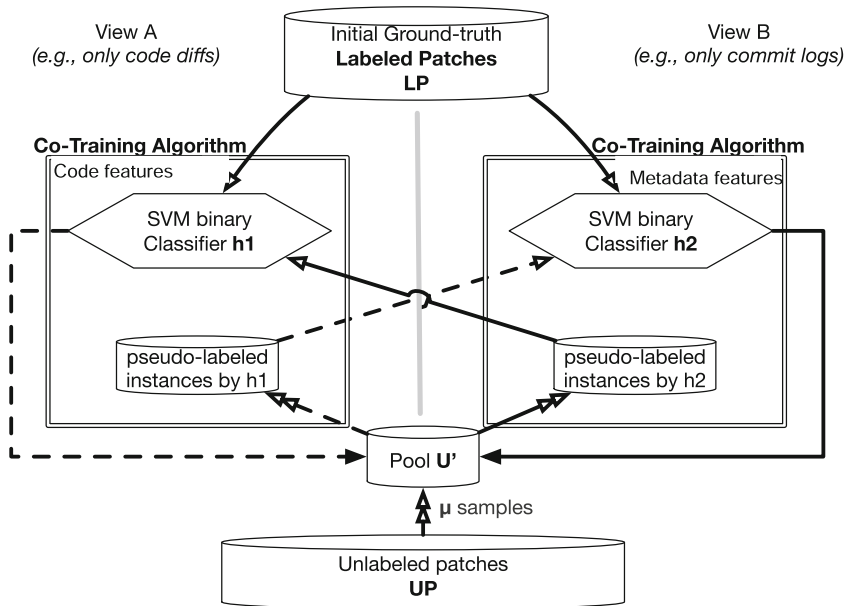
Fig. 6 Co-Training (Figure extracted from Sawadogo et al. 2020)

3.2.4 Co-Training Results

Co-Training with VCC Features Performance is improved slightly (cf. Fig. 7 vs Fig. 2) when Co-Training is used in conjunction with VCC Features. This improvement, however, does not appear to change with the size increase of the training set (whether 1000 or 10 000).

When testing with the Unlabelled Test, performance drops for all attempts. Therefore, no improvement can be concluded in this aspect.

Co-Training with New Features Figure 8 presents the results for a Co-Training process based on New Features. It includes variations for the training set (with 1000 and 10 000 unclassified commits) and, tests with and without the unclassified commits. On testing without the unlabelled Test set, one can conclude that the increase of 1000 unlabelled already helps perform better than the baseline green curve of Fig. 5. An increase of the dataset by 10 000 is further contributing to detect more VCCs.

3.2.5 Co-Training Analysis

> The Co-Training we implemented does not seem to be of particular help for the identification of VCCs.

This finding is clear when we consider the unclassified commits, in which cases the performance metrics dramatically drop. There seems to be an effect, though, for the New Features when only considering the Ground Truth.
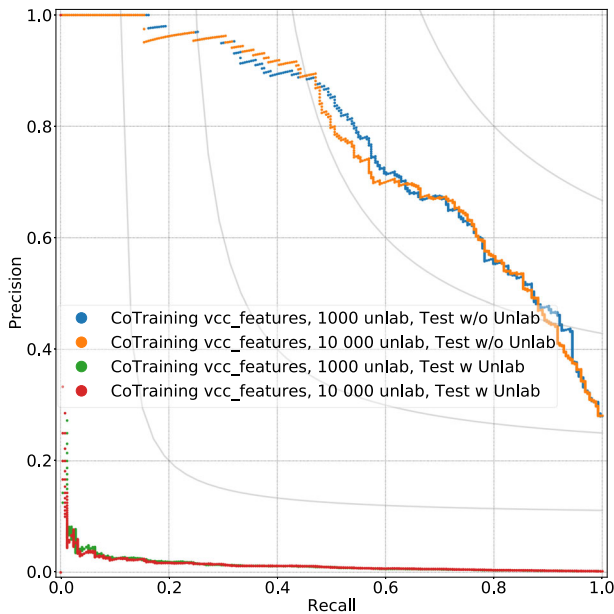
Fig. 7 Co-Training Performance using VCC Features' set

4 Related Work

The possibility of automatically finding vulnerabilities in code bases has long been identified by researchers as a worthy investigation target. In this section, we present a selection of significant prior works that we group by families of approaches.

4.1 Static Analysis for Vulnerability Detection

First released in May 2001, Flawfinder

performs static analysis of C and C++ programs and detects calls to a manually curated list of sensitive APIs (Wheeler 2001). Examples of such APIs widely recognised as sensitive are `strcpy`, `random` or `syslog`.

Splint (Larochelle and Evans 2001) is another static security testing tool, which performs lightweight analyses of ANSI C code and augments the code with annotations that set constraints on each C statement. It notably reveals the risks of buffer overflows, and alteration of the flow of instructions around loops and `if`s. Splint does not pretend to be complete nor sound but a good first pass at a very small cost. It was evaluated on BIND and wu-ftpd and uncovered a few buffer overflows, both known and by-then-unknown.

Find-Sec-Bugs[10] targets Web applications written in Java, and searches for potential vulnerabilities by matching high-level patterns that model problematic code pieces. Find-Sec-Bugs was made available to developers through a convenient IDE plugin.

Recently, Arusoaie et al. (2017) compared several open-source, security-oriented, Static Analysers for C and C++ code. Among the tools compared are:
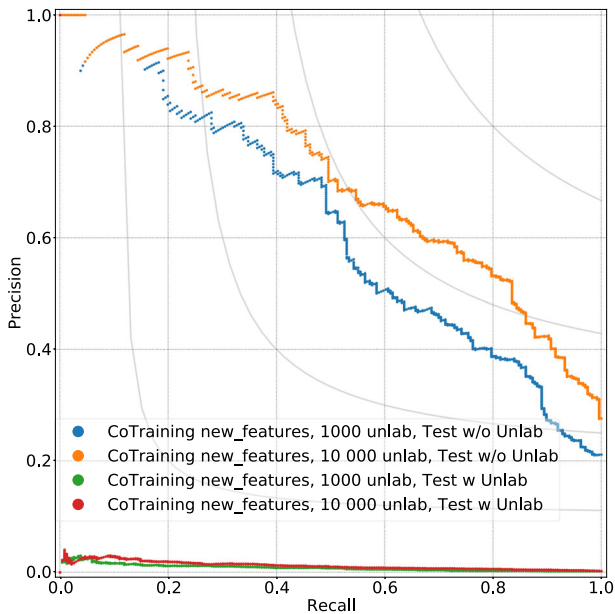
---

[10]https://find-sec-bugs.github.io

Fig. 8 Co-Training Performance using New Features set

–   **Frama-C** (Signoles et al. 2012), that leverages Static- and Dynamic-Analysis, Formal verification, and Testing;
–   **Clang**[11], that can find bugs such as memory leaks, 'use after free' errors, and dangerous (though valid) type casting;
–   **Oclint**[12], that performs analyses of Abstract Syntax Trees to find known patterns of dangerous code constructs;
–   **Cppcheck**[13], that specialises in finding undefined behaviours, and that strives to produce very few False Positives;
–   **Infer**[14], that catches memory safety errors by trying to build formal proofs of programs, and then interpreting failures of proof as bugs;
–   **Uno** (Holzmann 2002), that offers an approach aiming at detecting a limited number of errors, but with high precision;
–   **Sparse**, that was developed by Torvalds et al. (2003) specifically for the Linux kernel and thus can detect low-level errors in (among other things) bitfields operations or endianness;
–   **Flint++**[15], that can detect and warn developers about dangerous coding practices.
–   **git-vuln-finder**[16], that is based on C/C++ pattern matching.

---

[11] https://clang-analyzer.llvm.org

[12] http://oclint.org

[13] http://cppcheck.sourceforge.net

[14] https://fbinfer.com

[15] https://github.com/JossWhittle/FlintPlusPlus

[16] https://github.com/cve-search/git-vuln-finder

Arusoaie et al. (2017) were able to compare those approaches both quantitatively and qualitatively, and characterised Frama-C as the most precise approach, Oclint as the tool uncovering most dangerous behaviours, and Cppcheck as presenting a very low false-positive rate.

Taint analysis allows to follow the path data travels inside a program. This can allow uncovering vulnerabilities that would not be detectable by analysing one function/class/package at a time. Such approaches were proposed by Arzt et al. (2014) for Android applications in order to locate insecure use of data caused by the interactions of several software components.

Yamaguchi et al. (2014) demonstrated an approach that combines Abstract Syntax Trees (AST), Program Dependence Graphs (PDG), and Control Flow Graph (CDG). They were able to discover 18 new vulnerabilities in the Linux kernel.

A recent implementation was tried by Wang et al. (2016) with BUGRAM that generates n-gram sequences and considers the least likely as a bug. BUGRAM was run on 16 Java projects and found 14 confirmed bugs that other state-of-the-art tools were not able to find.

Martin et al. (2005) introduced a query language to search patterns of dangerous use, such as non-encrypted password hard-disk writing or possibility left for a SQL injection.

Livshits and Lam (2005) presented a framework available as an Eclipse plug-in to perform various static analyses. Their approach managed to find 29 security errors, two of which in widely used Java software: hibernate and the J2EE implementation.

4.2 Vulnerability Detection with Symbolic Execution

Symbolic execution has also long been identified by researchers as a promising technique to detect vulnerabilities in software. It enables some flexibility on the testing by using unknown symbolic variables rather than hard-coded-like asserting tests. Symbolic execution methods were notably experimented in 2008 by the tool KLEE that found 56 new bugs, including 3 in COREUTILS (Cadar et al. 2008).

A good review of the use of Symbolic execution for software security was published in 2013 by Cadar and Sen (2013).

More recently, Li et al. (2016a) leveraged CIL—a C intermediate language—library to statically analyze the source code, allowing backward tracing of the sensitive variables. Then, the instrumented program is passed to a concolic testing engine to verify and report the existence of vulnerabilities. Their approach focuses on buffer overflows and was reportedly not able to deal with nested structures in C code, function pointers and pointer's pointer.

4.3 Vulnerability Detection with Dynamic Analysis

Another important technique for software security is Dynamic Analysis, where programs under test are actually run and monitored. Fuzzing, which automatically generates inputs and tests a program on them, has rapidly come to play a major role in software vulnerability detection. Fundamentally, a fuzzer is an infinite loop which mutates an input seed and launches the target program on the mutated seed. If the target crashes, a bug is detected. Manual analysis will tell if the bugs is a vulnerability or not. AFL is a popular fuzzer for C/C++ programs (Zalewski 2017). Recent works (Zhu et al. 2019; Klees et al. 2018) use it as the reference. AFL instruments the target program to keep track of the coverage. If a mutated seed increases the coverage, the seed is kept to be mutated further. FuzzIL is a

fuzzer for Javascript VM (Groß 2018). Like AFL, it uses coverage to rank seeds. JQF (Pad-hye et al. 2019) or Kelinci (Kersten and Luckow 2017) are coverage-guided fuzzers to test Java programs.

Approaches have augmented Symbolic execution with actual execution of parts of programs, allowing to overcome limitations of symbolic execution. Such hybrid methods are called *concolic*, as they mix both **conc**rete and symb**olic** execution.

MACE (Cho et al. 2011), uses model-inference to direct concolic execution. This approach improves the exploration of the state-space of programs, thus allowing to find more vulnerabilities than tools with less coverage.

4.4 Vulnerability Detection with Code Metadata

Often, code nowadays comes with large amounts of associated metadata, such as bug tracking and code versioning information.

This metadata was quickly identified as a treasure trove ready to augment vulnerability detection approaches. In 2005, it was shown by Śliwerski et al. (2005) that changes made on Fridays to the Mozilla and Eclipse projects were more likely to introduce problems than the changes made in other days.

Kim et al. (2008) considered change log, author, change date, source code, change delta and metadata on 12 well-known software projects (Apache HTTP, Bugzilla, Eclipse, Post-greSQL, etc). They were able to reach an average precision of 0.61 for a recall of 0.6 for vulnerability introducing commits.

Vulture was demonstrated by Neuhaus et al. (2007). It is able to learn known vulnerabil-ities to detect new ones. Vulture managed to obtain a 70% precision on the Mozilla project, while not only detecting vulnerabilities, but also pinpointing their location.

Wijayasekara et al. (2012) proposed to mine bug databases as some of these bugs are only revealed to be vulnerabilities years after. In another work, this idea was experimented on the Linux Kernel for data between 2006 and 2011 (Wijayasekara et al. 2014). They reported a precision of 0.02, but noted that this performance is better than random.

(Meneely et al. 2013) found that, on Apache HTTPD, VCCs were related with bigger commits as non-VCC while tracking 68 vulnerabilities and their 124 manually-found related VCCs.They note as well that bigger commits were related, generally, with the introduction of new features.

VulPecker (Li et al. 2016b) chose to focus on patch hunks and code similarity analysis. It led Li et al. (2016b) to discover 40 vulnerabilities not in the NVD database, 18 of which were still unpatched.

4.5 Machine Learning Application for Vulnerability Analysis

A large body of work in the literature has proposed to use machine learning to discover vulnerability patterns in an entire code base, without considering commits individually. Ghaffarian and Shahriari (2017) provide a thorough literature survey on various approaches in this direction. One of the key finding reported by the authors is that the field of vulnerability prediction models was not yet mature.

Literature approaches have employed learning techniques on diverse programming lan-guages and software systems: Chang et al. (2008) have applied a HMFSM (Heuristic Maximal Frequent Subgraph Mining) to four C programs (make, openssl, procmail and amaya). Their approach uses a a mix of static analysis and data mining to extract patterns that were then associated with their frequency: the more frequent a pattern, the safer it is

considered. In their evaluation, they managed to find 3800 violations of well-known patterns. Zimmermann et al. (2010) proposed to use a measure of code complexity (described by McCabe 1976) to predict the presence of vulnerabilities in Windows Vista. Using Linear Regression, they manage to have a precision below 64% for a relatively low recall of 21% on a ten-fold validation process. Yamaguchi et al. (2013) have presented CHUCKY, an approach to identify anomalous or missing checks on C programs. It is a combination of taint analysis and machine learning that results in finding up to 96% of missing checks by comparing a piece of code to the most similar ones. Scandariato et al. (2014) extracted text from 182 releases of 20 Android applications to generate feature vectors, using a feature discretisation method proposed by Kononenko (1995). This approach achieved good performance for detecting vulnerabilities within a project, but lower performance for inter-project detection. DEKANT was proposed to generate a model out of sliced pieces of PHP applications and WordPress plugins (Medeiros et al. 2016). This model, based on a set of annotated source code, serves as the basis for the discovery of new vulnerabilities.

Researchers have explored various code representations for learning vulnerability properties. Feng et al. (2016) used machine learning on CFGs. Their tool, Genius, identified 38 potentially vulnerable firmware, 23 of which were manually confirmed. Similarly, Lin et al. (2018) have tokenised Abstract Syntax Trees (AST) to feed a deep learning classifier (Bi-LSTM) to obtain a model of vulnerabilities. This model was then applied to a new project and enabled early vulnerability detection. Recently, Ban et al. (2019) also used Bi-LSTM on ASTs from C and C++ datasets. In contrast to these works, Alohaly and Takabi (2017) presented an approach that balances text and structural features. Tested on phpAdmin and Moodle, their results were slightly below those of an usual bag of words technique.

Other papers focused on the importance of the extracted features. For example, Shin and Williams (2011) tried to focus on the correlation between code complexity features and the presence of vulnerabilities. The overall performance was rather low in term of completeness (letting no vulnerable program pass unflagged (Ghaffarian and Shahriari 2017)) with an overall precision of 12%, while the recall reached 67% to 81% depending on the project, respectively Firefox and Wireshark. Though, another paper, namely Moshtari et al. (2013) replicated this study with much more success using Bayesian Networks (as used by Shin and Williams (2011)) only focusing on Firefox and adding more complete information they had on the vulnerabilities through the allocated Common Weakness Enumeration (i.e., the vulnerability type). They even reached greater success changing either for IBK algorithm or Random Tree by Random Committee, by reaching a Recall of 92% and a Precision of 98% for the latter case, but still only on Mozilla. On cross-project attempt (adding Eclipse, Apache Tomcat, Linux kernel 2.6.9 and OpenSCADA) it drops at 32% for the Precision and 7% for the Recall. It is to mention that Mozilla presents a ground truth of on average 2300 vulnerabilities split into 1000 files. Other projects considered on the cross-project analysis do only so from 12 files (OpenSCADA) to 814 (Eclipse written in Java).

Goseva-Popstojanova and Tyo (2018) investigated what features to consider for vulnerability detection, and concluded that the features do not affect significantly the classification performance. The best performing algorithm was different depending not only on the features but more importantly on the dataset.

4.6 Vulnerability Detection at Commit Level

A few articles try to address the issue of automated detection of vulnerabilities at commit level.

Yang et al. (2017) focuses on automatically detecting vulnerability-contributing changes in the Mozilla Firefox project. The tool extracts features from commits and uses a random forests classifier to detect VCCs. By first using an estimated number of potential VCCs present in the code under analysis, they claim to produce fewer False Positives than VCCFinder. Sabetta and Bezzi (2018) consider the code modified by a commit as a text document, and then leverage Natural Language Processing techniques to feed multiple machine learning classifiers. One of Wan (2019)'s contribution is to filter commits by excluding or including those matching a list of keywords. For example, their filtering step can discard up to 92% of commits, hence vastly reducing the effort needed to analyse the suspicious commits. However, in each of these works, the artefact are not available so, cannot compare against neither VCCFinder nor our baseline approach. Moreover, being unavailable, these approaches cannot be used as baseline for the research community.

Other works have directly mentioned and inherited from VCCFinder. Directly trying to improve on VCCFinder, in a 5 pages technical report, Yamamoto (2018) aims at decreasing the number of false-positive results yielded by VCCFinder. To that end, he proposes to separate additions from deletions in the commits to extract code-related features. The results presented in this technical paper are claimed to be slightly better than those of VCCFinder. However, being yet unpublished, and by only proposing a marginal variation with VCCFinder, we opted to only consider VCCFinder for our reproduction/replication work. Zhou and Sharma (2017) compare different algorithms for automatically discovering security issues. Albeit mentioning that VCCFinder uses LinearSVM, they only consider information from the commit message, gathered using regular expressions, and from bug reports. In opposition with VCCFinder and our baseline approach no information is taken from the patch code itself. The experimental results provided in this paper do not allow us to clearly compare the performance of their approach to that of VCCFinder, nor to our baseline.

Finally, even if they do not propose an ML based approach to detect vulnerability at commit level, Hogan et al. (2019) address the issue of the reliability of the labelled data taking VCCFinder as an example. They simplified the version of the project scrapper available online for VCCFinder, re-adapted the code to make it work regarding their focus and manually analysed the commits considered as VCCs. They conclude that only 58% of the commits that would be considered as ground truth, if they relied on VCCFinder's technique, are actually contributing to a vulnerability. This is an issue we did not have to address since we attempted to replicate the performances presented in VCCFinder original paper using data provided by the authors, not to check the validity of the ground truth construction method. The issue raised by Hogan et al. (2019) underlines an important problem for the field that had already been mentioned by Goseva-Popstojanova and Tyo (2018).

5 Conclusion

Vulnerability detection is a key challenge in software development projects. Ideally, vulnerabilities should be discovered when they are being introduced, i.e., by flagging the suspicious vulnerability-contributing commits. VCCFinder, presented in 2015 at the CCS flagship security conference held the promise of detecting vulnerability-contributing commits at scale using machine learning. Since the research direction that this approach initiated has not boomed since then, we have proposed to revisit it. First, we attempted (and failed) to replicate the approach and to replicate the results. Then, we propose to build an alternative approach for the detection of vulnerability-contributing commits using a new feature

sets (whose extraction is clearly replicable) and a semi-supervised learning technique based on co-training to account for the existence of a large set of unlabelled commits. Our experimental results indicate that the proposed approach does not yield as good performance as the ones reported in the VCCFinder publication. Nevertheless, it constitutes a strong and reproducible baseline for the research community. Our artefacts are publicly available at https://github.com/Trustworthy-Software/RevisitingVCCFinder.

References

Alohaly M, Takabi H (2017) When do changes induce software vulnerabilities? In: 2017 IEEE 3rd International conference on collaboration and internet computing (CIC). pp 59–66. https://doi.org/10.1109/CIC.2017.00020

Arusoaie A, Ciobâca S, Craciun V, Gavrilut D, Lucanu D (2017) A comparison of open-source static analysis tools for vulnerability detection in c/c++ code. In: 2017 19th International symposium on symbolic and numeric algorithms for scientific computing (SYNASC), IEEE. pp 161–168

Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Le Traon Y, Octeau D, McDaniel P (2014) Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware tain analysis for android apps. In: Proceedings of the 35th ACM SIGPLAN Conference on programming language design and implementation, association for computing machinery, New York, NY, USA, PLDI '14, pp 259–269. https://doi.org/10.1145/2594291.2594299

Association for Computer Machinery (2020) Artifact review and badging. https://www.acm.org/publications/policies/artifact-review-badging-current, accessed November27, 2020

Balcan MF, Blum A (2005) A pac-style model for learning from labeled and unlabeled data. In: International conference on computational learning theory, Springer. pp 111–126

Ban X, Liu S, Chen C, Chua C (2019) A performance evaluation of deep-learnt features for software vulnerability detection. Concurr Comput Pract Exp 31(19):e5103. https://doi.org/10.1002/cpe.5103

Blum A, Mitchell T (1998) Combining labeled and unlabeled data with co-training. In: Proceedings of the Eleventh annual conference on computational learning theory, association for computing machinery, New York, NY, USA, COLT' 98, pp 92–100. https://doi.org/10.1145/279943.279962

Cadar C, Sen K (2013) Symbolic execution for software testing: Three decades later. Commun ACM 56(2):82–90. https://doi.org/10.1145/2408776.2408795

Cadar C, Dunbar D, Engler D (2008) Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on operating systems design and implementation, USENIX Association, USA, OSDI'08, pp 209–224

Castelli V, Cover TM (1995) On the exponential value of labeled samples. Pattern Recogn. Lett. 16(1):105–111

Chang R, Podgurski A, Yang J (2008) Discovering neglected conditions in software by mining dependence graphs. IEEE Trans. Softw. Eng. 34(5):579–596. https://doi.org/10.1109/TSE.2008.24

Cho CY, Babiundefined D, Poosankam P, Chen KZ, Wu EX, Song D (2011) Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In: Proceedings of the 20th USENIX Conference on security, USENIX Association, USA, SEC'11. pp 10

Fan RE, Chang KW, Hsieh CJ, Wang XR, Lin CJ (2008) Liblinear: A library for large linear classification. J Mach Learn Res 9:1871–1874

Feng Q, Zhou R, Xu C, Cheng Y, Testa B, Yin H (2016) Scalable graph-based bug search for firmware images. In: Proceedings of the ACM SIGSAC Conference on computer and communications security, association for computing machinery, New York, NY, USA, CCS '16, pp 480–491. https://doi.org/10.1145/2976749.2978370

Ghaffarian SM, Shahriari HR (2017) Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM Comput Surv (CSUR) 50(4):1–36

Goseva-Popstojanova K, Tyo J (2018) Identification of security related bug reports via text mining using supervised and unsupervised classification. In: 2018 IEEE International conference on software quality, reliability and security (QRS), pp 344–355. https://doi.org/10.1109/QRS.2018.00047

Groß S (2018) Fuzzil: Coverage guided fuzzing for javascript engines. Master's thesis, Karlsruhe Institute of Technology

Hogan K, Warford N, Morrison R, Miller D, Malone S, Purtilo J (2019) The challenges of labeling vulnerability-contributing commits. In: 2019 IEEE International symposium on software reliability engineering workshops (ISSREW). IEEE, pp 270–275

Holzmann GJ (2002) Uno: Static source code checking for userdefined properties. In: 6th World conference on integrated design and process technology, IDPT '02

Hsu CW, Chang CC, Lin CJ (2003) A practical guide to support vector classification. Tech. rep., Department of Computer Science, National Taiwan University. http://www.csie.ntu.edu.tw/cjlin/papers.html

Kersten R, Luckow KS (2017) Poster: Afl-based fuzzing for java with kelinci. In: ACM Conference on computer and communications security

Kim S, Whitehead EJ Jr, Zhang Y (2008) Classifying software changes: Clean or buggy? IEEE Trans. Softw. Eng. 34(2):181–196. https://doi.org/10.1109/TSE.2007.70773

Klees G, Ruef A, Cooper B, Wei S, Hicks M (2018) Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp 2123–2138

Kononenko I (1995) On biases in estimating multi-valued attributes. In: Proceedings of the 14th International joint conference on artificial intelligence - Volume 2, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, IJCAI'95, pp 1034–1040

Krsul IV (1998) Software vulnerability analysis. Purdue University West Lafayette, IN

Larochelle D, Evans D (2001) Statically detecting likely buffer overflow vulnerabilities. In: Proceedings of the 10th Conference on USENIX security symposium - Volume 10, USENIX Association, USA, SSYM'01

Levin A, Viola P, Freund Y (2003) Unsupervised improvement of visual detectors using co-training. In: IEEE, p 626

Li H, Oh J, Oh H, Lee H (2016a) Automated source code instrumentation for verifying potential vulnerabilities. In: Hoepman JH, Katzenbeisser S (eds) ICT Systems security and privacy protection, Springer International Publishing, Cham. pp 211–226

Li Z, Zou D, Xu S, Jin H, Qi H, Hu J (2016b) Vulpecker: An automated vulnerability detection system based on code similarity analysis. In: Proceedings of the 32nd Annual conference on computer security applications, association for computing machinery, New York, NY, USA, ACSAC '16, pp 201–213. https://doi.org/10.1145/2991079.2991102

Lin G, Zhang J, Luo W, Pan L, Xiang Y, De Vel O, Montague P (2018) Cross-project transfer representation learning for vulnerable function discovery. IEEE Trans Industr Inform 14(7):3289–3297. https://doi.org/10.1109/TII.2018.2821768

Livshits VB, Lam MS (2005) Finding security vulnerabilities in java applications with static analysis. In: Proceedings of the 14th Conference on USENIX security symposium - Volume 14, USENIX Association, USA, SSYM'05, p 18

Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? In: Proceedings of the 2005 International workshop on mining software repositories, association for computing machinery, New York, NY, USA, MSR '05. p 1–5. https://doi.org/10.1145/1083142.1083147

Martin M, Livshits B, Lam MS (2005) Finding application errors and security flaws using pql: A program query language. In: Proceedings of the 20th Annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, association for computing machinery, New York, NY, USA, OOPSLA '05, pp 365–383. https://doi.org/10.1145/1094811.1094840

McCabe TJ (1976) A complexity measure. IEEE Trans Softw Eng SE-2(4):308–320. https://doi.org/10.1109/TSE.1976.233837

Medeiros I, Neves N, Correia M (2016) Dekant: A static analysis tool that learns to detect web application vulnerabilities. In: Proceedings of the 25th International symposium on software testing and analysis, association for computing machinery, New York, NY, USA, ISSTA, vol 2016, pp 1–11. https://doi.org/10.1145/2931037.2931041

Meneely A, Srinivasan H, Musa A, Tejeda AR, Mokary M, Spates B (2013) When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In: 2013 ACM / IEEE International symposium on empirical software engineering and measurement. pp 65–74. https://doi.org/10.1109/ESEM.2013.19

Moshtari S, Sami A, Azimi M (2013) Using complexity metrics to improve software security. Comput Fraud Secur 2013(5):8–17

Neuhaus S, Zimmermann T, Holler C, Zeller A (2007) Predicting vulnerable software components. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA, CCS '07, pp 529–540. https://doi.org/10.1145/1315245.1315311

Padhye R, Lemieux C, Sen K (2019) Jqf: coverage-guided property-based testing in java. pp 398–401. https://doi.org/10.1145/3293882.3339002

Perl H, Dechand S, Smith M, Arp D, Yamaguchi F, Rieck K, Fahl S, Acar Y (2015) Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In: Proceedings of the 22nd ACM SIGSAC Conference on computer and communications security, association for computing machinery, New York, NY, USA, CCS '15, pp 426–437. https://doi.org/10.1145/2810103.2813604

Rieck K, Wressnegger C, Bikadorov A (2012) Sally: A tool for embedding strings in vector spaces. J Mach Learn Res 13(1):3247–3251

Sabetta A, Bezzi M (2018) A practical approach to the automatic classification of security-relevant commits. In: 2018 IEEE International conference on software maintenance and evolution (ICSME). pp 579–582,. https://doi.org/10.1109/ICSME.2018.00058

Sawadogo AD, Bissyandé TF, Moha N, Allix K, Klein J, Li L, Le Traon Y (2020) Learning to catch security patches. arXiv:2001.09148

Scandariato R, Walden J, Hovsepyan A, Joosen W (2014) Predicting vulnerable software components via text mining. IEEE Trans. Softw. Eng. 40(10):993–1006. https://doi.org/10.1109/TSE.2014.2340398

Shin Y, Williams L (2011) An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In: Proceedings of the 7th International workshop on software engineering for secure systems, pp 1–7

Signoles J, Cuoq P, Kirchner F, Kosmatov N, Prevosto V, Yakobowski B (2012) Frama-c: a software analysis perspective, vol 27. https://doi.org/10.1007/s00165-014-0326-7

Torvalds L, Triplett J, Li C, Oostenryck LV (2003) Sparse - a semantic parser for c. https://sparse.wiki.kernel.org/index.php/Main_Page accessed january 2020

Vapnik V (2013) The nature of statistical learning theory. Springer science & business media, New York

Wan L (2019) Automated vulnerability detection system based on commit messages. PhD thesis

Wang S, Chollak D, Movshovitz-Attias D, Tan L (2016) Bugram: Bug detection with n-gram language models. In: Proceedings of the 31st IEEE/ACM International conference on automated software engineering, association for computing machinery, New York, NY, USA, ASE 2016, pp 708–719. https://doi.org/10.1145/2970276.2970341

Wheeler DA (2001) Flawfinder. https://dwheeler.com/flawfinder/, accessed April 2020

Wijayasekara D, Manic M, Wright JL, McQueen M (2012) Mining bug databases for unidentified software vulnerabilities. In: 2012 5th International conference on human system interactions. pp 89–96. https://doi.org/10.1109/HSI.2012.22

Wijayasekara D, Manic M, McQueen M (2014) Vulnerability identification and classification via text mining bug databases. In: IECON 2014 - 40th Annual Conference of the IEEE Industrial Electronics Society, pp 3612–3618. https://doi.org/10.1109/IECON.2014.7049035

Yamaguchi F, Wressnegger C, Gascon H, Rieck K (2013) Chucky: Exposing missing checks in source code for vulnerability discovery. In: Proceedings of the 2013 ACM SIGSAC Conference on computer & communications security, association for computing machinery, New York, NY, USA, CCS '13, pp 499–510. https://doi.org/10.1145/2508859.2516665

Yamaguchi F, Golde N, Arp D, Rieck K (2014) Modeling and discovering vulnerabilities with code property graphs. https://doi.org/10.1109/SP.2014.44

Yamamoto K (2018) Vulnerability detection in source code based on git history. https://doi.org/10.13140/RG.2.2.28338.09922. (Unpublished)

Yang L, Li X, Yu Y (2017) Vuldigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes. In: GLOBECOM 2017 - 2017 IEEE Global communications conference, pp 1–7. https://doi.org/10.1109/GLOCOM.2017.8254428

Zalewski M (2017) American fuzzy lop. http://lcamtuf.coredump.cx/afl/

Zhang T, Oles F (2000) The value of unlabeled data for classification problems

Zhou Y, Sharma A (2017) Automated identification of security issues from commit messages and bug reports. In: Proceedings of the 2017 11th Joint meeting on foundations of software engineering, association for computing machinery, New York, NY, USA ESEC/FSE 2017, pp 914–919. https://doi.org/10.1145/3106237.3117771

Zhu X, Feng X, Jiao T, Wen S, Xiang Y, Camtepe S, Xue J (2019) A feature-oriented corpus for understanding, evaluating and improving fuzz testing, pp 658–663

Zimmermann T, Nagappan N, Williams L (2010) Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In: 2010 Third international conference on software testing, verification and validation, pp 421–428. https://doi.org/10.1109/ICST.2010.32