Evaluating and Improving Commit-based Static Analysis

Bachelor thesis by Jan Philipp Wagner

1. Review: Prof. Dr. Max Mühlhäuser
2. Review: Nikolaos Alexopoulos
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science
Department
Telecooperation Lab
SPIN

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Jan Philipp Wagner, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt,

Jan P. Wagner

Abstract

Static analysis of software deals with the task of finding vulnerabilities in source code without actually executing it. In the recent past, progress has been made in this field by analyzing *git commits* instead of specific files. Here, the state of the art uses machine learning- and text mining techniques to predict whether a specific commit is likely to make the software vulnerable based on its text features and metadata. These tools, although promising, have not seen widespread adoption. Reasons for this might be their unavailability to the masses along with them not having been extensively validated. Therefore, this thesis has the goal of creating such a tool of our own that has been validated and tested on ground truth data and is available to everyone. This is especially challenging because ground truth data for commits that introduced a vulnerability is extremely rare. Our approach to this problem is scouring the web for datasets that have been constructed manually as well as automatically tracing vulnerabilities back to their origin. We find that previous methods for tracing vulnerabilities only have a 40% accuracy. We are able to slightly improve this accuracy as well as identify a subset of vulnerabilities where the accuracy is 63%. Further, we find that our replication of the state of the art classifier is able to identify 27% of Vulnerability-Contributing Commits made to the Linux kernel repository in 2016 and 2017 with a 1.4% precision. We make slight improvements to this model to achieve the same recall with 2.2% precision.

Contents

1. Introduction

Static analysis of software deals with the task of finding vulnerabilities in source code without actually executing it. A fairly new approach to this field of research originated in the 2015 paper *VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits* [11]. VCCFinder uses a linear Support Vector Machine model that takes into account GitHub metadata and combines it with code-metric data to identify *git commits* that are likely to introduce a vulnerability. The idea is that commit metadata can give loads of interesting insight into the nature of a patch such as the experience of the contributing developer or the frequency of how often the code changes.

Despite significantly lowering the false positive rate of conventional static analysis tools like Flawfinder [3] on the dataset created by Perl et al., VCCFinder (or similar tools [17]) has not seen widespread adoption in any active open source projects to the best of our knowledge. In addition to the VCCFinder GitHub repository[1] being incomplete, a reason for this might be the limited dataset the tool was tested on, as it contained non ground truth data. Therefore, we want to evaluate a replication of their model on a ground truth dataset we created ourselves. This is challenging, as past attempts to recreate VCCFinder failed due to the inability of building a sufficiently sized dataset for training [12]. We approach this problem by combining manually produced datasets from security researchers [8], [14] with data we gained from automatically tracing vulnerabilities back to their origin. In order to accomplish the latter, we utilize an improved version of the *git blame* based heuristic that was also used in the creation of the dataset VCCFinder was trained on.

Lastly, we contribute a model of our own that is closely based on VCCFinder but slightly improves its accuracy through the addition of new or more expressive metadata features and adjustments to the bag-of-words model that VCCFinder uses to represent code. We make the model as well as our dataset publicly available for the community to use[2].

[1] `https://github.com/hperl/vccfinder`

[2] `https://github.com/jp-wagner/vccrosshair`

2. Related Work

In 2015 Perl et al. introduced VCCFinder [11]. As it was the first static analysis tool to perform predictions on specifically a commit-level, it constitutes the basis for this thesis. The tool utilizes concepts from machine learning (linear Support Vector Machines) as well as natural language processing (bag-of-words model) to identify commits that are prone to introduce vulnerabilities. What separates VCCFinder from other approaches to static analysis is that it not only makes predictions based on data extracted from code but also takes commit metadata into account. Perl et al. conceptualized that metadata points, such as the experience of the authoring developer or how frequently code changes, can be valuable for finding vulnerabilities. They then demonstrated this by showing that their model produces 99% less false positives than Flawfinder [3], a conventional static analysis tool.
This was tested on a dataset containing 219 Vulnerability-Contributing commits (*VCCs*) and 79220 regular commits from 68 different repositories. To obtain these VCCs, they used a *git blame* based heuristic. This heuristic takes a commit that fixed a vulnerability, and through the use of the *git blame* command, traces back to the commit where the vulnerability originated. Perl et al. found that their heuristic identifies a correct VCC very reliably (96.9% accuracy) and the dataset created with it was made available for future research.

Later Yang et al. further capitalized on the ideas brought forward by Perl et al. [17]. They also developed a tool for classifying commits into prone to introduce a vulnerability and not prone to introduce a vulnerability by accessing valuable *git* metadata. However, they opted for a Random Forests technique instead of a linear Support Vector Machine. Additionally, they provide a regression model that focuses on the return on investment of auditing lines of code. This emphasis on effort-awareness aims to ease the workload of developers even more. However, due to unavailability of the dataset by Perl et al. at the time, they could not compare their tool with VCCFinder but report similar results on their dataset.

For the gathering of their training data, Yang et al. made some slight adjustments to the *git blame* based heuristic also used for VCCFinder. By putting additional constraints such as avoiding to blame comments, they tried to make it more precise. They report a 95.5% accuracy of their heuristic.

Meneely et al. were the first to analyze *git* metadata in relation to vulnerabilities [7, 9, 8]. They coined the term Vulnerability-Contributing Commit when they linked 127 vulnerabilities in the Apache HTTP server back to the commits where they first originated. In the analysis of these commits, they could find a correlation of so-called *interactive code churn* metrics, which are derived from *git* metadata, with faults and vulnerabilities. They also found that VCCs are unusually large and often contributed by developers that have low experience with the repository.

A machine learning approach to predicting software vulnerabilities based on text mining of source code was first taken by Scandariator et al. [13]. Also using a bag-of-words model, they examined 20 Android apps over the course of two years and demonstrated that their method shows good performance for both precision and recall.

Lastly, Zhou et al. compared a tool they developed to VCCFinder and found that it had an even better performance (54.55% increase in precision at same recall) [19]. However, the goal of their classifier seems to be to find commits (and bug reports) that are related to vulnerabilities. Since this would include commits that merely fixed a security bug instead of introducing one, we do not think that a comparison to VCCFinder is viable. Thus, we still consider VCCFinder the state of the art for identifying VCCs.

3. Data

One of the key factors for achieving the goal of this thesis is a sound dataset. On the one hand we need data to evaluate the performance of VCCFinder and on the other hand we need that same data to train and validate our own classifier. Specifically, we are looking for a dataset of commits that introduced vulnerabilities in the past. We will refer to these commits as Vulnerability-Contributing Commits or *VCCs*. The goal is to use these VCCs to train a classifier that is able to distinguish between commits that did not introduce vulnerabilities and commits that did so. After that, the classifier can be used to predict whether a new commit is prone to introduce a vulnerability as well as finding not yet discovered vulnerabilities in older commits.
We can obtain VCCs in two different ways which are outlined in the following subsections.

3.1. Manually Identified VCCs

The straightforward approach is to look for datasets of VCCs that have been manually identified by security researchers. Unfortunately, datasets like that are rare to come across, and thus their size might not be sufficient for training a classifier. Still, we were able to find two such datasets:

3.1.1. Ubuntu CVE Tracker

The largest dataset for a single project that we were able to extract is from the Ubuntu CVE Tracker [14]. In this project, the Ubuntu Security Team gathered lots of Common Vulnerabilities and Exposures (CVE) data for vulnerabilities affecting the Linux kernel. Here we can find, among other information, mappings from the commit that fixed a vulnerability (*fixing commit*) to its VCC (3.1). With the help of Python and some regular expressions, this enabled us to build our first dataset containing 1025 of these mappings.

```
Patches_linux:
 break-fix: 3a50597de8635cd05133bd12c95681c82fe7b878 23567fd052a9abb6d67fe8e7a9ccdd9800a540f2
```

Figure 3.1.: excerpt from retired/CVE-2016-0728 from the Ubuntu CVE Tracker.

3.1.2. Vulnerability History Project

We also were able to extract mappings from fixing commits to VCCs from Andrew Meneely's Vulnerability History Project[1]. He is a security researcher currently working at the Rochester Institute of Technology. A lot of his prior work also focused on examining VCCs [7, 9, 8]. The Vulnerability History Project contains mappings for Chromium, Apache HTTPD, Apache Struts and Apache Tomcat. Using it, we were able to add 373 new mappings that bring a variety of projects as well as a variety of programming languages (C, C++ and Java instead of just C) to our dataset.
However, it should be noted that Meneely's definition of a VCC is slightly different from ours. In his paper [9] it is stated that:

> "Some vulnerabilities might have multiple regions in a given file where a fix was required. In that case, we treat each of those regions with a separate detection script[2] to maximize the number of potential VCCs we can find."

Thus, his approach is, by design, prone to generate multiple VCCs to a single vulnerability. In contrary, the Ubuntu CVE Tracker usually looks for a single point of failure when marking a commit as VCC, which seems more fitting for our purposes. Nevertheless, we think that Meneely's mappings can still be beneficial in training our classifier.

3.2. Automated Tracing of VCCs

The second approach is the one also used in [11]. Here, the idea is to first find a large amount of fixing commits and then use a *git blame* based heuristic to automatically map these fixing commits to VCCs. *Git blame* finds which commit last changed a specific line in a given file and so it can be used for tracing a vulnerability back to its origin.
The reason why we use this approach is that fixing commits are much easier to identify

[1] `https://github.com/VulnerabilityHistoryProject`
[2] Although Meneely et al. did use detection scripts, they also manually reviewed the mappings they produced.

than VCCs. That is because most of the time the fixing commit is mentioned in the National Vulnerability Database[1] (NVD) entry for a vulnerability or the commit message of a fixing commit mentions the CVE ID of the vulnerability it fixed. On the other hand, a commit that introduces a vulnerability usually does not mention so in its commit message, for obvious reasons. More on how we obtained fixing commits can be read in Manuel Brack's Bachelor thesis [1].

Due to the fact that this approach is automated, we can use it to build a much larger dataset than what is possible by manually finding VCCs. Perl et al. claim that they had great success (97% accuracy) with their heuristic, therefore we want to explore it a bit more in the next chapter.

[1] https://nvd.nist.gov/

4. Heuristic

In order to create a large dataset of Vulnerability-Contributing Commits (VCCs), we want to automate the process of mapping a commit that fixed a vulnerability to the commit that first introduced it. In their paper, Perl et al. claim that the heuristic (depicted in 4.1) they used for that exact purpose has a 96.9% accuracy. They determined this by taking a 15% random sample of all VCCs found by the heuristic and manually checking them. Their heuristic for mapping a fixing commit to a VCC is as follows:

1. Ignore changes in documentation such as release notes or change logs.
2. For each deletion, *blame* the line that was deleted.
3. For every continuous block of code inserted in the fixing commit, *blame* the lines before and after the block.
4. Finally, mark the commit *vulnerable* that was blamed most in the steps above. If two commits were blamed for the same amount of lines, blame both.

They also mention that further improving this heuristic is an interesting avenue for future research. This next section is dedicated to that exact purpose. The two main goals are evaluating the heuristic and possibly improving it.

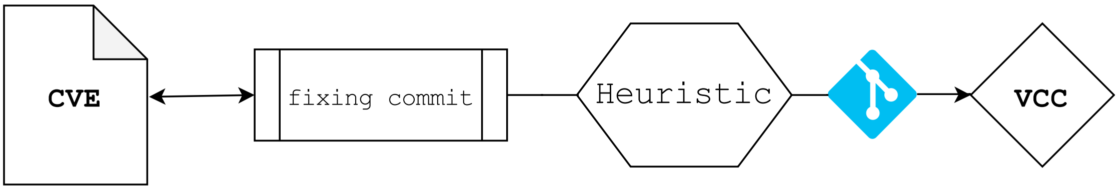


Figure 4.1.: Git blame based heuristic

4.1. Evaluation

4.1.1. Comparing the Mappings

Upon request, Henning Perl (author of [11]) provided us with the dataset they produced utilizing their heuristic. We then implemented a clone of the heuristic according to the description given in the paper. To test whether or not our clone is an accurate representation of what was used for VCCFinder, we check if we are able to reproduce their data with our own implementation.

The dataset contains 692 mappings of fixing commit to VCCs from various repositories. If we run our implementation of the VCCFinder heuristic over the same fixing commits, we find that in 97% of the cases, we can trace the vulnerability back to the same VCC. We chalk up the missing 3% to manual remapping done by Perl et al. to clean up their data. Thus, we deem our implementation of the VCCFinder heuristic an accurate clone of the one used by Perl et al.

However, when comparing these mappings to the mappings that we extracted from the Ubuntu CVE tracker, we find a big discrepancy. Out of the 142 fixing commits that were present in both sets, only 52.82% are mapped to the same VCC (4.2). This means that 47.18% of the time the heuristic used by Perl et al. produced different mappings than the manual analysis that was performed or at least approved by the Ubuntu team.

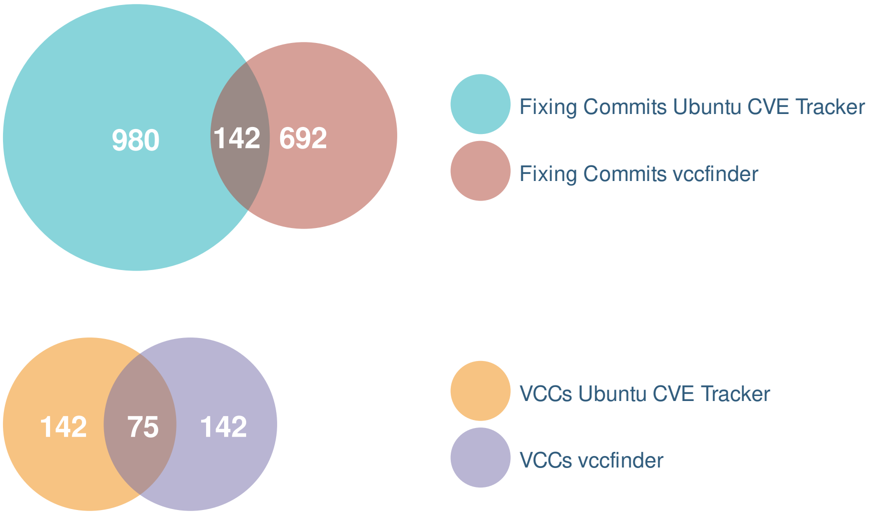


Figure 4.2.: Overlap of the two datasets

In order to investigate the cause of the discrepancy, we manually analyze 10 fixing commits that were mapped to different VCCs by the heuristic and the Ubuntu team. We find that in 9 out of ten cases, the Ubuntu mapping was accurate, while the one produced by the heuristic was not and in one case both possible VCCs were valid (A.1). Reasons for the rather low accuracy of the heuristic may be:

- A fix might be necessary in files that are different from the ones where the vulnerability was introduced, thus blaming in these files will lead to wrong commits.
- The heuristic cannot trace code that has been copied from other files.
- Some vulnerability types, such as concurrency or semantic, seem to be inherently more difficult to trace.

Further testing against the Ubuntu dataset with our clone of the heuristic shows that its accuracy is closer to 40% than the 96.9% claimed in [11]. Thus, we can conclude that producing mappings from fixing commit to VCC seems much harder than what is implied in previous work [11] [17]. A possible reason for this might be confirmation bias. Developers also seem to already be aware of this fact, as otherwise regression tracking (which finding VCCs falls under) would not be the highly discussed[1] issue that it is. Fortunately, there are types of classifiers that can deal with some amount of noise which would inevitably be produced by such a heuristic. This means that it is not completely useless for our purposes. Still, noise in the validation- or test set is unacceptable. Thus, it is highly important to us to evaluate VCCFinder as well as our classifier on ground truth data only.

4.2. Analysis: What Makes a Vulnerability Hard to Trace?

In the previous section, we showed that the heuristic for tracing vulnerabilities back to their VCCs used in [11] has a much lower accuracy than what was previously assumed. One step towards finding out why the heuristic makes mistakes and possibly improving the accuracy is to determine the characteristics of commits that are accurately mapped versus ones that are not. For that, we identified two groups of fixing commits within the Ubuntu dataset:

The first group consists of all fixing commits for which the heuristic was accurate, meaning that it blamed a corresponding VCC the most often out of all possible commits. We call this group *easy commits*. It consists of 537 different commits.

The second group consists of the fixing commits for which the heuristic did not blame the corresponding VCC a single time. We call this group *hard commits*. It consists of 296 different commits.

We analyzed the commits in each group by looking at the Common Weakness Enumeration (CWE) ID, Common Vulnerability Scoring System (CVSS) score and type of vulnerability[2] they fixed, as well as the number of lines of code they added/removed/changed and how many files they altered.

[1] https://lore.kernel.org/lkml/3519198.TemPj1OATJ@vostro.rjw.lan/
https://yarchive.net/comp/linux/regression_tracking.html
https://github.com/mozilla-b2g/bisect_b2g

[2] We obtain the data for the type of a vulnerability from cvedetails.com. The vulnerability type is a more coarse grained categorization than the CWE.

4.2.1. CWE ID and Vulnerability Type

The CWE ID of a vulnerability describes a common software weakness that is being abused in it. The whole dataset contains 44 different CWE IDs of which 31 are also present in *hard commits* and 34 in *easy commits* (4.2).
One might hypothesize that vulnerabilities fixed in *hard/easy commits* are more common to have a specific CWE ID. To explore this aspect we used Pearson's chi-squared test. The null hypothesis holds iff the observed frequency of a specific CWE ID does not significantly ($\alpha = 0.05$) deviate from the expected frequency. We obtain the expected frequencies by looking at the general distribution of CWE IDs across all commits in the dataset. We found that we have to reject the null hypothesis in four cases:

Hard commits are significantly more likely to fix vulnerabilities with CWE 264 (Permissions, Privileges, and Access Controls) ($\rho = 0.00008$).
Notes: Access control vulnerabilities seem to often require a fix that is independent from where the vulnerability originated and thus the heuristic struggles to map them correctly.

Easy commits are significantly less likely to fix vulnerabilities with CWE 264 (Permissions, Privileges, and Access Controls) ($\rho = 0.008$).
Notes: Same as above.

Easy commits are significantly more likely to fix vulnerabilities with CWE 190 (Integer Overflow or Wraparound) ($\rho = 0.047$).
Notes: Integer Overflows or Wraparounds are usually fixed at exactly the same place where they were introduced by adding a simple check. Thus, it makes sense that the heuristic likely maps these commits correctly.

Easy commits are more likely to fix vulnerabilities with CWE 200 (Information Exposure) ($\rho = 0.038$).
Notes: Oftentimes, vulnerabilities with CWE 200 are fixed by initializing memory properly where information was leaked before the fix. Again, these fixes are small and exactly where the vulnerability originated and thus easy for the heuristic to handle.

Additionally, we find a more general correlation when applying Pearson's chi-squared test to analyze the distribution of vulnerability types across the two groups (4.1):

Easy commits are significantly more likely to fix vulnerabilities of the type *Obtain Infor-*

mation ($\rho = 0.0227$)
Notes: Same as CWE 200.

Type of vulnerability	All	Hard	Easy
Obtain Information	103	22	68
Overflow	165	51	95
Bypass a restriction	28	8	15
Denial Of Service	485	161	230
Execute Code	32	8	19
Memory Corruption	70	26	34
Gain privileges	90	31	34
Directory Traversal	2	1	1

Table 4.1.: Distribution of vulnerability type

CWE	All	Hard	Easy	CWE	All	Hard	Easy
16	2	0	1	369	4	1	2
17	8	2	4	388	4	0	3
19	5	0	3	399	64	24	31
20	72	19	38	400	11	0	4
22	2	1	1	404	5	1	1
59	1	1	0	415	14	3	8
94	2	1	0	416	62	23	29
119	120	40	64	476	56	15	30
120	5	1	4	532	1	1	0
125	36	9	22	534	1	1	0
129	4	0	2	617	1	0	1
189	32	15	13	665	2	1	0
190	17	1	15	682	2	0	2
191	1	0	1	704	2	2	0
200	61	13	44	754	1	1	0
264	56	32	16	763	1	0	0
269	4	1	2	772	6	2	3
276	4	0	4	787	25	9	13
284	9	4	5	835	6	1	3
287	1	0	0	843	1	0	1
310	9	3	6	862	3	0	2
347	1	1	0	None	237	56	137
362	45	11	22				

Table 4.2.: Distribution of CWEs

4.2.2. Number of Altered Files

One could also hypothesize that the heuristic has a harder time finding a correct VCC, if the fixing commit changed a lot of files instead of, for example, only making alterations in a single one. To explore this hypothesis we use the Mann-Whitney U test. The null hypothesis holds iff the number of altered files is distributed equally between *easy/hard* commits and the total population. We say that an effect is significant if $\rho < 0.0125$ corresponding to 0.01/4 accounting for a Bonferroni correction for multiple testing for altered files and added/removed/changed lines of code. We have to reject the null hypothesis for easy commits:

Easy commits alter significantly less files (ρ = 0.0001).
Notes: This observation intuitively makes sense. If a fix is distributed over multiple files one would expect that the vulnerability is harder to trace back to a VCC. Another explanation could be that the fixing commit made changes in other files that are actually unrelated to the fix. The average number of altered files (4.3) also suggests a difference in the distribution.

Commits	Average	Median
All	1.9	1
Easy	1.3	1
Hard	2.23	1

Table 4.3.: Averages and Medians of altered files

4.2.3. Added/Removed/Changed Lines of Code

To explore whether or not the distribution of added/removed/changed lines of code in *easy/hard* commits differs from all commits, we again use the Mann-Whitney U test. We apply the same Bonferroni correction for multiple testing with $\rho < 0.0125$ indicating a significant difference. We have to reject the null hypothesis two times:

Easy commits add significantly less lines of code (ρ = 0.0007).
Notes: Since the heuristic can only blame lines before and after added lines, they lead to less precise blames than deleted lines. This is because the lines before and after are not necessarily related to the newly added lines. So it makes sense that easy commits add less lines.

Easy commits change significantly less lines of code (ρ = 0.006).
Notes: A possible reason for this might be, that the more lines are changed, the more likely it is that multiple commits are being blamed. So the chance that the top blamed commit is a true VCC might become smaller. Note, that this observation most likely would have been missed by just looking at averages and medians (4.4), as they are susceptible to outliers.

Commits	Added		Removed		Changed	
	Average	Median	Average	Median	Average	Median
All	11.46	5	80.37	2	91.84	7
Easy	7.07	4	11.57	1	18.63	6
Hard	12.60	4	6.78	1	19.38	6

Table 4.4.: Averages and Medians for lines added/removed/changed of code

4.2.4. CVSS Score

For analyzing whether or not the CVSS score of the vulnerabilities are differently distributed across the sets, we again use the Mann-Whitney U test but did not find any correlations. The deviation between averages and medians (4.5) is also minimal.

Commits	Average	Median
All	5.45	4.9
Easy	5.44	4.9
Hard	5.29	4.9

Table 4.5.: Averages and Medians of CVSS score

4.2.5. Limitations

The dataset we extracted from the Ubuntu CVE Tracker is decently large and shows strong indication that it is highly accurate (A.1). However, we have to consider some limitations when using it:

Source bias: Since the entire dataset consists of vulnerabilities found in the Linux kernel, there is a heavy bias towards certain types of vulnerabilities. For example, the dataset does not contain a single Cross-Site-Scripting vulnerability (for obvious reasons), even though they make up 12.5% of all reported vulnerabilities as a whole [4].

Completeness: Even when looking only at the Linux kernel, our dataset is far from complete. There are still many mappings of fixing commit to VCC that the Ubuntu CVE

Tracker does not contain. Of the 2357 vulnerabilities found in the Linux kernel [2] we cover about one third.

Quality: Even though we deem the quality of the mappings by the Ubuntu CVE Tracker as very high, we cannot assume that they are 100% accurate. Mapping fixing commits to their VCC is by no means a trivial task and often even professionals have differing opinions on where and when a vulnerability was first introduced.

4.2.6. Conclusion

From these findings, ideas can be drawn on how we might be able to improve the heuristic: Judging from one of the strongest correlation we found, that easy commits add far less lines of code, we can hypothesize that blames due to additions of code are less likely to point to the correct VCC. Thus, we might want to take blames stemming from added lines of code less into consideration.

Knowing that easy commits change far fewer files does not help us improve the heuristic directly. Nonetheless, this gives us information about which kind of fixing commits we can more likely accurately map to their VCC and so might help build a training set for our classifier that has less noise. Knowing which CWEs are more prominent in commits that are harder to map can also be helpful for the same reasons.

It should be noted that only 173 commits were neither *hard* or *easy*. This vastly limits the margin the heuristic can be improved, as only these commits are likely to additionally be mapped correctly by an improved heuristic.

All in all, insight gained from this analysis can be useful for improving the heuristic used by Perl et al. And while it mostly showed us significant differences for commits that the heuristic can map correctly already, we can still apply this knowledge to build a dataset with less noise.

4.3. Improving the Heuristic

4.3.1. Approach

In order to improve the heuristic in a meaningful way, we use the concept of training-, validation and test sets from the area of machine learning. We choose to use this approach even though our heuristic does not utilize machine learning techniques because we want to ensure that it will also perform well on commits that are not present in our dataset. Since we deemed the Linux kernel dataset we extracted from the Ubuntu CVE tracker very high quality, our test set consists purely of mappings from it. More specifically, we choose the mappings for the most recent fixing commits (every fixing commit to a CVE from 2019) so that we assure that our heuristic will work on future commits as well. The validation set also contains only mappings from the Ubuntu dataset and the training set consists of the remaining Ubuntu mappings and the mappings created by Andrew Meneely [8].

4.3.2. Improvement Ideas

Before diving into what actually ended up increasing the performance of the heuristic, in this section we want to discuss all the different ideas that we tried out and why we tried them. What follows is a list of these ideas. It should be noted that some of them have already been implemented by Yang et al. [17], we mark these with an asterisk:

-w flag*: Setting the -w flag when using *git blame* ignores white space when comparing the parent's version with the child's version of a line to find where the lines came from.
Rationale: Since adding/removing white space does not change the functionality of the code (at least in the languages we are considering), using the -w flag might yield better results.

-M flag: Setting the -M flag when using *git blame* detects moved or copied lines within a file.
Rationale: If a line was copied from within the file, we might want to blame the commit that added the original line and not the copy.

-C flag: Setting the -C flag when using *git blame* detects lines moved or copied from other files that were modified in the same commit.
Rationale: If a line was copied from another file, we might want to blame the commit that added the original line and not the copy.

Increasing the blame radius: Instead of only blaming one line before and after inserted blocks, blame multiple lines before and after each block.
Rationale: In our analysis (4.2) we concluded that the VCCFinder heuristic often times does not blame the true VCC a single time, thus maybe we should try to blame more lines.

Ignore comments*: Comments are not taken into account.
Rationale: Comments do not change the functionality of the code, thus should not be taken into account when determining the VCC.

Weighted blaming: Take specific lines more heavily into account.
Rationale: We might be able to find a general rule for which lines are more likely to lead us to a VCC and thus should be taken more heavily into account.

> **Weight lines containing a specific keyword more heavily:**
> Rationale: For vulnerabilities of a specific type/with a specific CWE some lines containing certain keywords might be more important in determining the VCC than others (e.g. removals of lines containing "free" in a fix for a "use after free" vulnerability).
>
> **Weight additions less heavily**
> Rationale: Blames around added lines might be less important for determining the VCC. This is also what the analysis 4.2 suggests.
>
> **Weight additions more heavily**
> Rationale: We should still test if blames around added lines might be more important for determining the VCC.
>
> **Analyze commit message:** If specific files or functions are mentioned in the commit message of the fixing commit, weight blames in these files/functions more heavily.
> Rationale: If the commit message of the fixing commits mentions specific files, these might be more likely to contain lines that lead to the VCC.
>
> **Analyze NVD summary:** If specific files or functions are mentioned in the NVD summary of the vulnerability, weight blames in these files/functions more heavily.
> Rationale: If the NVD summary of the vulnerability mentions specific files, these might be more likely to contain lines that lead to the VCC.
>
> **Adjust weights according to CWE:** If a vulnerability has a specific CWE, adjust weights for blames from deletions/additions.
> Rationale: The importance of blames due to deletions/additions might correlate with the CWE of the vulnerability.

4.3.3. Own Heuristic

In order to improve upon the heuristic used for VCCFinder, we do a number of exploratory experiments combining the aforementioned ideas. We measure the validity of an idea or a combination by its performance on the training- and the validation set. It is only valid if it raises accuracy on both sets. The combination of ideas that performs the best is our final heuristic. During this process we make two main observations:

Observation number 1:
The maximum accuracy that could *theoretically* be achieved with this method is around 70%. That is because in 30% of the cases we examined, the true VCC was not blamed a single time by the heuristic[1]. This means that in those cases the true VCC can never be the most blamed commit. Moreover, this 70% accuracy would only be possible if the accumulated weight for the true VCC was the highest out of all commits every time, which is very unlikely. This fact significantly limits the margin of how much the heuristic can be improved.

Observation number 2:
Possibly what surprised us the most while working on this, is that ignoring lines that do not change the functionality of the code (i.e. comments) does not increase the accuracy but rather decreases it. This at first seems counterintuitive. If removing/adding comments can never fix a vulnerability, then why should blaming them lead to the origin of the vulnerability? A possible explanation for this is that, often times when a vulnerability is being fixed, comments that were made when introducing that vulnerability are also removed. Blaming those removals then also leads to the true VCC.

What worked?
Unfortunately, a lot of the ideas listed in 5.4 did not improve the heuristic on the training- as well as the validation set. Oftentimes, we were able to increase the accuracy on the training set but then the accuracy on the validation set would decrease. This is equivalent to overfitting a classifier and indicates bad generalization, meaning that the heuristic would likely also perform worse on entirely new commits. However, two of the ideas from the previous section did improve the heuristic on both sets:

1. Setting the -w flag when blaming.
2. Analyzing the NVD summary for a given vulnerability. For that we parse the CVE sum-

[1] Increasing the blame radius did not have a significant effect on that.

mary with regular expressions to search for mentions of vulnerable functions. Changes to mentioned functions are then weighted heavier than regular changes. We tried out several different weights and found that we achieved the best results with a weight of three.

The overall improvements to the heuristic are very minor. Table 4.6 shows a comparison of our heuristic and the one used in [11]. It should be noted that when the heuristic blames multiple commits the same amount of times, we create a mapping for each commit. This is because, in practice, we cannot manually analyze all of these cases like Yang et al. did [17], as this happens quite frequently. No manual intervention of course further lowers the accuracy. In total as well as on the test set alone our heuristic increases accuracy by 1.4%.

Heuristic	Training	Validation	Testing	Total
VCCFinder	True: 434 False: 738	True:166 False: 234	True: 73 False: 69	True: 673 False: 1041
Ours	True: 446 False: 702	True: 167 False: 233	True: 74 False: 66	True: 687 False: 1001

Table 4.6.: Comparison between the heuristic used for VCCFinder and our heuristic.

4.3.4. Confidence Value

Since the amount of noise even our improved heuristic produces is rather high, we are concerned that this might negatively impact the training of our classifier. In order to mitigate this, we assign a binary confidence value to each VCC the heuristic finds. The goal is to mark VCCs that are more likely to be correct. To ensure that this approach generalizes, we use the same training, validation, test split as before. The process of deriving whether or not we are confident in a VCC is as follows:

We are *not confident*, if two or more commits are blamed most often, or if the CWE of the vulnerability is 264.
Rationale:

- We lack the manpower to manually examine every time the heuristic blames multiple commits which happens quite frequently.

- We use the knowledge gained from the analysis in 4.2 to determine which vulnerabilities are hard to trace based on their CWE.

We are *confident*, if we are not *not confident* and the difference in blames for the most blamed commit and the second most blamed commit is greater than the total blames divided by three and the number of different commits blamed by the heuristic is less than five.
Rationale:

- Running the heuristic over our training set showed that it has higher accuracy if the total number of different commits it blames is low. This is also suggested by our analysis which showed that *easy* commits change less lines, thus the heuristic likely blames less different commits for them.
- If the difference between the top most blamed and the second most blamed commit is high, we can be more confident that the most blamed commit is correct, as a small difference would indicate that the second most blamed commit is almost as viable as the top blamed.
- The numerical values we choose (three and five) show the best results on our training- and validation set.

Table 4.7 shows the precision and recall this method has in determining whether a VCC found by the heuristic is correct or not. When interpreting this data we find that we are able to identify a 31.04% subset of VCCs, in which 68% (precision) of VCCs are correct. This is a significant increase over the 39.26% accuracy of using the VCCFinder heuristic. We are aware that the distribution of CWEs might be different in this subset and the total and that this could potentially bias our classifier later on.

	Training	Validation	Testing	Total
Correct & confident	209	106	39	354
Correct & not confident	237	61	35	333
False & confident	110	47	13	170
False & not confident	592	186	53	831
Precision	0.66	0.69	0.75	0.68
Recall	0.47	0.63	0.53	0.52

Table 4.7.: Precision and Recall of confidence value

4.4. Building the Dataset for Our Classifier

This section will explain how we use the VCCs available to us to build the dataset for training- and testing the classifier.

4.4.1. Test Set

Like we established earlier, we do not tolerate any noise in the VCCs of our test set. This is why we choose to only include VCCs acknowledged by the Ubuntu Security Team here. Additionally, we decided to simulate a real world scenario, thus our evaluation aims to answer the question of how our classifier would have fared if it had been actively used in the development of the Linux kernel. Therefore, our test set consists of every commit that was made to the Linux kernel repository in the year 2016 and 2017. The only exclusion being commits that changed over 2000 lines of code. The rationale behind this is that flagging commits larger than this is not very helpful for developers due to the sheer amount of code that would have to be audited. Perl et al. seem to have placed the same restriction judging from the VCCFinder repository. We choose 2016 and 2017 because the number of known VCCs for a more recent time frame is not sufficient. Table 4.8 shows the distribution of VCCs to non-VCCs for the test set. Note, that every non-VCC actually has to be considered an unclassified commit. This is because there is a possibility that a commit introduced a vulnerability which has not been linked to it yet or is entirely unknown.

Repository	**VCCs** ground truth	**non-VCCs** unclassified
linux	129	127602

Table 4.8.: Test set

4.4.2. Training Set

For the training set we use our improved heuristic on fixing commits accumulated by Manuel Brack for his thesis [1]. We also add the VCCs found by Andrew Meneely for chromium and httpd as well as the VCCs from the Ubuntu CVE Tracker that are not from 2016 or 2017. Furthermore, we place the same restriction of using no commits that changed more than 2000 lines. In total we found 5745 VCCs across 8 different repositories (4.9), which is a significant increase from the 640 VCCs found for VCCFinder, although we cover less repositories. We categorize these VCCs in *ground truth, confident and not confident* in order to assign different weights to them later in the training process. For the non-VCCs we simply choose a number of random commits from each repository that do not overlap with the known VCCs.

Repository		**VCCs**		**non-VCCs**
	not confident	confident	ground truth	unclassified
chromium	1381	331	89	6511
httpd	115	68	121	4188
linux	0	0	496	6937
gecko-dev	718	210	0	4646
FFmpeg	147	79	0	8101
openssl	70	38	0	5807
postgres	50	6	0	5836
wireshark	148	84	0	8217
tcpdump	85	34	0	3745
qemu	70	37	0	7540
Total	**2784**	**1383**	**706**	**61528**

Table 4.9.: Training set

5. Classifier

This chapter outlines how we use the finished dataset from the previous section to train a classifier of our own. We will go into how linear Support Vector Machines work and what we did to improve upon the ideas brought forward by Perl et al. [11]. Furthermore, we will compare our classifier to a mature static analysis tool called Flawfinder [3] as well as a replication of VCCFinder. Lastly, we will discuss CVEs our classifier would have detected as well as commits it flagged that were labeled as unclassified.

5.1. Linear Support Vector Machine

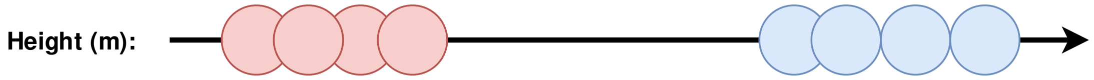
Just like VCCFinder, we choose a linear Support Vector Machine (SVM) as our method for classifying commits. This has three main reasons:

- SVMs can produce good results even if the training data is relatively small.
- SVMs are able to scale with very high dimensional feature vectors.
- SVMs can provide reasons for their classifications.

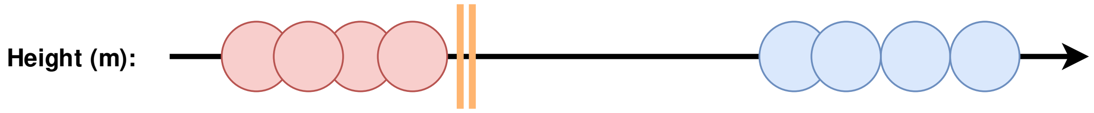
In order to understand why this is the case, we have to explain how exactly SVMs work.

5.1.1. Support Vector Classifiers in the First Dimension

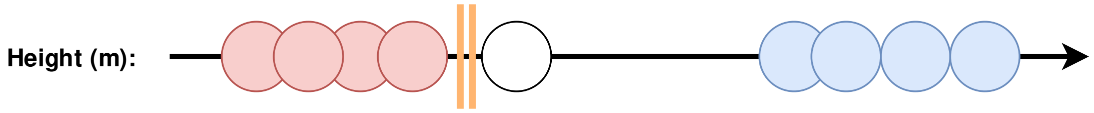
Assume we want to classify people into male and female solely based on their height. The image below shows possible training data for such a problem, with red dots representing females and blue dots representing males.



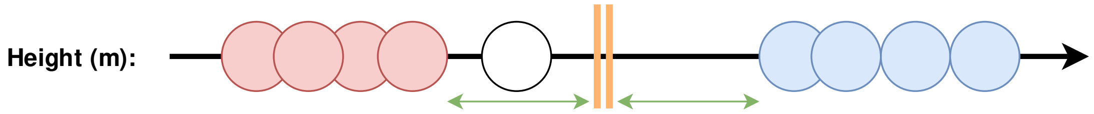
To decide whether an unlabeled data point (the gender of the person is unknown) should be classified as male or female, a threshold can be chosen. Every person with a height below the threshold is classified as female and every person with a height above the threshold is classified as male. The orange line shows a possible threshold that fits our training data.



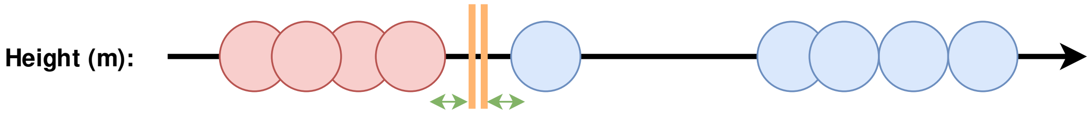
This, however, is a rather poor choice because an unlabeled data point with a height just above the threshold would be classified as male even though it is much closer to the female data points.



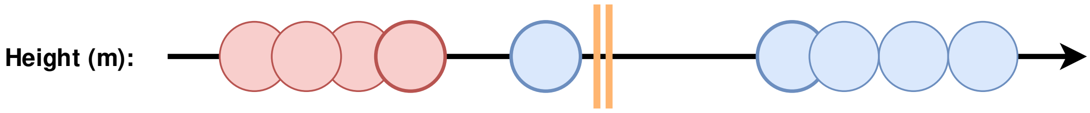
A better choice would be to set the threshold at the midpoint between the inner edges of each cluster. The distance between these edges and the threshold is called the *margin* (green), thus this is a *Maximal Margin Classifier*. Now, the unlabeled data point gets classified as female which is much more likely according to our training data.



This type of classifier is applicable in some scenarios but problems arise when the training data contains outliers. In the next example, a male with below average height in the training set causes the threshold with the maximal margin to be too close to the cluster of females again.

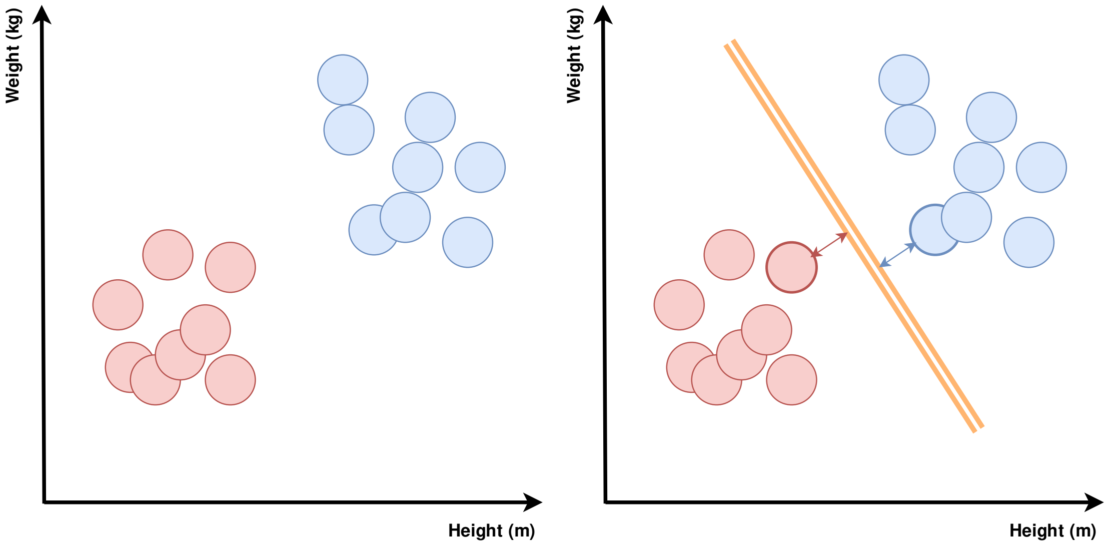


To fix this issue, the classifier has to be allowed to purposely miss-classify data points from the training data in order to achieve a better performance when classifying unknown data. This is called a *Soft Margin Classifier* or *Support Vector Classifier*. The two data points at the edges and every data point within the margin are called support vectors hence the name.



5.1.2. Support Vector Classifiers in Higher Dimensions

Assume that in addition to our height data, we also knew the weight of each person in the training set. This data can be visualized in a two dimensional coordinate system.



Instead of an explicit threshold value, the Support Vector Classifier would now form a one dimensional straight line and base its decisions on the distance of a new data point to that line. For three dimensional data, the Support Vector Classifier would form a two-dimensional plane and for n-dimensional data the Support Vector Classifier would form an $n-1$-dimensional hyperplane. The hyperplane can be written as a set of points $\vec{x}$ satisfying

$$\vec{w} \cdot \vec{x} - b = 0$$

where $\vec{w}$ is the normal vector to the hyperplane, $\frac{b}{||\vec{w}||}$ is the offset of the hyperplane from the origin along the vector $\vec{w}$ and the outer support vectors satisfy $\vec{w} \cdot \vec{x} - b = \pm 1$ depending on which class they are in.

For our prediction model, we are using a *linear* Support Vector Machine. A linear SVM is a type of Support Vector Classifier that assumes that the data it deals with is linearly separable. Thanks to this assumption, they are able to process very high dimensional feature vectors in a moderate amount of time. In addition to that, they can also provide a humanly understandable reason for why they make a decision. This is because one can easily see which feature had the biggest impact in a decision by checking which one has the greatest distance to the hyperplane.

5.2. Features

Our SVM classifier will make decisions on whether or not a commit is prone to to introduce a vulnerability based on a high dimensional feature vector representing the commit. To calculate these feature vectors we have to extract specific raw data for each commit. Table 5.1 shows this raw data along with descriptions for each data point and what *git* command was used to extract it. New data points are highlighted in bold, if a data point is marked with an asterisk it has also been used for VulDigger [17], if it is marked with a tilde it has been used for VCCFinder.

Raw data	Description	Extraction[1]
past different authors*∼	Number of authors that changed the same files as the commit in the past	git rev-list
file count*	Number of different files the commit changed	git diff
past changes*∼	Number of times the files in the commit were changed in the past	git rev-list
hunk count[2]∼	Number of hunks in the commit	git diff
commit message∼	The commit message of the commit	git show
added code*∼	Lines of code that were added by the commit	git show
deleted code*∼	Lines of code that were deleted by the commit	git show
author contribution percentage	Percentage of commits to the repository by the author	git rev-list
authored day	Day on which the commit was authored	git show
authored hour	Hour of the day on which the commit was authored	git show
author touch count	Number of times the author worked on the same files in the past	git rev-list

[1] Find the exact commands for extraction in the GitHub repository
[2] A hunk is a continuous block of changes in a commit

Table 5.1.: Table of raw data used to create the feature vectors.

The next subsections will outline how we use the raw data from each commit to create the high dimensional feature vector that represents it.

5.2.1. Metadata Features

The key idea brought forward in [11] is that combining *git* metadata with code data can be beneficial in identifying commits that are prone to introduce a vulnerability. This is why metadata features are a vital element of our classifier. Some of the more interesting features include:

Feature: Average past different authors
Calculation: $\frac{\text{past different authors}}{\text{file count}}$
Rationale: A high value of this feature may indicate that the experience of working on the files changed by the commit is distributed across many developers. It has been shown that this is correlated with faults and vulnerabilities [7].

Feature: Author contribution percentage
Calculation: *raw data point*
Rationale: New contributors to the repository are more likely to make mistakes [9].

Feature: Author experience
Calculation: $\frac{\text{author touch count}}{\text{past changes}}$
Rationale: If the authors experience with the files in the commit is low, the probability that she/he introduces a vulnerability may be higher [7].

Feature: Authored hour
Calculation: *raw data point*
Rationale: If a commit was authored in a 4 am grind compared to more usual working hours, the developer might have been unconcentrated and thus could have made a mistake.

Find a full list of the extracted metadata features in A.2.

5.2.2. Code Metric Features

We also want to consider how often specific C/C++ keywords are being used in the commit. The rationale behind this is that Perl et al. showed that at least some of the keywords are differently distributed across VCCs and unclassified commits [**perl2025vccfinder**]. To access this data we split *added code* and *deleted code* at the C/C++ delimiters (\s + - * / ; > < = ( ) [ ] { }) and simply count the occurrences of each keyword. In addition to considering the *total* occurrences of each keyword, we also add the *average* use per line of code to our feature vector. Lastly we keep track of how many functions are inserted/removed. To measure this we use following regular expression to find function declarations in the code:

```
\s*(?:(?:inline|static)\s+){0,2}\w+\s+\*?\s*\w+\s*\([^!@#$+%^;]+?\)\s*\{
```

5.2.3. Code/Commit Message Features

For code- and commit message features we utilize a bag-of-words model. Bag-of-words come from the area of natural language processing and are a method of representing text while disregarding grammar and word order. Our bag-of-word consists of a set of tokens S, where each token represents the occurrence of a specific word in a text.
For tokenization of the code, we again use the C/C++ delimiters to determine where a word stops and for the tokenization of the commit message we use the stop words of the English language. We can add this bag-of-words to the feature vector of a commit c by appending $b(c, s)\ \forall s \in S$, with

$$b(c,s) = \begin{cases} 1 & if\ token\ s\ is\ in\ c \\ 0 & else. \end{cases}$$

to it. This results in a very large, although sparse, feature vector, but like discussed in the previous section, this high dimensionality does not pose a problem for our linear SVM.
Except for arguably the tokenization, this is also how VCCFinder processes code/commit message data. We want to note that we also tried using three separate bags-of-words for added code, deleted code and the commit message respectively, but did not see an increase in performance with this approach. The same is true for removing comments before tokenizing the code[1], the rationale being that comments do not change code

[1] This was done utilizing the GNU Compiler Collection: $gcc -fpreprocessed -dD -E code.c

functionality and thus should maybe not be considered. We suspect that while comments have no influence on the functionality, they still can give the classifier, just like they give humans, insight into what the code is supposed to do.
Another failed attempt of improving the bag-of-words used for VCCFinder involves adjusting the appended values for term frequency times inverse text-frequency instead of just adding a binary flag that indicates if a token appeared. Reasons for why this lowers the performance will be discussed in the next section.

5.3. Pre-processing

The pre-processing of data is one of the most important steps in any machine learning project. It refers to the act of raising the quality of training data by minimizing the amount of noise, as well as ensuring that all the different features are treated equally. For the implementation of our pre-processing as well as the rest of the classifier, we utilize the scikit-learn library for Python [10].
We already applied a kind of pre-processing when we sorted the VCCs available to us into *confident*, *not confident* and *ground truth*, thus labeling which data we should give more weight to when training the classifier. Regarding treating every feature equally, we tried two different methods where one significantly outperformed the other. We still want to highlight the former method as it initially seemed like the better option and understanding why it is not was an interesting learning experience during our research.

5.3.1. Maximum Absolute Scaler

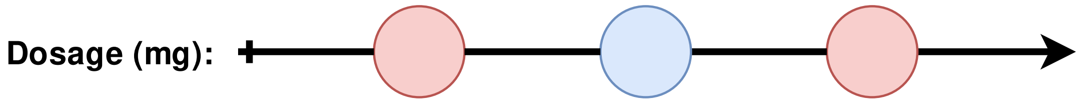
The first pre-processing method we tried is called a Maximum Absolute Scaler. Its main purpose is to scale each feature of every feature vector to the same range of possible values (usually from -1 to 1). This is important because otherwise, features where the range of possible values is larger than others (e.g. *added lines* 0-2000, *author contribution percentage*: 0-1), have more of an impact on the distance from the feature vector to the hyperplane.
The reason we thought this method was superior is that it keeps the nuances of the continuous features, which we thought would result in more accurate predictions. The method discussed in the next section loses some of these nuances but has a different key advantage.

5.3.2. K-bins Discretization

The method of pre-processing we end up using is called K-bins Discretization. With K-bins Discretization every value of the feature vector becomes binary. But Instead of allocating a single dimension for each feature, features get sorted into one of multiple bins. Each bin holds a range of possible values a feature might have and is represented by its own dimension in the feature vector. For example before pre-processing, the feature vector for a commit that added 13 lines of code would hold the value 13 at the dimension allocated for *number of added lines*. After pre-processing, we would have several new zero dimensions representing bins that the value 13 did not fall into and one non-zero dimension representing the bin the value did fall into:

$$\begin{pmatrix} \dots \\ \dots \\ 13 \\ \dots \\ \dots \end{pmatrix} \begin{matrix} \dots \\ \dots \\ \text{number of added lines} \\ \dots \\ \dots \end{matrix} \longrightarrow \begin{pmatrix} \dots \\ 0 \\ 1 \\ 0 \\ \dots \end{pmatrix} \begin{matrix} \dots \\ \text{0-10 added lines} \\ \text{10-20 added lines} \\ \text{20-50 added lines} \\ \dots \end{matrix}$$

This method was also used for VCCFinder. At first we thought the Maximum Absolute Scaler would be a better way of pre-processing because it describes the commit in a more precise way. However, K-bins Discretization brings a very useful property to our model that the Maximum Absolute Scaler does not. By transforming the continuous features into nominal ones, we can introduce non linearity to our linear model. To explain how that works we can again look at an example in lower dimensions:
Assume we want to classify drug dosages. For a dosage to be effective it can't be too little (underdose) or too high (overdose) but has to be just right. The following one dimensional graph shows three different dosages (3 mg, 5 mg, 8 mg) where only the 5 mg dosage (blue) is effective:



Unlike the example in 5.1.1, this problem is not linearly separable because we cannot find a threshold that separates the two groups without at least one miss-classification. However, by using K-bins Discretization we can. For that we create three bins: One for dosages from 0-3 mg, one for 3-6 mg and one for 7-∞ mg. Now instead of three values we have three 3 dimensional vectors:

$$\begin{pmatrix}1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\end{pmatrix}$$

When plotting these three vectors, we can see that now we are able to correctly separate the classes with a linear plane (5.1).

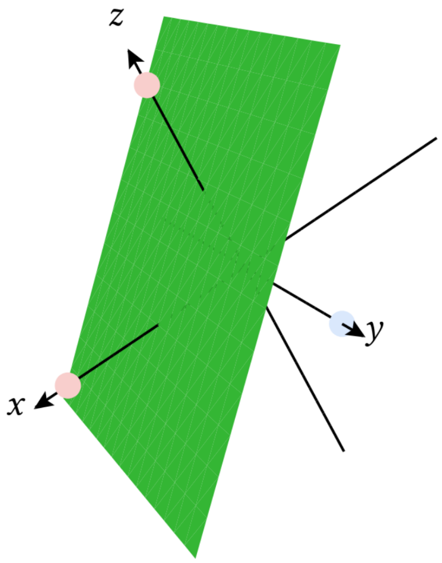


Figure 5.1.: Graph of the feature vectors and hyperplane after K-bins Discretization.

This is also the reason why we decided against adjusting the values in the bag of words for frequency times inverse text-frequency and instead just have one bucket for each word that stores whether the specific word occurs (1) or not (0). To decide the widths of bins for the features that are not part of the bag-of-words, we use what is called a *quantile* strategy. This means that we choose the width of each bucket so that every bucket holds roughly the same amount of commits from our training set.

5.4. Improvements over VCCFinder

One of the main goals of this thesis is to improve upon the design by Perl et al. In this section we want to highlight the main differences between our classifier and VCCFinder as well as discuss why they provide an increase in performance or other reasons why we consider them improvements.

Confidence weighting: In chapter 5 we showed that the heuristic used by Perl et al. as well as our heuristic often incorrectly flags commits as VCC. By assigning a confidence measurement to each VCC we are able to identify a subset that should be over 60% more accurate than the whole set. This helps us to minimize the noise in our training set by weighting *ground truth* and VCCs we are *confident* in heavier than VCCs we are *not confident* in. Giving higher weight to specific VCCs while training, ensures that the classifier is less likely to miss-classify these VCCs when calculating its hyperplane. This has been shown to raise the performance level of SVMs [18].

Features: We found several additional features that can be beneficial in distinguishing VCCs from non-VCCs:

- File count - VCCFinder does not track how many files a commit alters. A Mann-Whitney U test shows that the distribution of altered files indeed differs significantly from the VCCs and the unclassified commits in our training set ($\rho = 4.39472593956 * 10^{-285}$, effect size = 37% [15]).
- Author contribution percentage - While VCCFinder has a feature with the same name, there it only refers to the *overall* contributions of an author, while we look at the contributions *at the time of committing*. Our issue with their feature is that an author can have a high *overall* contribution percentage even if the commit in question is the first one she/he ever contributed to the repository. Since this feature is supposed to measure the authors experience with the repository, we argue that our version is superior.
- Author touch count - VCCFinder does not track how many times an author worked on the files changed in the commit before. We think that lower experience with the files can indicate that the author is more likely to make mistakes and thus think this is a sensible feature to include.
- Authored day and authored hour - VCCFinder utilizes time features but only measures them by their unix time. As far as we know they do not consider at what time of the

day a commit was authored. We think that the time of day especially but also the day of the week, can correlate with the level of concentration a developer has when authoring a commit and therefore with the likeliness of her/him making a mistake.

- Averages - VCCFinder only considers the total amount of *added lines, past different authors, used keywords*, etc. but never the averages (per file/inserted lines/deleted lines). We think that the averages can give additional insight into the nature of a commit.
- VCCFinder does not distinguish between *added*- and *deleted keywords*, which we think should be separated.

Tokenization: In the VCCFinder paper it is clearly stated that they use "spaces and newlines" for the tokenization of the code and the commit messages. We think that this is a poor way of generating decisive tokens as shown by the example below and thus opt for using the C/C++ delimiters as well as spaces and newlines for our tokenization.

Example line of code:

```
lsb = i2c_smbus_read_byte_data(client, ISL29003_REG_LSB_SENSOR);
```

VCCFinder tokens:
'lsb', '=', 'i2c_smbus_read_byte_data(client,', 'ISL29003_REG_LSB_SENSOR);'

Our tokens:
'lsb', 'i2c_smbus_read_byte_data', 'client', 'ISL29003_REG_LSB_SENSOR'

Independence from GitHub: VCCFinder uses metadata features extracted from *GitHub* for its feature vectors. We choose to only rely on *git* metadata features and so our tool is applicable to more software repositories.

5.5. Evaluation and Comparison

This section discusses how our classifier performs on the test set we created. Further, we will compare the results to Flawfinder as well as our replication of VCCFinder.

5.5.1. Comparison to Flawfinder

Flawfinder (version 2.0.11) is a mature, open source static analysis tool for C and C++ code. It comes with an integrated database of functions that are known to make code prone to be vulnerable. To detect vulnerabilities or "flaws", the tool tries to match the source code against these functions. For example, the use of *strcpy()* may indicate a risk of a buffer overflow vulnerability while *chmod()* is often associated with race conditions. It can detect strings and comments too and thus is able to ignore them. Since Flawfinder is also able to operate on commits, it makes sense to compare its performance to the one of our classifier.

To review a commit with Flawfinder, one has to create a *unified diff* for each file the commit changed. For this, the *git diff* command can be used to compare the commits version of a file to its parents version. The generated diff is given to Flawfinder using the -P flag. Flawfinder can then analyze the files but only reports hits that relate to lines that were changed in the commit. Note that these lines do not necessarily have to be part of the commit. To automate this process we wrote this bash script:

Listing 5.1: Bash script to use Flawfinder on commits.

```
#!/bin/bash

# the first given argument is the hash of a commit
commit_hash=$1

# iterate over each file that was changed in the commit
for path in $(git diff --name-only $commit_hash~ $commit_hash)
do
   # create an empty file at that path
   install -D /dev/null $path
   # write the commits version of the file into the path
   git show $commit_hash:$path > $path

   # create the diff of the commits version of the file and
   # the parents version
   git diff --output=recent.patch $commit_hash~:$path
       $commit_hash:$path

   # use flawfinder but only report hits that relate to lines
   # changed in the patch
   # if the patch does not exist (the file was newly created
   # in the commit) use flawfinder on the file directly
```

```
    flawfinder -P recent.patch $path || flawfinder $path
done
```

Table 5.2 shows the results of running the script over every commit in our test set. Flawfinder also has an option to reduce false positives that we can enable by using the -F flag. As the table shows, it successfully reduces the reported false positives by 22.8% but also misses out on four of the true positives.

5.5.2. Comparison to VCCFinder

VCCFinder is the first static analysis tool that used commit metadata in combination with code metric data to detect if a commit is prone to introduce a vulnerability. Since our classifier is closely based on it and supposed to improve upon the design contributed by Perl et al., we want to compare the two in this section. In order to do this, we first have to replicate VCCFinder to the best of our abilities, since its GitHub repository is unfortunately incomplete.

Differences between VCCFinder and Our Replication

We know that our replication is not a 100 percent accurate representation of what was used by Perl et al. The differences/uncertainties we are aware of are:

- VCCFinder uses several GitHub repository features (*number of commits, number of contributors, star count, fork count, programming language*). These features are the same for every commit from the same repository. Since we worried that this could bias our classifier and wanted to stay independent from GitHub, we did not include them in our data gathering process and thus cannot use them for our replication of VCCFinder.
- We are unable to recreate VCCFinders *modified functions* feature because we are not sure what exactly it measures.
- For their *contribution percentage* feature, Perl et al. look at the *overall* contributions of an author to a repository. We only have the contributions of an author *at the time of committing* available to us but we argue that this is a better feature (5.4).

- The VCCFinder paper only glances over how their K-bins Discretization pre-processing is configured. We do not know how many different bins they used for each feature as well as the bin widths. Regarding the bin widths, we think that a quantile strategy like we use for our classifier should be very close to what is described in their paper:

  > "If the numerical values are rather evenly distributed, we apply a uniform grid, whereas for features with skewed distribution we apply a logarithmic partitioning"

- The VCCFinder paper has a different list of C/C++ keywords that are represented in the feature vectors than the VCCFinder repository. The list of keywords in the paper is much shorter. We choose to use the longer list from the repository for our replication.

Despite the differences, we think that our replication should yield comparable results.

5.6. Results

Comparing our classifier to Flawfinder and our replication of VCCFinder, as discussed in the previous sections, we find improvements over both.

5.6.1. Flawfinder

For the sake of comparison, we adjust the confidence threshold of the classifiers so that they match Flawfinders precision, recall and false positives rate respectively. For this, we set the level of confidence the classifiers have to reach in order to flag a commit as VCC accordingly. Table 5.2 shows that our replication of VCCFinder is able to reduce Flawfinders false positive rate by 72.63% whereas our classifier reduces it by 83.22% with the same amount of true positives. While this is still a very significant decrease, it is a step down from the 99% reduction achieved with VCCFinder reported by Perl et al. in their paper. Reasons for this might be:

- Assuming Perl et al. used the latest version of Flawfinder at the time of the papers release, they did their comparison on version 1.31. We used Flawfinder version 2.0.11 for our comparison which might have become better at flagging vulnerable commits.

- According to their paper, Perl et al. only consider a commit flagged by Flawfinder if the tool reports a hit for a line that was actually changed in the commit. Instead, we follow the instructions[1] on how to review a patch with Flawfinder given by its creator. This also flags a commit if it *relates* to a hit.
- We showed that there is likely a non-negligible amount of falsely labeled VCCs in the test set by Perl et al. (4.1.1), which might have biased the comparison.
- Against our expectations, the differences we described in 5.5.2 might be sufficient to significantly impact VCCFinders performance.

The reduction of false positives is still large enough to drastically lower the amount of code audits that would have to be conducted to find the same amount of VCCs with Flawfinder. Thus, we consider the classifiers a viable alternative to it.

[1] `https://dwheeler.com/flawfinder/#reviewing_patches`
this might have not been written at the time of VCCFinders release

	True Positives	False Positives	False Negatives	True Negatives	Precision	Recall
Flawfinder						
regular	35	9138	94	118436	0.0038	0.2713
with -F	31	7054	98	120520	0.0044	0.2403
VCCFinder						
same	35	2501	94	125073	0.014	0.2713
recall	31	2010	98	125564	0.015	0.2403
same false	62	9138	67	118436	0.0067	0.4806
positives	58	7054	71	120520	0.0082	0.4496
same	101	26137	28	101437	0.0038	0.7829
precision	93	20600	36	106974	0.0044	0.7209
Our model						
same	35	1533	94	126041	0.0223	0.2713
recall	31	1217	98	126357	0.025	0.2403
same false	79	9138	50	118436	0.0086	0.6124
positives	71	7054	58	120520	0.01	0.5504
same	99	25663	30	101911	0.0038	0.767
precision	98	22018	31	105556	0.0044	0.7597

Table 5.2.: Comparison of the classifiers with Flawfinder.

5.6.2. VCCFinder

We further compare our classifier to the replication of VCCFinder by examining their precision and recall graphs. Figure 5.2 shows that our classifier has a closer point to the optimum (upper right corner) as well as a bigger area under the curve than VCCFinder (VCCFinder: 0.011, ours: 0.018). This indicates that our classifier is an overall improvement over VCCFinder.

We want to note that the difference in performance marginally decreases if we also use our improved way of tokenizing the text data points for VCCFinder, although our classifier still performs slightly better overall. We have to take the description of the tokenization (5.4) given by Perl et al. at face value since they could not provide us with the code for their bag-of-words model. However, we suspect that they used a tokenization similar to ours in order to achieve their great results but failed to explain it properly in the paper.

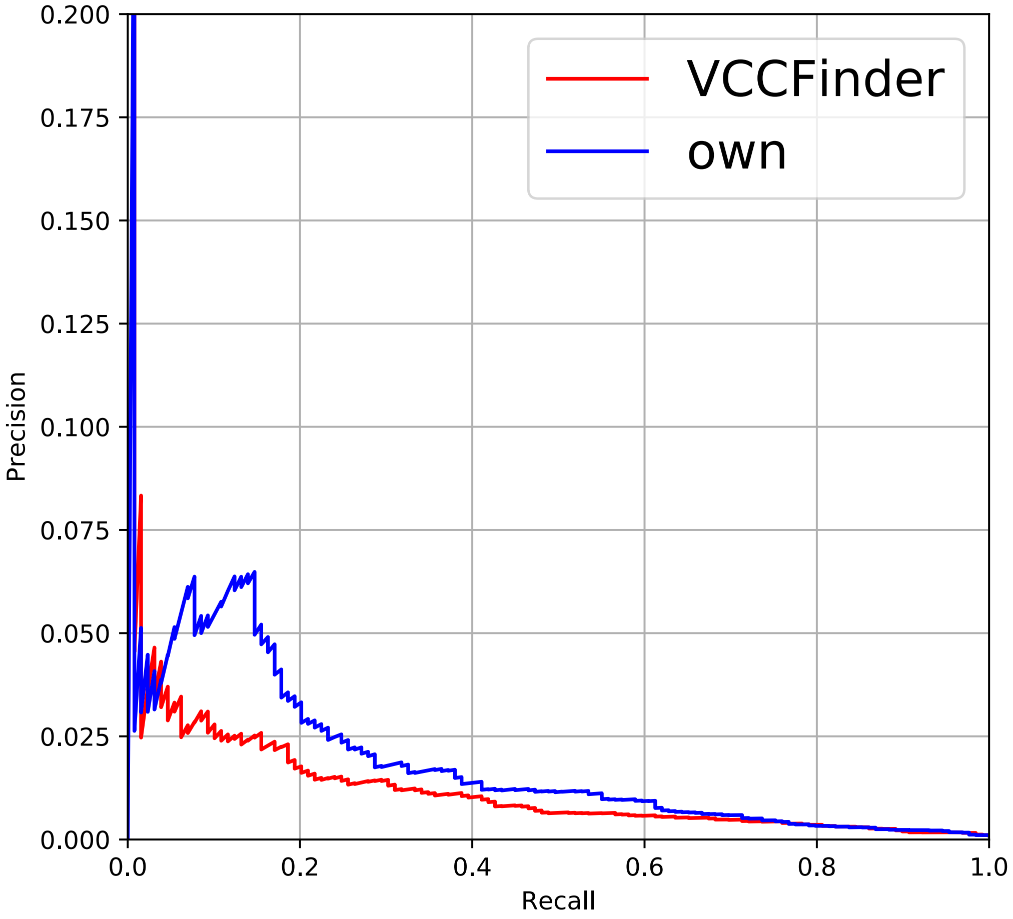


Figure 5.2.: Precision and recall graph of VCCFinder and our classifier.

Moreover, we want to address an issue regarding the training process. When choosing the hyperparameter $c = 1$, as done by Perl et al., when training our replication of VCCFinder, we notice severe overfitting. The c parameter adjusts the *softness* of the margin: The lower c, the more the classifier is willing to miss-classify data points in the training set in order to find a more general hyperplane. Choosing $c = 1$ leads to almost perfect performance on the training set, due to the SVM refusing to mislabel outliers. This is possible because the number of features is much larger than the number of samples in our training set which was also true for the training set by Perl et al. This performance can not be upheld on the test set, meaning the classifier overfit. Instead, we opt for a softer margin with $c = 0.191$ as determined by a standard 10-fold Cross-Validation process. With this value for c the performance on training- and test set becomes more similar.

5.6.3. Limitations

This evaluation has two main limitations regarding the test data that we want to address. As previously mentioned, we prioritize evaluating how the classifier would have fared in a real life developing environment. This has the disadvantage that the amount of VCCs in our test set is too small to make well grounded statistical claims about its performance. In addition, we would need more ground truth data from different software projects than the Linux kernel to demonstrate that our classifier can be used on a general basis. At the time of writing, we do not have enough ground truth data available to us but plan on conducting further evaluation in future work. We would like to also highlight that we do not know how many unkown/not CVE worthy vulnerabilities are contained in our test set. Thus, our true positives must be considered a lower bound and the false positives an upper bound. So the results for both classifiers as well as Flawfinder might be slightly better than reported.

5.7. Case Study

In this section we want to further answer the question of how our classifier would have fared if it was actively used in the developing of the Linux kernel in 2016 and 2017. For this we want to discuss some example VCCs that our classifier flags if we match Flawfinders number of false positives in detail.

CVE-2017-15126
Commit: *893e26e61d*
Confidence: 1.02
Fix: *384632e67e*
Notes: This commit introduced a use-after-free vulnerability in fs/userfaultfd.c. The issue is related to the handling of fork failure when dealing with event messages. Failure to fork correctly can lead to a situation where a fork event will be removed from an already freed list of events with userfaultfd_ctx_put().
The main reason for why this commit was flagged is the occurrence of the word "ptr_err" in the code. While the use of "ptr_error" has not much to do with the actual use-after-free vulnerability, it is interesting that the classifier links this word to VCCs. The metadata feature that contributed the most to the feature vector of this commit being further away from the hyperplane is the large amount of code the commit added. This is very common for the commits that get flagged by the classifier and underscores the fact that VCCs are usually rather large [9].

CVE-2018-10853
Commit: *129a72a0d3*
Confidence: 0.009
Fix: *3c9fa24ca7*
Notes: This is a very interesting commit which the classifier almost missed, most likely due to the low number of added lines. The commit introduced a flaw in the way the Kernel Virtual Machine hypervisor emulates specific instructions. The flaw being that it did not check the current privilege level while emulating unprivileged instructions. Thus, an unprivileged guest user or process could use this to potentially escalate privileges.
Interestingly, the top feature for why the commit was flagged is the occurrence of "CVE" in the commit message. The word occurs because this VCC is also a fixing commit for CVE-2017-2584. It has been shown that fixing commits are more likely to introduce a vulnerability themselves [5], which our classifier seemingly has picked up on. Other top features include the occurrence of "kvm" which is promising considering the nature of the vulnerability and the low contribution percentage of the author as the top metadata feature.

CVE-2018-7740
Commit: *ff8c0c53c4*
Confidence: 0.23
Fix: *045c7a3f53*
Notes: This is one of two commits that enabled malicious users to cause a denial of service by crafting applications that make mmap (memory map) system calls and have a large

page offset argument.
The top reason the classifier flagged this commit is the occurrence of the word "bug". The commit message contains the word "bug" because this commit fixed a bug that has been detected by a different vulnerability detection tool, the syzkaller fuzzer. This again suggests that the SVM learned that fixing commits are at higher risk of introducing vulnerabilities themselves. The top metadata feature again is the authors contribution percentage.

5.8. Flagged Unclassified Commits

Even though we reduced the false positive rate by Flawfinder significantly, running our classifier over the test set still produces a non-negligible amount of commits that are flagged as VCC but were labeled as unclassified. While they count as false positives in the evaluation, there is a high possibility that these commits are also prone to introduce a vulnerability but were not identified as VCC. We choose two different of these cases where the confidence of the classifier was particularly high to discuss in detail.

Commit: *989782dcdc*
Confidence: 8.497
Fix: *a5fb8e6c02*, *6b3944e42e*
Notes: This was by far the flagged unclassified commit with the highest confidence. This commit introduces a leak on the cell refcount in *afs_lookup_cell_rcu()* due to non-clearance of the default error in the case a NULL cell name is passed. The bug was also later detected by the syzkaller fuzzer[1] but not assigned a CVE.
The top reasons for why the classifier flagged this commit are the occurrences of "initialise" and "bug" in the code as well as the large number of added lines of code and the low contribution percentage by the author.

Commit: *8db6c34f1d*
Confidence: 3.639
Fix: *355139a8db, 1f5781725d, dc32b5c3e6, 76ba89c76f, 6b240306ee*
Notes: This commit added various changes that later had to be fixed. The most prominent one being that the commit calls the function *d_find_alias()* instead of *d_find_any_alias()* and thus does not handle unhashed *dentry* objects correctly. This bug was reported and assigned an ID[2].

[1] `https://syzkaller.appspot.com/text?tag=CrashReport&x=11043d76e00000`
[2] `https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1736808`

The top reason for why this commit was flagged is surprisingly the occurrence of the word "100000" in the commit message. In this commit it refers to the value of the first userid as seen on the host when using *lxc-usernsexec* to run *chown* as root user in a new user namespace. The use of the word "security" in the commit message is also among the top reasons for why the commit was blamed. The top metadata features again are the high number of added lines of code and the low author contribution percentage.

6. Conclusion

In this thesis, we showed that the heuristic for mapping a fixing commit to its Vulnerability-Contributing Commit (VCC) used in previous works is not as accurate as was previously assumed. Furthermore, we introduced a heuristic of our own that shows slightly better results with the added benefit of a confidence measurement that indicates when the commit blamed by the heuristic is more likely to be correct. Combining these automatically traced VCCs with VCCs that were manually found by security researches, we created a large scale dataset that we make available to the community. Finally, we used this dataset to train a classifier that can to a degree successfully distinguish between commits that are prone to introduce a vulnerability and commits that are not. When comparing our classifier to other commit-based static analysis tools, we evaluated that it greatly outperforms the mature open-source tool Flawfinder and has slightly better performance than the classifier it was closely based on.

In addition to what was addressed in 5.6.3, we see two main avenues for future research. Creating a more proficient way of mapping fixing commits to their VCC still remains an interesting problem. While we do not think that a simple blame based algorithm like discussed in this thesis can be much further optimized, we are curious if a machine learning approach could yield better results.

The second idea for future work is combining the information retrieved from *git* metadata, which [11, 17, 7] and we proven is very valuable, with a more expressive intermediate representation of code than a bag-of-words. In addition to potentially giving better insight into how the code functions, this could help to further pinpoint the flaw of a commit. This would be helpful, since VCCs are usually rather large which makes code audits time consuming. Two possible ways that we suggest are code gadgets [6] and code property graphs [16].

Bibliography

[1] Manuel Brack. *A large-scale statistical Analysis of Vulnerability Liftetimes in Open-Source Software*. `https://fileserver.tk.informatik.tu-darmstadt.de/NA/Thesis_MB.pdf`.

[2] *cvedetails.com: Vulnerability list of the Linux kernel*. `https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/Linux-Linux-Kernel.html`. Accessed: 2020-01-13.

[3] *D. A. Wheeler. Flawfinder*. `https://dwheeler.com/flawfinder/`. Accessed: 2020-03-18.

[4] *Distribution of vulnerabilities by type*. `https://www.cvedetails.com/vulnerabilities-by-types.php`. Accessed: 2020-05-20.

[5] Yasutaka Kamei et al. "A large-scale empirical study of just-in-time quality assurance". In: *IEEE Transactions on Software Engineering* 39.6 (2012), pp. 757–773.

[6] Zhen Li et al. "Vuldeepecker: A deep learning-based system for vulnerability detection". In: *arXiv preprint arXiv:1801.01681* (2018).

[7] Andrew Meneely and Oluyinka Williams. "Interactive churn metrics: socio-technical variants of code churn". In: *ACM SIGSOFT Software Engineering Notes* 37.6 (2012), pp. 1–6.

[8] Andrew Meneely et al. "An empirical investigation of socio-technical code review metrics and security vulnerabilities". In: *Proceedings of the 6th International Workshop on Social Software Engineering*. 2014, pp. 37–44.

[9] Andrew Meneely et al. "When a patch goes bad: Exploring the properties of vulnerability-contributing commits". In: *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE. 2013, pp. 65–74.

[10] Fabian Pedregosa et al. "Scikit-learn: Machine learning in Python". In: *the Journal of machine Learning research* 12 (2011), pp. 2825–2830.

[11] Henning Perl et al. “Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits”. In:

[12] Fraser Price. “Analyzing Vulnerabilities in the Chromium Browser”. In: ().

[13] Riccardo Scandariato et al. “Predicting vulnerable software components via text mining”. In: *IEEE Transactions on Software Engineering* 40.10 (2014), pp. 993–1006.

[14] *Ubuntu CVE Tracker*. `https://git.launchpad.net/ubuntu-cve-tracker`. Accessed: 2020-03-18.

[15] Hans W Wendt. “Dealing with a common problem in Social science: A simplified rank-biserial coefficient of correlation based on the U statistic”. In: *European Journal of Social Psychology* 2.4 (1972), pp. 463–465.

[16] Fabian Yamaguchi et al. “Modeling and discovering vulnerabilities with code property graphs”. In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 590–604.

[17] Limin Yang, Xiangxue Li, and Yu Yu. “VulDigger: A Just-in-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes”. In: *GLOBECOM 2017-2017 IEEE Global Communications Conference*. IEEE. 2017, pp. 1–7.

[18] Xulei Yang, Qing Song, and Yue Wang. “A weighted support vector machine for data classification”. In: *International Journal of Pattern Recognition and Artificial Intelligence* 21.05 (2007), pp. 961–976.

[19] Yaqin Zhou and Asankhaya Sharma. “Automated identification of security issues from commit messages and bug reports”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017, pp. 914–919.

A. Appendix

A.1. Appendix: Report by Nikolaos Alexopoulos

A.1.1. Introduction

During their research, Jan Wagner and Manuel Brack noticed some discrepancies between Vulnerability-Contributing-Commits (VCCs) as reported by the VCCFinder heuristic and the Ubuntu Kernel Security Team. Ten of those disputed CVEs were randomly picked for further manual analysis. The picked CVEs, along with their fixing and reported VCCs, are listed in the following table.

CVE	Fix	Ubuntu Security Team VCCFinder
CVE-2014-5206	a6138db815df5ee542d848318e5dae681590fccd	0c55cfc4166d9a0f38de779bd4d75a90afbe7734 495d6c9c6595ec7b37910dfd42634839431d21fd
CVE-2013-4127	dd7633ecd553a5e304d349aa6f8eb8a0417098c5	1280c27f8e29acf4af2da914e80ec27c3dbd5c01 2839400f8fe28ce216eeeba3fb97bdf90977f7ad
CVE-2013-1819	eb178619f930fa2ba2348de332a1ff1c66a31424	74f75a0cb7033918eb0fa4a50df25091ac75c16e 3e85c868a697805a3d4c7800a6bacdfc81d15cdf
CVE-2013-7348	d558023207e008a4476a3b7bb8706b2a2bf5d84f	e34ecee2ae791df674dfb466ce40692ca6218e43 e23754f880f10124f0a2848f9d17e361a295378e
CVE-2014-5045	295dc39d941dc2ae53d5c170365af4c9d5c16212	8033426e6bdb2690d302872ac1e1fadaec1a5581 35759521eedf60ce7d3127c5d33953cd2d1bd35f
CVE-2012-1097	c8e252586f8d5de906385d8cf6385fee289a825e	bdf88217b70dbb18c4ee27a6c497286e040a6705 5bde4d181793be84351bc21c256d8c71cfcd313a
CVE-2014-0196	4291086b1f081b869c6d79e5b7441633dc3ace00	d945cb9cce20ac7143c2de8d88b187f62db99bdc d6afe27bfff30fbec2cca6ad5626c22f4094d770
CVE-2013-1979	83f1b4ba917db5dc5a061a44b3403ddb6e783494	257b5358b32f17e0603b6ff57b13610b0e02348f dbe9a4173ea53b72b2c35d19f676a85b69f1c9fe
CVE-2011-1479	d0de4dc584ec6aa3b26fffea320a8457827768fc	a2ae4cc9a16e211c8a128ba10d22a85431f093ab 2d9048e201bfb67ba21f05e647b1286b8a4a5667
CVE-2013-4470	c547dbf55d5f8cf615ccc0e7265e98db27d3fb8b	e89e9cf539a28df7d0eb1d0a545368e9920b34ac c31d5326902cebffcd83b1aede67a0e0ac923090

A.1.2. Analysis

Results of the manual analysis follow.

CVE-2014-5206

Date: 08/18/2014
Type: CWE-264 – Permissions, Privileges, and Access Controls
Description: The do_remount function in fs/namespace.c in the Linux kernel through 3.16.1 does not maintain the MNT_LOCK_READONLY bit across a remount of a bind mount, which allows local users to bypass an intended read-only restriction and defeat certain sandbox protection mechanisms via a “mount -o remount” command within a user namespace.
Notes: VCCFinder blames a commit that cleans up code. It changes the expression for the mount flag from a literal constant to a bitwise operation on constants defined in the header file. No functionality difference should exist.
Reason for discrepancy: This is an interesting one. It seems that the vulnerability is introduced in the commit blamed by the Ubuntu security team that allows non-root users to perform operations on namespaces. Before then, there was no reason to protect the flags. Specifically in the SYSCALL definition of umount.
Verdict: The commit blamed by the Ubuntu teams seems to be the VCC. It seems very difficult for a blame-based heuristic to pinpoint this commit. The VCC pinpointed by VCCFinder is definitely wrong though, as it is just refactoring. Namespaces are a very high-risk location for those kind of vulnerabilities. This VCC is also responsible for the related CVE-2014-5207. Interestingly, these vulnerabilities did not affect the Debian stable distribution, where by default user namespaces are disabled. Commits that change a lot of if statements that affect system calls should be considered high-risk, especially when they advertise introducing intrinsically high-risk functionalities. Maybe a good recommendation would be, when dealing with access control bugs to look more into changes to if statements. Another recommendation would be to consider such files (i.e. files that regulate access) as high-risk. Another observation is that such kinds of semantic flaws can only be automatically found when metadata, such as the commit messages are available.

CVE-2013-4127

Date: 07/29/2013
Type: CWE-399 – Resource Management Errors
Description: Use-after-free vulnerability in the vhost_net_set_backend function in drivers/vhost/net.c in the Linux kernel through 3.10.3 allows local users to cause a denial of service (OOPS and system crash) via vectors involving powering on a virtual machine.
Notes: This is a use-after-free vulnerability and thus should be able to be detected automatically pretty easily. The VCC proposed by VCCFinder is not correct. It seems to only be copying code from one file to another. The commit proposed by the Ubuntu security team seems to be the correct VCCs. This is also noted in the commit message of the fix.
Reason for discrepancy: It is impossible to trace code back to its original file after it has been copied to another file. That being said, in this specific case, the vulnerability was introduced in the file that the code was copied to.
Verdict: The VCC of the Ubuntu Security Team is correct. If a heuristic finds code copying, then maybe it would make sense to look into previous commits of the source file. Naturally, for a use-after-free vulnerability, we can expect the VCC to free memory.

CVE-2013-1819

Date: 03/06/2013
Type:CWE-20 – Improper Input Validation
Description: The _xfs_buf_find function in fs/xfs/xfs_buf.c in the Linux kernel before 3.7.6 does not validate block numbers, which allows local users to cause a denial of service (NULL pointer dereference and system crash) or possibly have unspecified other impact by leveraging the ability to mount an XFS filesystem containing a metadata inode with an invalid extent map.
Notes: The correct commit is also blamed by the VCCFinder heuristic but is not the most dominant.
Reason for discrepancy: Empty line contributes to the mistake. Also, blaming where the declaration of variables takes place could generally be bad practice.
Verdict: The Ubuntu Security Team seems to have the correct commit. VCCFinder produces a wrong commit, however we can learn from this mistake.

CVE-2013-7348

Date: 04/01/2014
Type: CWE-399 – Resource Management Errors
Description: Double free vulnerability in the ioctx_alloc function in fs/aio.c in the Linux kernel before 3.12.4 allows local users to cause a denial of service (system crash) or possibly have unspecified other impact via vectors involving an error condition in the aio_setup_ring function.
Notes: A double-free vulnerability can be easily detected by automated tools. We expect to blame the commit that introduces the 2nd free. In this case, it is more complicated than that. The vulnerability is introduced by the Ubuntu commit. This commit does not add a new free command. It changes the point where the error handling code "goes-to".
Reason for discrepancy: The fix was to remove one of the free instructions. The more general free instruction was removed, however the vulnerability was introduced in the more specific function. The phrase in the commit message: "clean up ioctx_alloc()'s error path" is what points to the problem.
Verdict: The Ubuntu commit is correct, however it was not trivial at all to see. Go-to in error handling seems to be dangerous. It is difficult to see how a blame-based heuristic would find the right commit.

CVE-2014-5045

Date: 08/01/2014
Type: CWE-59 – Improper Link Resolution Before File Access ('Link Following')
Description: The mountpoint_last function in fs/namei.c in the Linux kernel before 3.15.8 does not properly maintain a certain reference count during attempts to use the umount system call in conjunction with a symlink, which allows local users to cause a denial of service (memory consumption or use-after-free) or possibly have unspecified other impact via the umount program.
Notes: This is simple. The Ubuntu-blamed commit created the file and introduced the vulnerability.
Reason for discrepancy: The VCCFinder commit, as its message says, is massaging the code.
Verdict: The Ubuntu commit is the correct one. The VCCFinder commit just refactored the code in the area. Maybe going 2 steps back in time would have been beneficial. It does not seem very difficult to find this VCC automatically. The patterns existed before.

CVE-2012-1097

Date: 05/17/2012
Type: NVD-CWE-Other
Description: The regset (aka register set) feature in the Linux kernel before 3.2.10 does not properly handle the absence of .get and .set methods, which allows local users to cause a denial of service (NULL pointer dereference) or possibly have unspecified other impact via a (1) PTRACE_GETREGSET or (2) PTRACE_SETREGSET ptrace call.
Notes: This is a strange one. The two commits blamed by the Ubuntu team and VCCFinder were made the same day by the same person. The VCCFinder commit just adds some code to the first commit. This vulnerability in general has to do with error-checking. The reason behind it is that some developers did not follow what the expected behaviour of the functions were based on their header-file definitions. It is tough to blame a specific commit for such a vulnerability.
Reason for discrepancy: The VCCFinder commit adds some code to the earlier Ubuntu commit. Another possible commit to blame is 4206d3aa1978e44f58bfa4e1c9d8d35cbf19c187.
Verdict: This vulnerability appeared when function implementations started diverging from the definitions in the header files. All 3 possible blame-able commits could be valid.

CVE-2014-0196

Date: 05/07/2014
Type: CWE-362 – Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Description: The n_tty_write function in drivers/tty/n_tty.c in the Linux kernel through 3.14.3 does not properly manage tty driver access in the "LECHO & !OPOST" case, which allows local users to cause a denial of service (memory corruption and system crash) or gain privileges by triggering a race condition involving read and write operations with long strings.
Notes: This is an interesting case. There was a lot of discussion in the Red Hat issue tracking system about this vulnerability [1]. Such kinds of concurrency bugs can be difficult to analyze. This bug seems to be caused by an update to the pty driver file, while the fix was introduced in the tty base driver file.
Reason for discrepancy: Concurrency bugs are sometimes difficult to handle. In this case it would have been better to blame the line in the middle of the added locks.

[1] https://bugzilla.redhat.com/show_bug.cgi?id=1094232

Verdict: The Ubuntu commit seems to be the correct one. There has been a lot of discussion about this bug.

CVE-2013-1979

Date: 05/03/2013
Type: CWE-264 – Permissions, Privileges, and Access Controls
Description: The scm_set_cred function in include/net/scm.h in the Linux kernel before 3.8.11 uses incorrect uid and gid values during credentials passing, which allows local users to gain privileges via a crafted application.
Notes: This is again a semantic bug. There is a clear mention in the fixing commit that the introducing commit is the Ubuntu one.
Reason for discrepancy: VCCFinder blames a commit that changed the code but not its functionality. The vulnerable code was in the called function body. The vulnerability was introduced 2 years before.
Verdict: The Ubuntu commit is the correct one. It seems very difficult to spot this VCC with blame heuristics. VCCFinder does a good job of blaming the specific lines, however the functionality of the blamed lines existed in a function call from earlier. It remains to be seen if it is worth it to look for such behavior (i.e. look if one of the blamed lines is a function call and then look if the blamed code was part of the function body...). In general, the approach where after pinpointing a VCC some checks are carried out to see if we should go even further back in time (maybe with git dissect?) could be beneficial. This is a very difficult case for blame-based VCCs.

CVE-2011-1479

Date: 06/21/2012
Type: CWE-399 – Resource Management Errors
Description: Double free vulnerability in the inotify subsystem in the Linux kernel before 2.6.39 allows local users to cause a denial of service (system crash) via vectors involving failed attempts to create files. NOTE: this vulnerability exists because of an incorrect fix for CVE-2010-4250.
Notes: As noted in the description, this is a regression bug. VCCFinder blames a lot of older commits, since the changes in the fix affect a lot of lines. The most blamed commit is much older (4 years older than the real VCC).
Reason for discrepancy: The double-free is not direct. It happens via a function.

Verdict: The Ubuntu commit is the correct one. It seems very difficult for heuristics to find this regression bug, since the fix changes a lot of things.

CVE-2013-4470

Date: 11/04/2013
Type: CWE-264 – Permissions, Privileges, and Access Controls
Description: The Linux kernel before 3.12, when UDP Fragmentation Offload (UFO) is enabled, does not properly initialize certain data structures, which allows local users to cause a denial of service (memory corruption and system crash) or possibly gain privileges via a crafted application that uses the UDP_CORK option in a setsockopt system call and sends both short and long packets, related to the ip_ufo_append_data function in net/ipv4/ip_output.c and the ip6_ufo_append_data function in net/ipv6/ip6_output.c.
Notes: Another semantic bug (access control). There are 2 very similar fixing commits, one each for IPv4 and IPv6.
Reason for discrepancy: There is no discrepancy according to the latest results. Maybe some change to the heuristic improved performance.
Verdict: No discrepancy. The improvement seems to come from not taking into account the comments.

A.1.3. Discussion

- The Ubuntu commits are as close to a ground truth dataset as we can get
- Manual analysis is difficult and it seems previous research fell in the trap of wanting to confirm what the heuristic produced.
- Variable declarations may not need to be considered (as well as empty lines)
- After having a candidate VCC, checking if you should blame one of its ancestors may be beneficial.
- Semantic bugs (e.g. access control) seem to be a bit more difficult to trace.
- Concurrency bugs seem difficult to trace.
- Disregarding copied or refactored code may improve performance.

- If a function was replaced, it may be beneficial to look inside the function and see if the functionality changed.
- Not taking into account the comments may be beneficial.
- For the use-case of commit-based static analysis, noise in the training data may even improve performance. Noise in the evaluation data is not acceptable. Hence, the suggestion would be to use the Ubuntu ground truth as the testing dataset and improve the heuristic based on the recommendations in this paper.
- Some vulnerability types (semantic, concurrency) are inherently more difficult to trace.

A.1.4. Conclusion

The VCCs from the Ubuntu Security Team seem to be as close to a ground truth as possible. They agree with the detailed manual analysis of the above-mentioned 10 bugs. There is no reason to believe that manual analysis by us would lead to better results than what is achieved by the Ubuntu Security Team (who takes in to account commit messages and RHL reports). Some vulnerabilities are inherently difficult to trace with automated heuristics. However, existing heuristics can be improved. Tracing the VCC manually seems much harder than implied in most previous work (maybe due to confirmation bias).

A.2. Complete List of Metadata Features

- past different authors*∼
- average past different authors per file*
- added line count*∼
- average added line count per file*
- deleted line count*∼
- average deleted line count per file*
- author experience
- past changes*∼

- average past changes per file count*
- author touch count
- average author touch count per file
- hunk count∼
- average hunk count per file
- file count*
- author contribution percentage[1]
- authored hour
- authored day

[1] VCCFinder and VulDigger both have features for this but they differ from ours