VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits

Henning Perl[*], Sergej Dechand[†], Matthew Smith[*†]
[*] Fraunhofer FKIE, Germany
[†] University of Bonn, Germany

Daniel Arp, Fabian Yamaguchi, Konrad Rieck
University of Göttingen, Germany

Sascha Fahl, Yasemin Acar
Saarland University, Germany

ABSTRACT

Despite the security community's best effort, the number of serious vulnerabilities discovered in software is increasing rapidly. In theory, security audits should find and remove the vulnerabilities before the code ever gets deployed. However, due to the enormous amount of code being produced, as well as a the lack of manpower and expertise, not all code is sufficiently audited. Thus, many vulnerabilities slip into production systems. A best-practice approach is to use a code metric analysis tool, such as Flawfinder, to flag potentially dangerous code so that it can receive special attention. However, because these tools have a very high false-positive rate, the manual effort needed to find vulnerabilities remains overwhelming.

In this paper, we present a new method of finding potentially dangerous code in code repositories with a significantly lower false-positive rate than comparable systems. We combine code-metric analysis with metadata gathered from code repositories to help code review teams prioritize their work. The paper makes three contributions. First, we conducted the first large-scale mapping of CVEs to GitHub commits in order to create a vulnerable commit database. Second, based on this database, we trained a SVM classifier to flag suspicious commits. Compared to Flawfinder, our approach reduces the amount of false alarms by over 99 % at the same level of recall. Finally, we present a thorough quantitative and qualitative analysis of our approach and discuss lessons learned from the results. We will share the database as a benchmark for future research and will also provide our analysis tool as a web service.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

DOI: http://dx.doi.org/10.1145/2810103.2813604.

Keywords

Vulnerabilities; Static Analysis; Machine Learning

1. INTRODUCTION

Despite the best effort of the security community, the number of serious vulnerabilities discovered in deployed software is on the rise. The Common Vulnerabilities and Exposures (CVE) database operated by MITRE tracks the most serious vulnerabilities. In 2000, around 1,000 CVEs were registered. By 2010, there were about 4,500. In 2014, almost 8,000 CVEs were registered. The trend seems to be increasing in speed.

While it is considered a best practice to perform code reviews before code is released, as well as to retroactively checking old code, there is often not enough manpower to rigorously review all the code that should be reviewed. Although open-source projects have the advantage that anybody can, in theory, look at all the source code, and although bug-bounty programs create incentives to do so, usually only a small team of core developers reviews the code.

In order to support code reviewers in finding vulnerabilities, tools and methodologies that flag potentially dangerous code are used to narrow down the search. For C-like languages, a wide variety of code metrics can raise warning flags, such as a variable assigned inside an if-statement or unreachable cases in a switch-statement. The Clang static analyzer [1] as well as the dynamic analyzer Valgrind [3] and others, can pinpoint further pitfalls such as invalid memory access. For the Linux kernel, the Trinity system call-fuzzer [2] has found and continues to find many bugs. Finally, static analysis tools like Flawfinder [34] help find possible security vulnerabilities.

Most of these approaches operate on an entire software project and deliver a (frequently very large) list of potentially unsafe code. However, software grows incrementally and it is desirable to have tools to assist in reviewing these increments as well as tools to check entire projects. Most open-source projects manage their source code with version control systems (VCS) such as Git, Mercurial, CVS or Subversion. In such systems, code – including vulnerable code – is inserted into the software in the form of commits to the repository. Therefore, the natural unit upon which to check whether new code is dangerous is the commit. However, most existing tools cannot simply be executed on code snippets contained within a commit. Thus, if a code reviewer wants to check the security of a commit, the reviewer must

execute the analysis software on the entire project and then check if any of the warnings relate to the commit. This can be a considerable amount of work, especially since many tools require source code to be annotated and dynamic tests would have to be constructed in a way that triggers the commit.

Static and dynamic code analysis tools focus exclusively on the code without the context of who wrote the code and how it was committed. However, code repositories contain a wealth of metadata which can be highly relevant to the code quality. For instance, it can be seen whether a committer is new to the project or if they are one of the core contributors. It is possible to see the time of day or night at which code was submitted and to monitor the activity of development in certain regions of code. Moreover, most existing code-metric-based tools have very high false-positive rates, creating a (sometimes impossibly) high workload and undermining trust in the effectiveness of the tools. For instance, Flawfinder tool created 5,460 false positives warnings for only 53 true positives on the dataset used in this paper. It is intuitively clear that code reviewers who want to find 53 vulnerabilities in a set of 5,513 flagged commits have a tough time ahead of them.

In this paper, we present a classifier that can identify potentially vulnerable commits with a significantly lower false-positive rate while retain high recall rates. Therefore, unlike most existing tools for vulnerability finding, we don't focus solely on code metrics, but also leverage the rich metadata contained in code repositories.

To evaluate the effectiveness of our approach, we conduct a large-scale evaluation of 66 GitHub projects with 170,860 commits, gathering both metadata about the commits as well as mapping CVEs to commits to create a database of *vulnerability-contributing commits* (VCCs) and a benchmark for future research.

We conducted a statistical analysis of these VCCs and trained a Support Vector Machine (SVM) to detect them based on the combination of code metric analysis and GitHub metadata. For our evaluation we trained our classifier only on data up to December 31, 2010 and ran our tests against CVEs discovered in 2011–2014.

In this dataset, our approach, called *VCCFinder*, produces only 36 false positives compared to Flawfinder's 5,460 at the same level of recall. This is a reduction of over 99 % and significantly eases the workload of code reviewers.

1.1 Our Contributions

In summary, we make the following contributions in this paper:

- We present VCCFinder, a code analysis tool that flags suspicious commits by using a SVM-based detection model. Our method outperforms Flawfinder by a great margin, reducing the false positives by over 99 % at the same level of recall. Our methodology is suited to work on code snippets, enabling us to analyse code at the commit level and making a lightweight analysis of new code far easier than requiring a full build environment to be set up for each test.
- We construct the first large-scale database mapping CVEs to vulnerability-contributing commits (VCCs). The database contains 66 GitHub projects, 170,860 commits and 640 VCCs. We conduct an extensive evaluation of the methodology used to create this database to ascertain its quality as a benchmark for future research.
- We present an extensive quantitative and qualitative evaluation of VCCFinder and discuss take-aways, including, for instance that, from a security perspective, gotos are not generally harmful but in combination with error-handling code they are responsible for a significant number of VCCs.

2. RELATED WORK

The discovery of vulnerabilities in program code is a fundamental problem of computer security. Consequently, it has received much attention in the past. In the following, we give a sample of the prior work most closely related to our approach.

Static analysis.

The set of static analysis tools can be thought of as a spectrum ranging from faster, lightweight approaches to slower but more thorough techniques. With VCCFinder being a lightweight tool, we compare ourselves to FlawFinder [34], a prominent representative of this class of tools. Other lightweight approaches include Rats [9], Prefast [8] as well as Splint [10], the later requiring manual annotations.

Regarding more thorough approaches Bandhakavi et al. [11] search for vulnerabilities in browser extensions by applying static information-flow analysis to the JavaScript code. Dahse and Holz [15] introduced a static analyzer for PHP that can detect sophisticated attacks against web applications. Finally, commercial tools like Coventry [5], Fortify [6], CodeSonar [4], and IBM Security AppScan Source (formerly Rational) [7] focus on a thorough analysis with configurable rulesets and consequently long run times.

Symbolic execution.

Cadar et al. [12] present KLEE, a symbolic execution tool, which requires manual annotation and modification of the source code. Also the runtime grows exponentially with the number of paths in the program, which limits the size of project which can be tested with KLEE. Thus it is not feasible to execute KLEE on the same scale as VCCFinder. However, it is an interesting area of future work to execute KLEE as a second step after VCCFinder. Klee would then only be used on the commits flagged by VCCFinder which hopefully would significantly reduce the effort needed to run KLEE. We see these tools as complementary and separate steps in the tool chain.

Dynamic analysis.

Cho et al. [14] use a combination of symbolic and concrete execution to build an abstract model of the analyzed application and find vulnerabilities in several open-source projects. Yamaguchi et al. [37] provide an analysis platform offering fuzzy parsing of code that generates a graph representing code suitable to be mined with graph-database queries. This approach allows application-specific vulnerability patterns to be expressed; however, in contrast to our approach, it requires manual specification of these patterns by the analyst. Holler et al. [20] used fuzzing on code fragments to find vulnerabilities in the Mozilla JavaScript interpreter and the PHP interpreter.

Software metrics.

Several authors have proposed to employ software metrics to home in on regions of code more likely to contain vulnerabilities. For example, Zimmermann et al. [38] perform a large-scale empirical study on Windows Vista, indicating that metrics such as code churn, code complexity [see 22, 19] and organizational measures allow vulnerabilities to be detected with high precision at low recall rates, while code dependency measures achieve low precision at high recall rates. However, Graylin et al. [18] point out that many of these metrics may be highly correlated with lines of code. In particular, they show empirically that the relation between *cyclomatic complexity* and lines of code is near-linear, meaning that no reduction in the amount of code to read is achieved in this way.

Repository analysis.

There is a range of research work looking at software repositories in relation to software vulnerabilities. The most relevant with respect to our project can be divided into two groups: those that look at code metrics and those that look at metadata.

Neuhaus et al. [26] use the vulnerability database of the Mozilla project to extract which software components have had vulnerabilities in the past and which imports and function calls were involved. They use this to predict which software components of the Mozilla Internet suite are most likely to contain more vulnerabilities. Unlike our approach, they do not use any metadata in their analysis and the results flag entire software components rather than single commits. The results are thus more generic in the sense that they can say only that one set of software components is more worth checking than others.

On the other side, work conducted by Meneely et al. and Shin et al. analyzes different code repository metadata in relation to CVEs [25, 23, 24]. Specifically, they check how features such as code churn, lines of code, or the number of reviewers from a project's repository and review system data correlate to reported vulnerabilities. They do this manually for the Mozilla Firefox Browser, Apache HTTP server and an excerpt of the RHEL Linux kernel. Unlike the work above and our work, they do not use this data to predict vulnerabilities; moreover, unlike our work, they do not combine the features but look at each separately.

Sadeghi et al. [28] aim to reduce the number of rules used by static analysis software. For this they looked at "categorized software repositories" (i.e. the Google Play Store) and evaluated how knowledge of the app's category can reduce the number of static analysis rules needed to still retain full coverage. For this, they compared Java programs on SourceForge (without a framework) to Android apps on F-Droid (written with an application development framework). From the app's category they were able to build a predictor that helps pick a subset of static analyzer rules to apply; therefore reducing the time the static analyzer needs. Their method works especially well with apps using a framework, such as Android apps. In contrast, while this work reduces the number of rules used for analysis, we prioritize the code needed to be analysed. These approaches are complementary.

Wijayasekara et al. [35] used bug-trackers to study bugs that afterwards have been identified as vulnerabilities. The work does not deal with finding unknown vulnerabilities. While this does not directly relate to our work, bug-trackers are an interesting additional source of information.

Kim et al. [21] mined logs of a project's SCM repository for bug-introducing changes using fixed keywords (such as "fix" or "bug"). They then extracted features from these commits and trained an SVM. Our approach differs from the authors' in three ways: first, we use as a base of our research the much smaller and thus harder set of critical vulnerabilities mined from the CVE database; second, we use additional features gathered from the SCM history such as past/future different authors; third we use a historical split to evaluate our system opposed to a (random) ten-fold cross validation. The later is important since it guarantees that our system was not trained on future commits to decide if some past commit introduced a bug. Unfortunately neither the code base nor the data is available so a direct comparison is not possible. We re-ran our experiments using random cross validation and found that it increased the precision for around 15 % with a recall between 0.4 and 0.6.

Śliwerski et al. [32] present preliminary results on a statistical analysis of the Eclipse and Mozilla projects. They mined the Eclipse project's bug database for fix-inducing changes. Then, they do a statistical analysis on their data, showing that most bugs were committed on Fridays. Our work goes beyond these results in several ways. The statistical analysis is more extensive, is done on a much larger dataset and is only the first step in our system. Based on the results we train a SVM and create an adaptable system to predict vulnerability inducing commits.

Thus, our work goes beyond the above approaches in several ways. We combine both code metrics as well as metadata in our analysis and use a machine-learning approach to extract and combine relevant features as well as to create a classification engine to predict which commits are more likely to be vulnerable. In contrast to the work above, we do this for a large set of projects in an automated way instead of hand-picking features and analyzing single projects.

Machine-learning techniques.

Machine-learning and data-mining approaches have been proposed by several authors for finding vulnerabilities. For example, Scandariato et al. [31] train a classifier on textual features extracted from source code to determine vulnerable software components. Moreover, several unsupervised machine-learning approaches have been presented to assist in the discovery of vulnerabilities. For example, Yamaguchi et al. [36] introduce a method to expose missing checks in C source code by combining static tainting and techniques for anomaly detection. Similarly, Chang et al. [13] present a data-mining approach to reveal neglected conditions and discover implicit conditional rules and their violations.

However, in order to identify vulnerabilities, these approaches concentrate only on features extracted from source code. In contrast, we show that additional meta information, such as the experience of a developer, are valuable features that improve detection performance.

3. METHODOLOGY

In this section, we describe how we created a database of commits that introduced known vulnerabilities in open-source projects and which features we extracted from the commits. We will share this database with the research community as a baseline to enable a scientific comparison

between competing approaches. We focus on 66 C and C++ projects using the version control system *Git* (see appendix A for the list). These 66 projects contain 170,860 commits and 718 vulnerabilities reported by CVEs.

3.1 Vulnerability-contributing Commits

In order to analyze the common features of commits that introduce vulnerabilities, we first needed to find out which commits actually introduced vulnerabilities. To the best of our knowledge, no large-scale database exists that maps vulnerabilities as reported by CVEs to commits. Meneely et al. and Shin et al. [25, 23, 24] manually created such mappings for the Mozilla Firefox Browser, Apache HTTP server and parts of the RHEL Linux kernel. We contacted the authors to inquire whether they would share this data, since we could have used that as a baseline for our larger analysis. Unfortunately, this was not possible at the time, although the data might be released in the future. To create a freely available database for ourselves and the research community, we set out to create a method to automatically map CVEs to vulnerability-contributing commits.[1]

Since at this point we are only interested in CVEs relating to projects hosted on Github, we utilized two data sources as starting points for our mapping. As a first source, we selected all CVEs containing a link to a commit of one of the 66 projects *fixing* a vulnerability as part of the "proof". As a second source for *fixing* commits, we created a crawler that searches commit messages of the 66 projects for mentions of CVE IDs. To check the accuracy of our mapping we took a random sample of 10 % and manually checked the mapping and found no incorrectly mapped CVEs. This gave us a list of 718 CVEs. This list is potentially not complete since there might be CVEs that do not link to the fixing commit and which are also not mentioned in the commit messages. However, this does not represent a problem for our approach since 718 is a large enough sample to train our classifier.

We then developed and tested a heuristic to proceed from these fixing commits to the *vulnerability-contributing commits* (VCCs). Recall that we are operating on Git commits, which means that we have access to the whole history of a given project. One (appropriately named) Git subcommand is `git blame`, which, given a file, for each line names the commit that last changed the line. The heuristic for finding the commit that *introduced* a vulnerability given a commit that *fixed* it 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.

   Rationale: If the fix needed to change the line, that often means that it was part of the vulnerability. Note that Git diffs only know of added and deleted lines. If a line was changed, it shows up as a deletion and an addition in the diff.

3. For every continuous block of code inserted in the fixing commit, *blame* the lines before and after the block

   Rationale: Security fixes are often done by adding extra checks, often right before an access or after a function call.

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.

Our heuristic maps the 718 CVEs of our dataset to 640 VCCs. The reason we have fewer VCCs than CVEs is that a single commit can induce multiple CVEs. To estimate the accuracy of our heuristic, we took a 15 % random sample of all VCCs flagged by our heuristic (i.e. 96 VCCs) and manually checked them. We found only three cases (i.e. 3.1 %) where our heuristic blamed a wrong commit for the vulnerability. All three of the mis-mappings occurred in very large commits. For example, one commit of libtiff[2] that fixes CVE-2010-1411 also upgrades libtool to version 2.2.8. The method we propose for VCCFinder is capable of dealing with noisy datasets, so for the purpose of this work, an error rate of 3.1 % is acceptable. However, improving our blame heuristics further is an interesting avenue for future research.

Apart from the 640 VCCs, we have a large set of 169,502 unclassified commits. We name these commits unclassified, since, while no CVE points to them, they might still contain unknown vulnerabilities.

At this point we now have a large dataset mapping CVEs to vulnerability-contributing commits. Our goal now is to extract features from these VCCs in order to detect further potential VCCs in the large number of unclassified commits.

3.2 Features

First we extracted a list of characteristics that we hypothesized could distinguish commits. One of our central hypotheses is that combining code metrics with GitHub metadata features is beneficial for finding VCCs. First, we test each feature separately using statistical analysis, e.g. for each feature we measured whether the distribution of this feature within the class of vulnerable commits was statistically different from the distribution within all unclassified commits.

Here is a list of hypotheses concerning metadata we started with:

- New committers are more likely to introduce security bugs than frequent contributors.

- It is good to "commit early and often" according to the Git Best Practices[3]. Therefore, longer commits may be more suspicious than shorter ones.

- Code that has been iterated over frequently, possibly by many different authors, is more suspicious than code that doesn't change often. Meneely and Williams [23] already analyzed these code churn features in their work. We integrate and combine these features below.

Table 1 shows a list of all features along with a statistical evaluation (cf. Section 3.4) of all numerical features except

[1] We have anonymously uploaded the database to `https://www.dropbox.com/s/x1shbyw0nmd2x45/vcc-database.dump?dl=0` so the reviewers can access the raw data during the review process. We will release the data to the community together with the paper. The file was created using `pg_dump` and can be read into a database using `pg_restore`. The dump file will create the three tables *cves*, *commits* and *repositories* in the schema *export*.

[2] https://GitHub.com/vadz/libtiff/commit/31040a39

[3] `http://sethrobertson.GitHub.io/GitBestPractices/#commit`

Feature	Scope	mean VCCs	mean others	U	effect size
Number of commits	Repository	282 171.39	103 980.95	32143126*	40 %
Number of unique contributors	Repository	524.99	236.90	30528184*	43 %
Contributions in project	Author	5 %	15 %	31263040*	42 %
Additions	Commit	306.19	71.54	20215148*	62 %
Deletions	Commit	73.93	37.46	42983290*	20 %
Past changes	Commit	627.17	385.53	40715632*	24 %
Future changes	Commit	792.46	396.63	36261346*	33 %
Past different authors	Commit	40.16	22.70	40292116*	25 %
Future different authors	Commit	136.58	51.44	29534644*	45 %
Hunk count	Commit	17.68	9.88	32348343*	40 %
Commit message [1]	Commit	—	—	—	—
Commit patch [1]	Commit	—	—	—	—
Keywords [2]	Commit	—	—	—	—
Added functions	Function	6.51	1.03	28724694*	46 %
Deleted functions	Function	1.07	0.49	50084674*	7 %
Modified functions	Function	6.79	3.59	41446509*	23 %

[1] These features are text-based and thus not considered in the statistical analysis.
[2] See Table 2 for a statisical analyis of each keyword.

Table 1: Overview of the features and results of the statistical analysis of the numeric features. Mann–Whitney U test significant (*) if $p < 0.00059$.

for project-scoped features. In the following, we discuss the features. For brevity reasons, we omit the discussion of self-explaining features here.All our analyses are based on commits. A commit can contain changes to one or more files. The metrics about files and functions are aggregated in the corresponding commit.

Features scoped by project are obviously the same for every commit in that project. However, in combination with other commit-based features, these can still become relevant.

3.2.1 Features Scoped by Project

Programming language The primary language the project is written in, as determined by GitHub through their open-source linguist library. In our analysis, we focused on projects written in either C or C++. The main reason for limiting our focus to one language was that we wanted to ensure comparability between the features extracted from the commit patches. When mixing different languages and syntaxes, this can't be ensured. We chose C and C++ specifically since many security-relevant projects (Linux, Kerberos, OpenSSL, etc.) are written in these languages.

Star count (number) The number of stars the project has received on GitHub. Stars are a user's way of keeping track of interesting projects, as starred projects show up on the own profile page.

Fork count (number) To fork a project on GitHub means copying the repository under your personal namespace. This is often the first step to contributing back to the project by then making changes under the personal namespace and sending a pull request to the official repository.

Number of commits (number) We counted the number of commits that are reachable from the main branches HEAD. The canonical main branch is "master", but some projects like bestpractical/rt use "stable" as the default branch. In those cases we used the branch set at GitHub by the maintainer of the project.

3.2.2 Features Scoped by Author

Contributions (percentage) How many commits the author has made in this project in percent, i.e. the number of commits authored divided by the number of total commits.

3.2.3 Features Scoped by Commit

Number of Hunks (number) As a hunk is a continuous block of changes in a diff, this number assesses how fragmented the commit is (i.e. lots of changes all over the project versus one big change in one file or function).

Patch (text) All changes made by the commit as text represented as a bag of words.

Patch keywords (number) For each patch, we counted the number of occurrences of each C/C++ keyword. See Table 2 for a statistical analysis of the different distributions of each keyword.

3.2.4 Features Scoped by File

Future changes (number) If the commit at hand is not the most current one, this is the number of times the file will be changed by later commits. We only use this feature for our historical analysis and not for the classifier, since this feature is naturally not available for new commits.

Keyword	mean VCCs	mean others	U	effect size
if	39.00	7.82	37013390*	70 %
int	31.30	7.02	39930128*	68 %
struct	32.38	3.66	39729656*	68 %
return	18.76	3.60	41342834*	67 %
static	15.17	3.58	45382955*	64 %
void	12.52	4.31	63935365*	49 %
unsigned	8.66	1.51	64440969*	48 %
goto	5.92	0.43	64798818*	48 %
sizeof	4.37	0.78	66764357*	46 %
break	5.56	0.84	74389604*	40 %
char	6.71	2.68	93400907*	25 %

Table 2: Statistical analysis of C/C++ keywords sorted by effect size [33], Mann–Whitney U test significant (*) if $p < 0.000357$.

3.3 Excluded Features

As can be seen in Table 1, the vast majority of the features depend only on data gathered from the version control system and not from additional information on GitHub or any other platform. In fact, we left out some features that were only available on some projects or for few commits since the data was too sparse to reveal anything reliable. We will briefly discuss why we excluded some features which might seem counter-intuitive.

One feature that would be promising but which we did not include was *issue tracker information*. GitHub provides an issue tracker and even links texts like "fixes #123" in the commit message to the corresponding issue. However, the projects which use this feature tend to be smaller projects, while the older and larger projects for which we have a rich set of CVE data predominantly use an external issue tracker. Thus, this feature is not useful for us at this time.

Another piece of information that is interesting – but unfortunately too sparse at the moment – is the content of the discussion surrounding the inclusion of a change into the main repository. For this information, features could be the length of the discussion, the number of people involved, or the mean experience (in terms of contributions) of the people involved. Projects that use GitHub's functionalities extensively often do this through "pull requests". A contributor submits a commit to his own, unofficial repository and subsequently notifies the maintainer of the official repository to pull in the changes he made. GitHub provides good support for this work flow, including the ability to make comments on a pending pull request. Although this data could be useful for the classification of commits, at this point, too few projects use this work flow to be useful.

3.4 Statistical Analysis of Features

For each numerical feature, we wanted to assess its fitness with respect to distinguishing VCCs from unclassified commits. We used the Mann–Whitney U test[4] in order to compare the distribution of a given feature within the set of commits with vulnerabilities against the set of all unclassified commits. The null hypothesis states that the feature is distributed independently from whether the commit contained a bug or not. If we can reject the null hypothesis, the feature is distributed differently in each set and thus is a promising candidate as input for the machine-learning algorithms.

We used the Bonferroni correction to correct for multiple testing for the 17 features we tested. Therefore, we test against the stricter significance level of 0.00059, which corresponds to a non-corrected $p \leq 0.01$ for each individual test. The date and time features (project age and commit with time zone) were converted to numerical features based on seconds that have elapsed since January 1, 1970 UTC (Unix epoch).

3.4.1 Features Scoped by Project

These features were attributed to the commit depending on the project the commit was taken from. Since all commits from a repository, whether containing vulnerabilities or not, have the same features, these features are too broad to actually distinguish commits. However, they can be valuable in combination with other features later on. For brevity, we do not discuss the features on their own here, though the table shows the significance testing.

3.4.2 Patch Keyword Features

For each commit we counted the occurrences of each of the following 28 C/C++ keywords: `bool`, `char`, `const`, `extern`, `false`, `float`, `for`, `if`, `int`, `long`, `namespace`, `new`, `operator`, `private`, `protected`, `sizeof`, `static`, `static`, `struct`, `switch`, `template`, `throw`, `typedef`, `typename`, `union`, `unsigned`, `virtual`, and `volatile`. We then used the Mann–Whitney U test to find out whether the given keyword is used more or less frequently in VCCs compared to unclassified commits. Table 2 shows a subset of those keywords with high significance and high effect. We say that an effect is significant if $p < 0.000357$, corresponding to 0.01/28, again accounting for a Bonferroni correction for multiple testing for the 28 keywords.

The effect size measures the percentage of pairs that support the hypothesis. For example, for the keyword `if`, the vulnerable commits contain more `if`s than the unclassified commits in 70 % of the cases. As can be seen by looking at the mean values for each distribution, if there is a statistical effect, the VCCs are more likely to contain those keywords compared to unclassified commits.

3.4.3 Features Scoped by Commit or File

All remaining features except for the number of deleted lines are distributed differently over VCC versus unclassified commits, with $p = 3.9 \times 10^{-6}$ the number of hunks being the least significant result. We note that the fact that a feature is distributed differently does not mean that this feature can be used to distinguish between the two sets. However, these results provide some hint as to why a machine-learning approach that uses a combination of these features can be successful.

The only feature where the difference was not significant was the number of deleted lines ($p = 4.6 \times 10^{-4}$), contrary to the number of added lines ($p = 3.9 \times 10^{-37}$), for which there is a significant difference in the distribution. When we manually looked at commits with known vulnerabilities and compared them to unclassified commits, we saw that the former often added a great deal of code, whereas the number of deleted or edited lines were the same as for un-

[4]The Mann–Whitney U test is used to test whether a value is distributed differently between two populations.

classified commits. This finding confirms the intuition that security bugs are not commonly introduced by code edits or refactoring, but that new code is a more likely entry points for vulnerabilities. To the best of our knowledge this fact has not been used to ease the workload of code reviewers.

3.4.4 Text-Based Features

One of the central tenets of our work is that combining code metrics with GitHub metadata can help with the detection of VCCs. While both the code and the metadata features detailed above are "hard" numerical features, there are also a number "soft" features contained in GitHub that can be helpful. These text-based features, like the commit message, cannot be evaluated using statistical tests as above, but will be integrated into the machine-learning algorithm using a generalized bag-of-words model as we will discuss in Section 4.1.

4. LEARNING-BASED DETECTION

The different features presented in the previous sections provide information for analyzing the search for suspicious commits and the discovery of potential vulnerabilities. As the large number of these features renders the manual construction of detection rules difficult, we apply techniques from the area of machine-learning to automatically analyze the commits and rank them so code-reviewers can prioritise their work. The construction of a learning-based classifier, however, poses several challenges that need to be addressed to make our approach useful in practice:

1. *Generality:* Our features comprise information that range from numerical code metrics to structured metadata, such as words in commit messages and keywords in code. Consequently, we strive for a classifier that is capable of jointly analyzing these heterogeneous features and inferring a combined detection model.

2. *Scalability:* To analyze large code repositories with thousands of source files and commits, we require a very efficient learning method which is able to operate on the large amount of available features in reasonable time.

3. *Explainability:* To help an analyst in practice, it is beneficial if the classifier can give a human-comprehensible explanation as to why the commit was flagged, instead of requiring an analyst to blindly trust a black-box decision.

We address these challenges by combining two concepts from the domains of machine-learning and information retrieval. In particular, we first create a joint representation for the heterogeneous features using a *generalized bag-of-words model* and then apply a *linear* Support Vector Machine (SVM)—a learning method that can be extended to provide explanations for its decisions and which is also efficient enough to cope with the large number of features which need to be analysed.

4.1 Generalized Bag-of-Words Models

Bag-of-word models have been initially designed for analysis of text documents [30, 29]. In order to combine both code metric based numerical features with GitHub metadata features, we generalize these models by considering a generic set of tokens S for our analysis. This set can contain textual words from commit messages as well as keywords, identifiers and other tokens from the code of a commit. In particular, we obtain these tokens by splitting the commit message and its code using spaces and newlines. Furthermore, we ignore certain tokens, such as author names and email addresses, since they might bias the generality of our classifier and could compromise privacy.

Formally, we define the mapping φ from a commit to a vector space as

$$\varphi : X \longrightarrow \mathbb{R}^{|S|}, \quad \varphi : x \longmapsto \big(b(x,s)\big)_{s\in S},$$

where X is the set of all commits and $x \in X$ an individual commit to be embedded in the vector space. The auxiliary function $b(x,s)$ returns a binary flag for the presence of a token s in x and is given by

$$b(x,s) = \begin{cases} 1 & \text{if token } s \text{ is contained in } x \\ 0 & \text{otherwise.} \end{cases}$$

To also incorporate numerical features like the author contribution into this model, we additionally convert all numerical features into strings. This enables us to add all arbitrary numbers to S and thereby treat both kinds of features equally. However, when using a string representation for numerical features we have to ensure that similar values are still identified as being similar. This is obviously not the case for a naive mapping, as "1.01" and "0.99" represent totally different strings.

We tackle this problem by mapping all numerical features to a discrete grid of bins prior to the vector space embedding. This quantization ensures that similar values fall into the same bins. We choose different bin sizes depending on the type of the feature. If the numerical values are rather evenly distributed, we apply a uniform grid, whereas for features with skewed distribution we a apply a logarithmic partitioning. For the latter, we apply the logarithmic function to its values and cut off all digits after the first decimal place.

To better understand this generalized bag-of-words model, let us consider a fictitious commit x, where a patch has been written by a user who did not contribute to a project before. The committed patch is written in C and contains a call to an API function which is associated with a buffer write operation. The corresponding vector representation of the commit x looks as follows

$$\varphi(x) \mapsto \begin{pmatrix} \cdots \\ 1 \\ 0 \\ \cdots \\ 1 \\ 0 \\ \cdots \end{pmatrix} \begin{matrix} \cdots \\ \texttt{AUTHOR\_CONTRIBUTION:0.0} \\ \texttt{AUTHOR\_CONTRIBUTION:10.0} \\ \cdots \\ \texttt{buf\_write\_func();} \\ \texttt{some\_other\_func();} \\ \cdots \end{matrix}$$

The two tokens indicative of the commit are reflected by non-zero dimensions, while all unrelated tokens are associated with zero dimensions. Note that the resulting vector space is high-dimensional and may contain several thousands of dimensions. For a concrete commit x, however, the vast majority of these dimensions are zero and thus the vector $\varphi(x)$ can be stored in a sparse data structure. We make use of the open-source tool Sally [27] for this purpose, which implements different strategies for extracting and storing sparse feature vectors.

4.2 Classification and Explainability

While in principle a wide range of methods are available for learning a classifier for the detection of vulnerability contributing commits, only few methods scale with larger amount of data while also providing explanations for their decisions. One technique satisfying both properties are *linear Support Vector Machines* (SVM). This variant of classic SVMs does not apply the kernel trick for learning, but instead directly operates in the input space. As a result, the run-time complexity of a linear SVM scales linearly in the number of vectors and features.

We implement our classifier for commits using the open-source tool LibLinear [17] that provides different optimization algorithms for linear SVMs. Each of these algorithms seeks a hyperplane w that separates two given classes with maximum margin, in our case corresponding to unclassified commits and vulnerability-contributing commits. As the learning is performed in the input space, we can use this hyperplane vector w for explaining the decisions of our classifier.

By calculating the inner product between $\varphi(x)$ and the vector w, we obtain a score which describes the distance from $\varphi(x)$ to the hyperplane; that is, how likely the commit introduces a vulnerability, $f(x) = \langle \varphi(x), w \rangle = \sum_{s \in S} w_s \, b(x, s)$.

As this inner product is computed using a summation over each feature, we can simply test which features provide the biggest contribution to this distance and thus are causal for the decision.

Finally, to calibrate free parameters of the linear SVM, namely the regularization parameter C and the class weight W, we perform a standard cross-validation on the training data. We then pick the best values corresponding to a regularization cost $C = 1$ and a weight $W = 100$ for the class of suspicious commits.

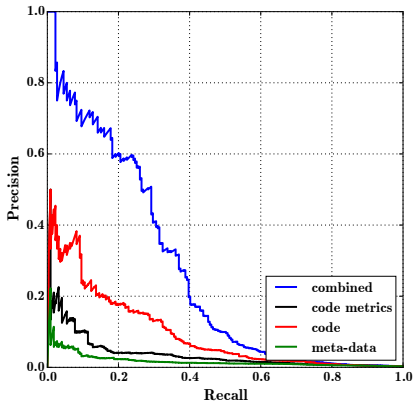
5. EVALUATION

We evaluate the effectiveness of our approach in several different ways. First, we use a temporal split between the training and test data to evaluate the predictiveness of the SVM. We picked 2011 as the split data to have the relation of two-thirds to one-thirds training vs test data.[5] Since we have the ground truth for the years 2011 to 2014 this method is the allows us to realistically and reliably test the effectiveness of VCCFinder.

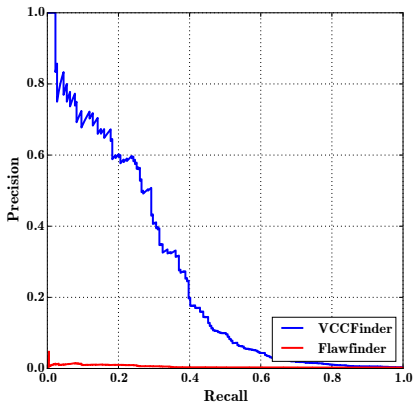
	Dataset		
	Historical	Test	Total
CVEs	469	249	718
VCCs	421	219	640
Unclassified commits	90,282	79,220	169,502

Table 3: Distribution of commits, CVEs, and VCCs.

[5]This is a standard approach to evaluate classifiers. The first dataset contains all commit data up until the 31st of December 2010. We use this dataset for the design and training of our classifier. The dataset can be considered the 'historical' dataset. The second 'testing' dataset contains all commit data from 2011 to 2014 which is then used to evaluate our approach. This simulates VCCFinder being used in the beginning of 2011 having being trained on all existing data at the time and then trying to predict the unkown VCCs of the future (2011 to 2014).



(a) Detection performance of VCCFinder using different feature sets.



(b) Detection performance of our approach and FlawFinder as precision-recall curve.

Figure 1: Detection performance of VCCFinder.

Second, we discuss the features learnt by the SVM as well as the true positives, i.e. vulnerabilities our classifier found in the test set. Third, we discuss the commits that are flagged by our classifier but lie outside the ground truth we have based on the CVEs. These could either be false positives or point to previously undetected vulnerabilities. Finally, we compare our approach to Flawfinder, an open-source static code analyzer.

Comparing feature sets.

We start with an evaluation of the impact of different feature sets and their combination on the detection performance of our classifier. Figure 1(a) shows the precision-recall curves for these experiments. To this end, we train a classifier on code metric features and meta-information. As can be seen, the classifier that combines all the features (shown in blue) out-performs the classifiers which only operate on a sub-set of the features, showing that combining the different features is beneficial. Figure 1(b) shows the precision recall curve of our VCCFinder compared to Flawfinder, which only operates on code metrics. The comparison with Flawfinder will be discussed in greater depth in section 5.3.

5.1 Case Study

The previous section shows the precision of our approach for the different levels of recall. In practice, developers can simply decide how many commits they can afford (time- and cost-wise) to review and VCCFinder will improve their chances of finding vulnerabilities. For the sake of comparison with Flawfinder, we now set VCCFinder's recall to the same as that of Flawfinder (i.e. 0.24 cf. Table 4) and discuss some examples of the VCCs which would have been flagged by VCCFinder if it had been run from 2011 to 2014[6]. In these four years, VCCFinder would only have flagged 89 out of 79688 commits for manual review compared to 5,513 commits flagged by Flawfinder. We believe this is a very manageable amount of code reviews to ask reviewers to do for a high return. Additionally, projects can increase the number of commits to review at any time. In the following, we present an excerpt of the vulnerabilities that VCCFinder found, when set at the very conservative level of Flawfinder's recall. We also discuss which features our classifier used to spot the VCCs.

CVE-2012-2119.

Commit `97bc3633be` *includes a buffer overflow in the macvtap device driver in the Linux kernel before 3.4.5, when running in certain configurations, allows privileged KVM guest users to cause a denial of service (crash) via a long descriptor with a long vector length*[7]. Considering metadata, our SVM detects this commit because of the edited file's high code churn, and because the author made few contributions to the Kernel in combination with the fact the the developer used `sockets`.

CVE-2013-0862.

FFmpeg commit `69254f4628` *introduces multiple integer overflows in the* process_frame_obj *function in libavcodec / sanm.c in FFmpeg before 1.1.2 that allow remote attackers to have an unspecified impact via crafted image dimensions in LucasArts Smush video data, which triggers an out-of-bounds array access*[8]. The SVM detected that the author contributed little to the project before as well as that the commit inserted a large chunk of code at once.

CVE-2014-1438.

In commit `1361b83a13`, *the* restore_fpu_checking *function in* arch/x86/include/asm/fpu-internal.h *in the Linux kernel before 3.12.8 on the AMD K7 and K8 platforms does not clear pending exceptions before proceeding to an EMMS instruction, which allows local users to cause a denial of service (task kill) or possibly gain privileges via a crafted application.*[9] The SVM detected a high amount of exceptions, a high number of changed code, inline ASM code, and variables containing user input such as `__input` and `user`.

CVE-2014-0148.

In commit `e8d4e5ffdb` of Qemu *the block driver for Hyper-V VHDX Images is vulnerable to infinite loops and other potential issues when calculating BAT entries. This is due to missing bounds checks for* block_size *and* logical_sector_size *variables*[10]. The SVM found that the patch of the VCC included many keywords indicating errorprone byte manipulation, such as "opaque", "*bs", or "bytes".

5.2 Flagged Unclassified Commits

While we discussed the known true positive hits of our classifier for the years 2011 to 2014 above, we also have 36 commits that were flagged as potentially dangerous, for which we have no known CVE. These are commits that need be checked by code reviewers. We have shared our results with several code reviewing teams and will follow responsible disclosure in all cases, so we cannot discuss the flagged commits at this time. However, we can already talk about one vulnerability found by VCCFinder in commit `d08d7142fd` of the FFmpeg project, since this vulnerability was fixed in commit `cca1a42653` before it ever was released. Thus, discussing the findings poses no harm to the FFmpeg project.

Commit `d08d7142fd` of FFmpeg introduces a new codec for Sierra Online audio files and Apple QuickDraw and was flagged in the 101 commits, but is not associated with a CVE. However, we discovered that in the newly created file `libavcodec/qdrw.c`, starting at line 72, the author does not check the size of an integer read from an adversary-supplied buffer.

```
for (i = 0; i <= colors; i++) {
  int idx;
  idx = BE_16(buf); /* color index */
  buf += 2;

  a->palette[idx * 3 + 0] = *buf++;
  buf++;
  a->palette[idx * 3 + 1] = *buf++;
  buf++;
  a->palette[idx * 3 + 2] = *buf++;
  buf++;
}
```

The macro `BE_16()` reads two bytes from the argument and returns an unsigned 16 bit integer. This means that an adversary controlling `buf` (e.g. through a malicious video) could address $3 \cdot 65535$ bytes of memory which will be filled by data from `buf` itself.

The SVM classified the commit because of raw byte manipulation, indicated by uses of "buf" as well as an inexperienced committer pushing a large chunk of code at once.

5.3 Comparison to Flawfinder

We compare our findings against *Flawfinder* [34] version 1.31, a static source code scanner. Flawfinder is a mature open-source tool that has been under active development since 2001 and fulfills the requirements of being able to process C and C++ code on the level of commits. When given a source file, Flawfinder returns lines with suspected vulnerabilities. It offers a short explanation of the finding as well as a link to the Common Weakness Enumeration (CVE) database.[11] For the comparison, we run Flawfinder on each

[6] As previously mentioned we use the years 2011–2014 as the test dataset, since we have ground truth data on which to base the discussion.

[7] `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-2119`

[8] `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0862`

[9] `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1438`

[10] `https://bugzilla.redhat.com/show_bug.cgi?id=1078212`

[11] `http://cwe.mitre.org/`

added or modified file of a commit. We then record the lines which Flawfinder flags that were inserted by the commit. Consequently, we say that Flawfinder marked a commit if it found a flaw in one of the lines the commit inserted.

We then evaluated both our tool and Flawfinder against the test dataset. Table 4 shows the contingency table, precision and recall for both tools. We argue that precision is the most important metric in this table and the one which should be used to compare Flawfinder and VCCFinder, as this value determines how many code locations a security researcher needs to look at in order to find a vulnerability. While a higher recall would *theoretically* mean that more vulnerabilities can be found, in practice they would be buried in a large amount of false positives. So for now, we accept that we will not find all vulnerabilities but create an environment in which it is realistic for a reviewer to check all flagged commits and achieve a decent success rate. Each row compares VCCFinder to Flawfinder with a different configuration. In the first row, we set VCCFinder's recall to that of Flawfinder's. As can be seen, VCCFinder's precision is significantly higher. Our approach improves the false positive rate by over 99 %! This is the most realistic configuration, since this configuration can be used in a real world setting. For the next comparison, we set VCCFinder's false positives to the same number as Flawfinder's. While of course the number of false positives is then prohibitively high, VCCFinder does find almost three times as many VCCs as Flawfinder. In the final comparison, we set VCCFinder's precision to Flawfinder's very poor value. While the number of false positives is prohibitively high, VCCFinder finds almost 90% of all VCCs compared to Flawfinder's 24%.

In Table 5 we also compare VCCFinder and Flawfinder based on their top results. In the first row, we select the top 100 flagged commits, then 500 and finally 1000. Among the top 100 commits, VCCFinder identifies 56 VCCs correctly, significantly reducing the amount of commits a security researcher would need to review before finding a commit containing a vulnerability. Compared to FlawFinder, its precision is more than 50 times higher and it already identifies more than 25% of all VCCs in the data set at this point.

VCCFinder significantly outperforms Flawfinder in all parameter configurations. Importantly, we were able to reduce the number of false positives to the point where it becomes realistic for reviewers to carefully check all flagged commits. This represents a significant improvement over the current state-of-the-art.

We would have liked to compare our approach to more alternatives; however, since most research papers have not published the datasets they worked on and since their tools are not applicable to commits at the scale at which we tested VCCFinder, this was not possible. We are releasing our VCC database and results to the community, so that future researchers have a benchmark against which different approaches can be compared.

6. TAKE-AWAYS

As the results above show, the performance of VCCFinder means that it can realistically be used in production environments without overburdening developers with a huge number of reviews. Since it can work on code snippets it can used automatically when new commits come in without requiring a complex test environment.

	TP	FP	FN	TN	Precision	Recall
Flawfinder						
Top 100	1	99	218	79121	0.01	0.00
Top 500	6	494	213	78726	0.01	0.03
Top 1000	13	987	206	78233	0.01	0.06
VCCFinder						
Top 100	56	44	163	79176	0.56	0.26
Top 500	88	412	131	78808	0.18	0.40
Top 1000	105	895	114	78325	0.11	0.48

Table 5: Confusion matrix of the tools by top X commits. T: True, F: False, P: Positive, N: Negative.

Apart from this, we would like to present some qualitative take-aways we found while developing and evaluating VCCFinder, which can be useful even without using the tool. While some of these take-aways confirm well-known beliefs, we found it interesting to see that our machine-learning approach also came to these conclusions and backed them up with quantitative data, but also generated new insights.

Error handling is hard.

When looking at the features the SVM learnt by classifying VCCs, we saw that the adage "gotos considered harmful" [16] still holds true today, as amongst others the keyword `goto` and the according jump labels such as `out:` and `error:` increase the likelihood of vulnerable code. We can confirm this by looking at Table 2. However, we found that the SVM also flags returning error values such as `-EINVAL` as potentially dangerous. Combined with gotos, these are common C mechanisms for error and exception handling. So unlike Dijkstra's argument that gotos are harmful because they lead to unreadable code, in our context gotos are considered harmful because they frequently occur in an error-handling context. So instead of merely detecting gotos, our SVM gives *exception and error-handling* code a higher potential vulnerability ranking. Our explanation of this effect is as follows: because it is easy to miss some cases, exception handling is easy to get wrong (e.g. Apple's goto fail bug in their TLS implementation[12]).

Variable Usage and Memory Management.

When examining highly ranked features of the SVM, we noticed that some memory management constructs lead to a higher vulnerability ranking. For instance, `sizeof(struct`, a high usage of `sizeof` in general, `len` and `length` as variable names occurred more often in vulnerable commits. In addition, we observed that variable names consisting of specific strings often occurred in VCCs: `buf`, `net`, `socket` and `sk`. While the presented keywords and variables alone do not lead to vulnerabilities, they may indicate more critical areas of the code.

[12] `https://www.imperialviolet.org/2014/02/22/applebug.html`

	True positive	False positive	False negative	True negative	Precision	Recall
Flawfinder	53	**5,460**	166	73760	0.01	0.24
VCCFinder						
– with same recall/true positives	53	36	166	79184	0.60	0.24
– with same number of false positives	144	5460	75	73760	0.03	0.66
– with same precision	185	24288	34	54932	0.01	0.84

Table 4: Comparison of the tools

Help new contributors.

We found that new contributors, i.e. contributors with less than 1 % of all commits in a given project, are about five times as likely to commit a vulnerability as their counterparts who frequently contribute. While new contributors authored 470 of 95,621 VCCs, or 0.49 %, frequent contributors authored only 244 of 255,074 VCCs, or 0.10 % (Pearson's χ^2: $p < 0.0001$). While this is of course also no big surprise, we hope quantifying this risk will help convince projects to introduce more stringent review policies.

Final Thoughts.

As both the evaluation and the take-aways show, commits have a myriad of possible reasons for being flagged as VCCs. These reasons can be code-based or metadata-based or, importantly, a combination of the two. The examples above give us an intuitive understanding of why this is. While our main recommendation is to use VCCFinder to classify potentially vulnerable commits to prioritize reviews, there also general recommendation which can be extracted from the classifier results.

7. LIMITATIONS

Our approach has several limitations. We selected 66 open-source projects written in C or C++ that created at least one CVE but otherwise varied in numbers of contributors, commits, or governance. We believe that applying our results to other projects using C or C++ should not threaten the validity. However, we can make no predictions on how VCCFinder performs on projects which to date have not received any CVEs. For generalization to other programming languages, the feature extraction and training will need to be re-done per language, so that the SVM does not mis-train based on differences in syntax.

We used a heuristic to map CVEs to VCCs. Our manual analysis of 15 % of these mappings showed that we have an error rate of 3.1 %. This needs to be taken into account by any project building on this dataset.

While we were able to map CVEs to VCCs, it is of course unknown how many unknown vulnerabilities are contained in our annotated database. Thus, our true positives must be considered a lower bound and the false positives an upper bound. So both VCCFinder's and Flawfinder's results might be better than reported and the relation between them could change. However, since VCCFinder outperforms Flawfinder by a large margin, it seems unlikely that the outcome would change.

Our experiments demonstrate that VCCFinder is able to automatically spot vulnerability-contributing commits with high precision; yet this alone does not ensure that an underlying vulnerability will be uncovered. Significant work and expertise is still necessary to audit commits for potential security flaws. However, our approach reduces the amount of code to inspect considerably and thus helps increase the effectiveness of code audits.

8. CONCLUSION

In this paper, we present and evaluate VCCFinder, an approach to improve code audits. Our approach combines code-metric analysis with meta data gathered from code repositories using machine-learning techniques. Our results show that our approach significantly outperforms the vulnerability finder Flawfinder. We created a large test database containing 66 C and C++ project with 170,860 commits on which to evaluate and compare our approach. Training our classifier on data up until 2010 and testing it against data from 2011 to 2014, VCCFinder produced 99% fewer false positives than Flawfinder, detecting 53 of the 219 known vulnerabilities and only producing 36 false positives compared to Flawfinder's 5,460 false positives.

To enable future research in this area, we will release our annotated VCC database and results so that future approaches can use this database both as a training set and as a benchmark to compare themselves to existing approaches. The community is currently lacking such a baseline and we hope to spur more comparable research in this domain.

We see a very large amount of interesting future work. While the results are already significantly better than the Flawfinder tool, we believe that we have only begun to scratch the surface of what can be ascertained by combining the different features. Further analyzing the results of the classifier will likely allow us to make more general recommendations on how to minimize the likelihood that vulnerabilities make it from the initial vulnerable commit into deployed software.

References

[1] Clang static analyzer. `http://clang-analyzer.llvm.org/`. Accessed: 2015-05-08.

[2] Trinity: A linux system call fuzzer. `http://codemonkey.org.uk/projects/trinity/`. Accessed: 2015-05-08.

[3] Valgrind. `http://valgrind.org/`. Accessed: 2015-05-08.

[4] CodeSonar® | GrammaTech static analysis. `https://www.grammatech.com/codesonar/`, visited August, 2015.

[5] Coverity Scan — static analysis. `https://scan.coverity.com/`, visited August, 2015.

[6] HP Fortify. `https://www.hpfod.com/`, visited August, 2015.

[7] IBM Security AppScan Source. https://www.ibm.com/software/products/en/appscan-source/, visited August, 2015.

[8] PREfast analysis tool. https://msdn.microsoft.com/en-us/library/ms933794.aspx, visited January, 2015.

[9] Rough auditing tool for security (RATS). https://code.google.com/p/rough-auditing-tool-for-security/, visited January, 2015.

[10] Splint – annotation-assisted lightweight static checking. http://splint.org/, visited January, 2015.

[11] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. VEX: Vetting browser extensions for security vulnerabilities. In *USENIX Security Symposium*, volume 10, pages 339–354, 2010.

[12] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[13] R.-Y. Chang, A. Podgurski, and J. Yang. Discovering neglected conditions in software by mining dependence graphs. *Software Engineering, IEEE Transactions on*, 34 (5):579–596, Sept 2008.

[14] C. Y. Cho, D. Babic, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song. MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *USENIX Security Symposium*, pages 139–154, 2011.

[15] J. Dahse and T. Holz. Static detection of second-order vulnerabilities in web applications. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 989–1003, San Diego, CA, Aug. 2014. USENIX Association.

[16] E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3): 147–148, 1968.

[17] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research (JMLR)*, 9:1871–1874, 2008.

[18] J. Graylin, J. E. Hale, R. K. Smith, H. David, N. A. Kraft, W. Charles, et al. Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship. *Journal of Software Engineering and Applications*, 2(03):137, 2009.

[19] M. H. Halstead. *Elements of software science*. Elsevier computer science library : operational programming systems series. North-Holland, New York, NY, 1977.

[20] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, Bellevue, WA, 2012. USENIX.

[21] S. Kim, E. J. Whitehead Jr, and Y. Zhang. Classifying software changes: Clean or buggy? *Software Engineering, IEEE Transactions on*, 34(2):181–196, 2008.

[22] T. J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[23] A. Meneely and O. Williams. Interactive churn metrics: Socio-technical variants of code churn. *SIGSOFT Softw. Eng. Notes*, 37(6):1–6, Nov. 2012.

[24] A. Meneely, H. Srinivasan, A. Musa, A. Rodriguez Tejeda, M. Mokary, and B. Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*, pages 65–74, Oct 2013.

[25] A. Meneely, A. C. R. Tejeda, B. Spates, S. Trudeau, D. Neuberger, K. Whitlock, C. Ketant, and K. Davis. An empirical investigation of socio-technical code review metrics and security vulnerabilities. In *Proceedings of the 6th International Workshop on Social Software Engineering*, SSE 2014, pages 37–44. ACM, 2014.

[26] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540. ACM, 2007.

[27] K. Rieck, C. Wressnegger, and A. Bikadorov. Sally: A tool for embedding strings in vector spaces. *Journal of Machine Learning Research (JMLR)*, 13(Nov):3247–3251, Nov. 2012.

[28] A. Sadeghi, N. Esfahani, and S. Malek. Mining the categorized software repositories to improve the analysis of security vulnerabilities. In *Fundamental Approaches to Software Engineering*, pages 155–169. Springer, 2014.

[29] G. Salton. Mathematics and information retrieval. *Journal of Documentation*, 35(1):1–29, 1979.

[30] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1986.

[31] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen. Predicting vulnerable software components via text mining. *Software Engineering, IEEE Transactions on*, 40(10): 993–1006, Oct 2014.

[32] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *ACM Sigsoft Software Engineering Notes*, 30(4):1–5, 2005.

[33] H. W. Wendt. Dealing with a common problem in social science: A simplified rank-biserial coefficient of correlation based on the u statistic. *European Journal of Social Psychology*, 2(4):463–465, 1972.

[34] D. A. Wheeler. Flawfinder. http://www.dwheeler.com/flawfinder/, visited January, 2015.

[35] D. Wijayasekara, M. Manic, J. L. Wright, and M. McQueen. Mining bug databases for unidentified software vulnerabilities. In *Human System Interactions (HSI), 2012 5th International Conference on*, pages 89–96. IEEE, 2012.

[36] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 499–510. ACM, 2013.

[37] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014.

[38] T. Zimmermann, N. Nagappan, and L. Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 421–428. IEEE, 2010.

APPENDIX

A. LIST OF REPOSITORIES

We used the following list of repositories: Portspoof, GnuPG, Kerberos, PHP, MapServer, HHVM, Mozilla Gecko, Quagga, libav, Libreswan, Redland Raptor RDF syntax library, charybdis, Jabberd2, ClusterLabs pacemaker, bdwgc, pango, qemu, glibc, OpenVPN, torque, curl, jansson, PostgreSQL, corosync, tinc, FFmpeg, nedmalloc, mosh, trojita, inspircd, nspluginwrapper, cherokee webserver, openssl, libfep, quassel, polarssl, radvd, tntnet, Android Platform Bionic, uzbl, LibRaw, znc, nbd, Pidgin, V8, SpiderLabs ModSecurity, file, graphviz, Linux Kernel, libtiff, ZRTPCPP, taglib, suhosin, Phusion passenger, monkey, memcached, lxc, libguestfs, libarchive, Beanstalkd, Flac, libX11, Xen, libvirt, Wireshark, and Apache HTTPD.