Automatic Inference of ‘hidden’ Security Fixes in Source Code Repositories

Daniel Gareev
University of Luxembourg
Email: daniel.gareev.001@student.uni.lu

Stanislav Dashevskyi
University of Luxembourg
Email: stanislav.dashevskyi@uni.lu

Abstract

The adoption of open-source software (OSS) in the software industry has continued to rise steadily over the past few years. Despite the community’s best effort, the number of vulnerabilities discovered in open-source software has been steadily increasing and shows no signs of slowing down. However, due to a large amount of code being produced, some of them may not go through public disclosure.

Empirical methods for software security testing and finding software defects rely on links between bug databases, such as National Vulnerability Database (NVD) and program code repositories. However, developers do not always report which commits perform bug-fixes, and this makes it hard for the users of OSS components to assess the need of upgrading to a newly released versions.

In this paper, we describe a simple approach using machine learning and natural language processing techniques to analyze source code repositories and automatically spot security-relevant commits.

We first present an introduction to the main concepts underlying machine learning and software security engineering. Then, we collect the dataset with different security fixes and train two independent classifiers using information on different features of commits, namely, their log messages and source code changes.

1 Introduction

Historical software data, especially bug reports and commit logs, is widely used in software engineering research, and can provide valuable information on the history and evolution of a software project [21].

Empirical methods for software security testing and security risk estimation often rely upon insights about security vulnerabilities that have been found and corrected in specific software projects in the past. Specifically, security bug-fixes can be extracted from source code repositories to understand the nature of these vulnerabilities and to develop more efficient methods for their detection and prevention in the future.

The main objective of this project is to identify the properties of various security fixes and build a classifier that can distinguish them from other types of changes that can be found in the source code repositories.

The core of the project lies on the assumption that vulnerability fix can be detected and analyzed considering the *commit message* and the set of structural elements of the source code changes (i.e. numerical data such as *total number of modified files,* as well as *number of added* and *deleted lines*).

The research on applying machine learning to source code analysis has been very active in recent years [1]. Similarly as in [1], our approach relies on applying machine learning on data available in code repositories, i.e., source code commits and their log messages, without the need of accessing public vulnerability databases. To do so, we must establish a *ground truth* dataset with the commits that fix defects, and those that do not. Then, we train machine learning models that can automatically spot security fixes.

2 Project description

2.1 Domain

The domain of the project mainly revolves around two following areas.

First, we are exposed to the empirical software engineering domain and the subfield of software security domain, namely security vulnerability analysis.

Second, as we are guided by the practical need to make models that make decisions with minimal human intervention, we use applied machine learning. We will look at the details of several most important machine learning techniques, and develop an intuition on how they work and when and where they are applicable. Specifically, we will look deeper at the application of supervised machine learning algorithms to train and build the classifier.

2.2 Objectives

The main objectives of the project are as follows:

- We first need to develop an in-depth, yet easy to follow, introduction to the main concepts underlying machine learning: What is learning? How can a machine learn? How do we quantify and process the resources needed to learn a given concept? Can we know if the learning process succeeded or failed? Then, we will look deeper at some of the subfields of machine learning such as supervised machine learning algorithms and natural language processing, as we use them in this project.
- Then, the objective of the project is to understand the vulnerability analysis process. Specifically, we will identify the properties of various security fixes and build a classifier that can separate them from other types of changes that can be found in source code repositories. By using the knowledge received in the first step, we propose machine-learning solution that audits source code repositories and automatically spots commits that fix vulnerabilities.
- To achieve this in practice, we need to define a set of open-source projects that contain security fixes and create initial training dataset with various security vulnerabilities collected from project’s open source repositories. This dataset will further be processed and used to train the classification models.
- Additionally, we must get acquainted with the skills in a wide range of technologies: *Python* programming language, *regular expressions*, *shell scripting*, *Git*, *Jupyter Notebook*, and libraries such as *Scikit-learn*, *Pandas, NumPy* and *matplotlib.*
- Lastly, evaluate the effectiveness of our approach in several aspects. To assess performance and improve results, we need to experiment with different machine-learning models, config-

urations and pre-processing methods and observe some of the most efficient combinations of these factors for the task based on the predictive performance and training time.

2.3 Constraints

The primary constraint of the project is the set of features we are able to collect. As we are limited in time and resources, it will be hard to collect some of the high-level features of commits. For example, certain features such as data extracted from the source code might be quite complex to fully implement during the project. Our solution is instead dependent on metadata to identify security-relevant OSS commits. This also imposes some additional restrictions on the design, namely that we should use some specific data collection and pre-processing strategies in the feature engineering process.

Besides, we are limited by the fact that there is a lack of databases with a considerable amount of open-source software security fixes, so we are restricted by the data collection opportunities and are required to present our own methods to collect the data.

Furthermore, considering a large number of samples in the dataset and difficulties in understanding the code structure or when the source code did not reflect the message, we may not perform a manual review of every single commit we include in our ground truth training dataset.

Last but not least, due to time constraints, we will not validate the generalizability of our system by deploying it on a large number of projects.

3 Background

3.1 Scientific Background

3.1.1 Software Security Engineering

3.1.1.1 Security Vulnerabilities

Software development is complex. Added problem complexity, design complexity, or program complexity increases the difficulty that a programmer encounters in the design and coding of the software system [12]. Errors and faults are being introduced in many stages of the software life-cycle [12]. The consequences of a class of system failures, commonly known as software vulnerabilities can violate security policies and can cause the loss of information, and reduce the value or usefulness of the software [12].

There is no single accepted definition of the term "software vulnerability", and hence it is difficult to objectively measure the features of vulnerabilities, or come up with any general definition [12]. For instance, Shirey [16] defined software vulnerabilities as "defects or weaknesses in system design, implement or operation management that can be used to break through security policies".

An in-depth understanding of the nature of vulnerabilities, and the methods that can be used to eliminate them or prevent them can be achieved by the application of learning, testing, and statistical tools on a representative collection of software vulnerabilities.

Finally, defining the nature and different characteristics of open-source software security problems can help both security research and software engineering teams and lead to improvements in the design and development of more secure software.

3.1.1.2 Empirical Software Engineering

Software testing will remain one of the best methods to ensure software dependability in the future [18]. Empirical studies as ours are essential to software testing research to understand, compare and improve new software testing techniques.

Moreover, in the past few years, there has been a growing interest and awareness in the empirical software engineering area of the importance of replicating studies [17]. Most researchers accept that no single study on a technology should be considered definitive because there are too many uncontrollable sources of variation exist, from one environment to another, and it is hard to extend them to all software development environments [17]. The goal is to build a unified, "empirically based body of knowledge" that takes into consideration the benefits and costs of various techniques to support the engineering of software [17].

In software engineering, this replication process enables us to build a body of knowledge about different basic principles of software development [17].

3.1.2 Machine Learning

The primary subject of this project is applied automated learning, or, as we will more often call it, Machine Learning (ML). That is, we want to program computers so that they can learn from input available to them [7]. Learning is the process of "converting experience into expertise or knowledge" [7]. "The input to a learning algorithm is training data, representing experience, and the output is some knowledge, which usually takes the form of another computer program that can perform some task" [7]. By using learning algorithms to data, it can learn from data to make informed decisions.

We can present the following more general definition by Samuel [19]: "Machine learning as the field of study that gives computers the ability to learn without being explicitly learned".

Learning problems can be categorized into two main types: *supervised* learning, that involves modeling the relationship between measured features of data and some label associated with the data and *unsupervised* learning, that involves modeling the features of a dataset without reference to any label, in which the data comes with additional attributes that we want to predict. [2]. In this project, we will be focusing on supervised machine learning algorithms.

3.1.2.1 Supervised Learning

In supervised learning, there are two problems:

- **Classification**: samples belong to two or more classes and we want to learn from labeled data to predict the class of unlabeled data [2]. Classification predictive modeling is the task of approximating a mapping function f from input variables X to discrete output variables y [2]. An example of a classification problem would be spam detection in email messages, in which the goal is to assign each input vector to one of a finite number of discrete categories (i.e. spam or non-spam). Another way to think of classification is as a discrete (as opposed to continuous) form of supervised learning where one has a limited number of categories and for each of the observation provided, one is trying to label them with the correct class [2]. Fig. 1 represents an example of a simple classification task, in which we are given a set of labeled points and want to use these to classify some unlabeled points.
- **Regression**: if the desired output consists of one or more continuous variables, then the task is called regression [2]. An

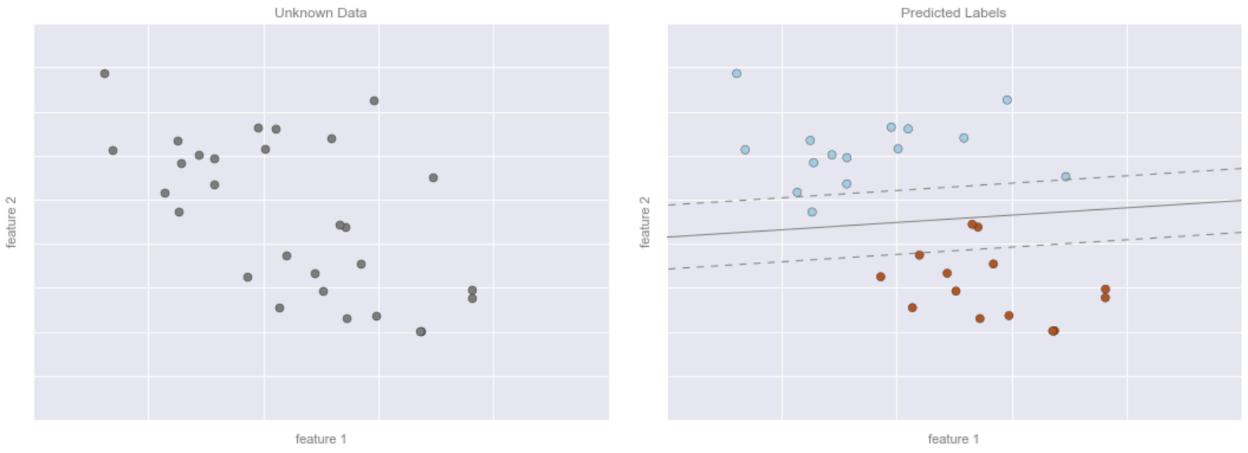


Fig. 1: Applying a classification model to data. Figure source: [14]

example of a regression problem would be the prediction of the price of rent prices as a function of time and location.

3.1.2.2 Natural Language Processing (NLP)

Natural Language Processing (NLP) is a subfield of Artificial Intelligence that is focused on enabling computers to interact with human languages, specifically, how to program computers to process and analyze large sets of natural language data [13].

Despite language being relatively easy for humans to learn, some specific tasks related to natural language processing come with their own challenges, as the ambiguity of language is what makes natural language processing hard for machines [13].

By using natural language processing techniques, developers can organize and process large amounts of data to perform tasks such as sentiment analysis, speech recognition, relationship extraction, among others.

3.1.3 Open Source Software

Open Source Software (OSS) revolves around the concept of free access and distribution of an end product, usually software or a program. Recently, OSS has become quite widespread across a variety of different uses.

The key difference between OSS and proprietary software is the availability of the source code and licensing type. The source code must be made available so that it can be inspected, modified and enhanced through multiple insights and viewpoints. As copyright material, software must be licensed. Hence, OSS is released under a license that has been certified to meet the criteria of the *Open Source Definition* [23].

3.1.4 Mining Software Repositories

The mining software repositories (MSR) field analyzes the rich data available in software repositories, such as version control repositories, issue tracking systems, etc. to uncover interesting information about software systems, projects and software engineering [11].

Many projects in the mining software repositories rely on software process data retrieved from bug tracking databases and commit logs of version control systems.

3.1.5 Version Control System

Version Control System (VCS) is a system that records changes to a file or set of files over time so that user can recall specific versions later. It is mainly designed to maintain software source code as the files being version controlled, though in reality it can be used with any type of file [9].

VCS allows to revert selected files back to a previous state, revert the entire project back, compare changes over time, see last modifications that might be causing a problem, check the author introduced an issue etc [9].

3.1.6 Common Vulnerability and Exposure (CVE) and National Vulnerability Database (NVD)

In this paper, we analyze *Common Vulnerabilities and Exposures* (CVE) list. These resource is widely used in Vulnerability Management Systems (VMSs) to check for known vulnerabilities in software products [8].

Common Vulnerability Enumeration (CVE) is a list, a free dictionary of vulnerabilities that provides a single reporting framework for known vulnerabilities not just for open source, but also for proprietary software [8].

3.2 Technical Background

3.2.1 Programming Language

Python is a powerful high-level programming language for general-purpose programming, comparable to Java, Ruby or Scheme [27]..

While there are several reasons for choosing Python as a programming language for this project, one of the main factors was Python's selection of machine learning-specific libraries and frameworks that facilitates the development process. Python's elegant and clean syntax allows easier readability for rapid testing of complex algorithms as well as focus on problem-solving. In addition, Python is an open-source programming language and is supported by a range of resources and high-quality documentation which is also applicable to the libraries written in Python that we use in the project such as Scikit-learn, pandas, NumPy and matplotlib.

3.2.2 Scikit-learn

The *Scikit-learn* project provides an open source machine learning library for the Python programming language. The ambition of the project is to provide efficient, well-established and yet simple machine learning tools within a programming environment that is accessible to non-experts and reusable in various scientific areas and contexts [5]. It also includes learning algorithms, model evaluation and selection tools, as well as data preprocessing procedures [5].

Scikit-learn is a library, i.e. a collection of modules containing functions that users can import into Python programs [5]. Using *scikit-learn* therefore requires basic Python programming knowledge. The interactive use is possible in the Python interactive interpreter, and its enhanced replacement IPython (now known as the Jupyter Notebook) [5].

The library has been designed around NumPy and SciPy libraries that must be installed before you can use scikit-learn [5]. This set of packages includes, as described in [5]:

- **NumPy**: Base n-dimensional array package with a handy numeric array datatype and fast array computing methods.
- **SciPy**: Library for scientific computing that extends NumPy further with common numerical operations.
- **Pandas**: Easy-to-use data structures and data analysis tools.
- **Matplotlib**: 2D/3D plotting library that can be used in the Python and IPython shells and the Jupyter Notebook.

The scikit-learn API is designed with the following principles in mind, as outlined in the scikit-learn API paper [5]:

- **Consistency**: All objects share a common interface drawn from a limited set of methods, with consistent documentation.
- **Inspection**: All specified parameter values are exposed as public attributes.
- **Limited object hierarchy**: Only algorithms are represented by Python classes; datasets are represented in standard formats (NumPy arrays, Pandas DataFrames, SciPy sparse matrices) and parameter names use standard Python strings.
- **Composition**: Many machine learning tasks can be expressed as sequences of more fundamental algorithms, and Scikit-Learn makes use of this wherever possible.
- **Sensible defaults**: When models require user-specified parameters, the library defines an appropriate default value.

The vision of the library is a level of robustness and support required for use in production systems. This means a deep focus is put on concerns such as ease of use, code quality, collaboration, documentation and performance.

3.2.2.1 Data Representation in scikit-learn

Machine learning is about learning some properties of data and creating models; for that reason, we need to understand how data can be represented in order to be understood by the computer. The best way to think about data within the scope of scikit-learn is in terms of tables of data [14].

Before we proceed, we need to define a dataset. Dataset is a *dictionary-like* object that holds the data and some specific metadata about the data [2]. ”A basic table is a two-dimensional grid in which the rows represent individual elements of the dataset, and the columns represent quantities related to each of these elements” [14]. In general, we call the rows of the matrix *samples*, and the numbers of rows *n_samples* [14]. Likewise, we call the columns of the matrix *features*, and the number of columns *n_features*, that represent a specific quantitative piece of information describing each sample [14]. The samples (i.e., rows) always refer to the ”individual objects described by the dataset”, whereas the features (i.e., columns) refer to ”distinct observation that describe each sample in a quantitative manner” [14].

The layout of table of features represents the data as a ”two-dimensional numerical array or matrix”, which we call the *features matrix*, which is often stored in a variable named X [14]. The features matrix is assumed to be two-dimensional, with shape *[n_samples, n_features],* and is contained in a Pandas DataFrame object [14].

In addition to the feature matrix, we work with a label array y, which we call *target array* or *class*. The target array is one-dimensional, with length *n_samples* and is contained in Pandas Series [14]. The target array is different from from features as it is the quantity we want to predict from the data [14].

To sum up, the expected layout of features and target values can be visualized as the following:

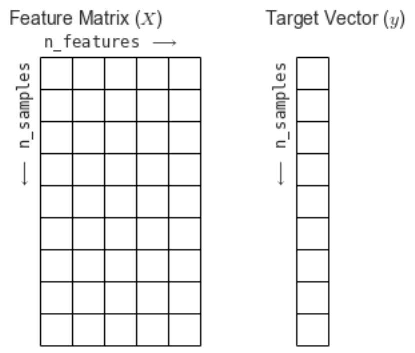


Fig. 2: Representation of features and target values. Figure source: [15]

There are four main requirements for data in scikit-learn:

1) Features and response (target array) are separate objects.
2) Features and response should be homogeneous (i.e numeric).
3) Features and response should be NumPy arrays.
4) Features and response should have specific shape (i.e. the data is always a 2D array, with the shape (*n_samples,n_features))*

3.2.3 Git

Git is an open source distributed version control system created to handle both small and large projects with speed and efficiency [9]. It is primarily used for source-code management in software development, but it can be used to keep track of changes in any set of files [9]. As a distributed revision-control system, it is focused on ”speed, data integrity, and support for distributed, non-linear workflows” [9].

4 BSPro - Automatic inference of 'hidden' security fixes in source code repositories

4.1 Requirements

4.1.1 Non-functional Requirements

- *Simple approach.* The ambition of this project is to translate the problem of classifying high-level properties of commits such as code changes to the problem of classifying metadata of commits (i.e. commit messages and numerical changes of code) using machine learning and natural language processing. Our machine learning models should not require sophisticated code analysis and should be functional within any programming language. We should use simple models architecture that require small computational effort.
- *Extensibility and reusability.* Extensibility and reusability is important for the code we use in the project. At this moment, we have the possibility to create datasets with a particular set of features, such as commit logs and some numerical information about commit(s). But there is much more space left for experimenting with a different sets of features. For example, based on this functionality, it will be possible to obtain a change log, that contains the information such as source code (names of classes, functions, variables etc), date and time, changed files etc. for each commit or committed file in the future. It is therefore important to deliver the code which is easy to understand, extend, modify and reuse.

4.1.2 Functional Requirements

- *Predictive performance and training time.* The primary goal of the project is to a build classifier that has relatively high predictive performance and acceptable training time.
- *Dataset collection.* We should collect training dataset *(ground truth)* with various security fixes collected from the open source repositories. This dataset will further be processed and used to train the classification models. As obtaining bug-fix commit(s) requires significant manual effort, we need to automate this process by creating a set of reusable and well-documented Python scripts that collect the data for training and testing.
- *Data requirements.* To enhance the classifier, we need to feed the machine learning models with a range of diverse data. For the most comprehensive learning experience, we must define data requirements for this project. We should provide diverse training data integrated from multiple sources and concerning various contextual case scenarios, collected across multiple time frames, to make algorithmic assessments more generic in a real-world use case.
- *Flexible data collection.* It is not only essential to have a dataset for the specific problem, but it is also important to have a possibility of trying out different conditions and testing different empirical methods on the dataset. Thus, we must ensure that our dataset generation progress is as flexible as possible so that we can use it in a wide range of contexts.
- *Evaluation and improvement.* We evaluate the effectiveness of our classifier in several aspects. To assess performance and improve overall results, we must experiment with different machine-learning models, model-specific optimizations (hyper-parameter tuning) as well as pre-processing methods, and observe some of the most efficient tradeoffs of these factors for the problem based on the predictive performance and training time.

4.2 Design

4.2.1 Dataset

This section describes the methodology used to obtain real security fixes, from the mining process to the samples evaluation and approval. The main goal of this approach is the identification and extraction of real security fixes. The research for new methods to retrieve data in this field is really important due to the lack of datasets with a considerable amount of security fixes to support static analysis studies [1]. We present in detail our approach for retrieving, processing, parsing and linking the data.

The preprocessing is the first step in building a machine learning model. At this step, we acquire and format the data for the future usage. We consider the following tasks to be a part of 'preprocessing' stage: extracting features from raw data, formatting the data, removal of highly correlated features, manual review and removal of outliers, standartization and normalization of the range of feature data, and creation of a test dataset.

For our study, we created a series of Python scripts that collect the data for our labeled dataset of known open-source vulnerabilities. That data is then used to train *Log Message Classifier* and *Patch Classifier* that will be able to separate security-relevant commits using machine learning algorithms. For *Log Message Classifier*, we take commit message as the only feature, whereas for *Patch Classifier* we take source code changes introduced by commit(s) such as total number of modified files, as well as number of added and deleted lines.

As obtaining bug-fix commit(s) requires significant manual effort, we needed to find some automated methods that could minimize the cost of this process. The first step was the development of the tools that can retrieve the data about real security fixes in the open-source repositories in order to create a labeled dataset.

Many projects in the mining software repositories field rely on data gathered from commit log files of version control systems [11]. Initially, our main idea to retrieve the right data resolved around using Github API, but due to the limitations represented by the limits of API calls and restricted search syntax (i.e. no regexp support), the solution of cloning the project's repositories and performing a local search using Git seemed to be more accessible. We collect data from GitHub to track source code changes and fixes from the projects. We collected data mostly from web applications of a total of 135 most active projects based on *openhub.net*, online community and public directory of free and open source software [26], such as Apache Tomcat, Spring Framework, Jenkins written mostly in Java, Python, Ruby and Javascript. To ensure that a project has a relatively high number of bug-fixes to study, we performed a manual search of a number of reported vulnerabilities on *cvedetails.com*, CVE security vulnerability information source [25], and filtered out the projects with a relatively low number of security vulnerabilities due to space constraints. We also tried to obtain the data with *cve-search* utility [10], a tool that imports *CVE* and *CPE* into a MongoDB and simplifies search and processing; it could be relatively easy to filter out security reports with references to Github with a simple regular expression `'https://github.com/.*/.*/commit/[0-9a-f]{40}'` by piping the output to the `grep` unix command to filter out all the links to commits and get lines that match that pattern. However, due to several factors such as the absence of regular expression search and quite complex pre-processing strategies we needed to employ in this case, we decided to obtain the initial data with Git.

Considering the large amount of data of such large-scale projects, the following step was the identification of the specific properties that are relevant to the security fixes. That is, we needed to extract commits that are clearly security fixes based on some patterns that are specifically related to vulnerabilities. Initially, our main focus was the *Top 10 OWASP 2018* [6] and other vulnerability taxonomies, but then more patterns were added and there is still place for much more. For each bug type, we created a regular expression pattern satisfying a range of possible combinations of words and acronyms that may describe a potential security fix. These words were mined on commit messages in order to find possible samples to test cases. The search of commits that may be included in our dataset is performed with several Python scripts responsible for matching our patterns with commit messages.

First, we used *common_vuln_commits* script to identify and extract the data about real security fixes. This script augments the dataset with the instances of the *positive* class (security-relevant) commits. The *common_vuln_commits* script performs a search in project's repositories using regular expression patterns that we created from for the *Top 10 OWASP 2018* [6] that are located in *keyword_mappings* and then we extended the tool to others. *Keyword_mappings* itself is represented as a dictionary-like object containing regular expression patterns. With this script, we managed to collect 4017 samples. Besides words related to each pattern

```
'''
This script augments the dataset with instances of
    the 'positive' class (security-relevant) commits
    .
The script performs a search over project's
    repositories using regular expression patterns
    that we created from for the Top 10 OWASP 2018
    that are located in 'keyword_mappings'.

'''
import subprocess
import os
from etc.keyword_mappings import vuln_types as
    mappings

def common_vuln_commits(command):
    for k, j in mappings.items():
        vuln_list = list(j)
        output_list = []
        for x in vuln_list:
            output_list.append("'" + x + "'")

        for e in output_list:
            subprocess.call(command % (i, i, e),
                shell='True')

#Listing the repositories cloned in the 'projects'
    directory.
path = './projects/'
projects = [f for f in os.listdir(path) if os.path.
    isdir(os.path.join(path, f))]


for index, i in enumerate(projects):
    common_vuln_commits('git -C ./projects/%s log --
        pretty=format:%s,\'"%%H","%%s"\' | grep -E -
        w %s >> common_vuln_commits.csv')
```

Listing 1: *common_vuln_commits* script

(e.g *injection, DoS, and XSS etc*), we added the miscellaneous patterns to identify additional cases where the message contains indications of a generic security fix. (Please refer to *Figure 3* for part of the rule set).

Fig. 3: Examples of regular expressions in *keywords_mappings* used to filter out security-related commits

Rule	Regular Expression
Code Execution	*'(code—expression—statement) execution,' 'execute ([a-zA-Z]+ *){0,3}[ ]{0,} (expression(s){0,1}—statement(s){0,1}—code)'*
Overflow	*'(buffer—integer) overflow'*
DoS	*'denial of service', 'dos'*
XSS	*'xss', 'cross[\-]{0,}site scripting', '(js—javascript) injection'*
CVE	*"(?i)CVE-\d{4}-\d{4,7}"*

Then, using another regular expression in *cve_id_commits* script we were able to detect the commits that introduce security fixes and are given the *CVE identifier*. CVE (also referred as 'CVE Entries', 'CVE IDs', 'CVE numbers', and 'CVEs') are unique, common identifiers for publicly known security vulnerabilities [8]. With this pattern, we have collected 908 samples.

```
'''
This script augments the dataset with instances of
    the 'positive' class (security-relevant) commits
    .
The script performs a search  over project's
    repositories using regular expression and is
    able to detect the commits with CVE IDs.

'''
import subprocess
import os


def cve_id_commits():
    #Listing the repositories cloned in the '
        projects' directory.
    path = './projects/'
    projects = [f for f in os.listdir(path) if os.
        path.isdir(os.path.join(path, f))]

    #Iterating the list of the projects with 'git
        log' and collecting commits logs matching '
        CVE-ID' pattern.
    for x in projects:
        command = 'git -C ./projects/%s log --pretty
            =format:%s,\'"%%H","%%s"\' | grep  -E
            "(?i)CVE-\d{4}-\d{4,7}" >>
            cve_id_commits.csv'
        subprocess.call(command % (x, x), shell='
            True')

cve_id_commits()
```

Listing 2: *cve_id_commits* script

After this, we created a script that augments the dataset with instances of the *negative* class (non-security-relevant) commits. Specifically, we used the same approach as in [1]; for each *positive* sample p from repository R, we took k random commits $n_1, ..., n_k$ (by date descending) from R and, under the assumption that security-relevant commits are relatively infrequent compared to other types of commits, we treated these as *negative* examples. Using this method, we have collected 23990 commits. Then, we created *testing_dataset* script that creates the dataset with instances of the

```
'''
This script augments the dataset with instances of
    the 'negative' class (non-security-relevant)
    commits.

'''

import subprocess
import os


def random_commits():
    #Listing the repositories cloned in the '
        projects' directory.
    path = './projects/'
    projects = [f for f in os.listdir(path) if os.
        path.isdir(os.path.join(path, f))]


    #Iterating the list of the projects with 'git
        log' and collecting commits logs for last
        185 commits in each of the repository.
    for x in projects:
        command = 'git -C ./projects/%s log -n 185
            --pretty=format:%s,\'"%%H","%%s"\' >>
            random_commits.csv'
        subprocess.call(command % (x, x), shell='
            True')

random_commits()
```

Listing 3: *random_commits* script

```
'''
This script creates the dataset with instances of
    the testing data.
We log the data about n last commits in each
    repository available in the 'projects' directory
    .

'''

import subprocess
import os

def testing_dataset():

    #Listing the repositories cloned in the '
        projects' directory.
    path = './projects/'
    projects = [f for f in os.listdir(path) if os.
        path.isdir(os.path.join(path, f))]

    #Iterating the list of the projects with 'git
        log' and collecting commits logs for each of
         the repository.
    for x in projects:
        command = 'git -C ./projects/%s log -n 1000
            --pretty=format:%s,\'"%%H","%%s"\' | sed
             1,185d >> testing_dataset.csv'
        subprocess.call(command % (x, x), shell='
            True')

testing_dataset()
```

Listing 4: *testing_dataset* script

testing commits. We can log the data about n last commits which are not used in the training dataset. Version Control Systems such as Git allow to get a change log file of all revisions/files by using one command [22]. Thus, we can simply export the change log data into a text file.

The general methodology behind constructing these scanners described above was centered around extracting data with `git log` command that shows commit history logs. Then, we use commit formatting options that are included in `git log`. Namely, the command takes `--pretty` option that prints the contents of the commit logs in the needed format. Then, we filter out the commits from repositories matching a specific regex pattern with `grep` unix command. After retrieving the logs, the raw data has to be parsed into the dataset. To do this, every time the tool identified a pattern, the commit log is written in the *Comma-separated values* (CSV) formatted document as an entry with the fields *project_name*, *hash* and *commit_message*, where *project_name* and *hash* are the directory of the repository in the local machine, unique hash identifier and commit message of the commit respectively. This is a hard procedure as some of the bug tracking data may require some additional parsing effort.

Lastly, we created *patch* script that creates a dataset for *Patch Classifier*. This script takes *the hash identifiers* and *directory* name of previously generated commits and augments the data by collecting numerical data about each source code change of each commit such *as total number of modified files*, as well as *number of added and deleted lines*. Since it is not clear whether a given commit in a repository is a security fix based merely on numerical data, we use this script on the existing labeled dataset and extract the features only after we know what commits are represented as security or non-security fixes. Differently from other scripts, we use `git show` command with `--stat` option to obtain the data.

After saving the initial data, a manual review is performed on two different types of data: 1) *Commit message* (to validate if the message really reflects indications of security fix because some of the commits recognized using regular expression patterns can lead to false-positives) and 2) *Source code* (to identify if the changes in source code representing the possible security fix exist or not). Normally, due to the challenges that come along validating the source code, one should check specific literature, vulnerabilities websites and many other sources [3]. Usually, the process involves giving a look at the source code and trying to highlight a few functions or problems that could represent the bug and then make a search on the internet based on the language, frameworks etc [3].

Besides the validation, to avoid including any possible outliers in the dataset (empty, repetitive, significantly large or otherwise invalid commits), we performed manual case-by-case analysis of such commits. In some cases, we were able to manually format the entries according to our dataset requirements and added them in the final dataset. Normally, when the data points were too distant and differed greatly from the other values in the dataset, the latter test cases were pointed out as non-viable and thus removed from the dataset. After this manual review, we were left with 28745 samples in total (please refer to *Figure 4* where we present a fraction of samples from the final output).

As we deal with a binary classification problem, there are only two classes that we want to predict from the data, namely 1 or 0. We then manually label filtered samples such that $1's$ would correspond to the *positive class* (i.e security commits) and $0's$ would correspond to the *negative class* (i.e non-security commits). We call this column *labels*, *target array* or *class* and it will be represented as one-dimensional array, with the length *n_samples*.

Due to time constraints, we did not perform a manual review of every single commit, so it can be the case that some of the false positives were not caught. Hence, this can have an effect on the

```
'''
This script takes the hashes and directories of
    previously generated commits in 'full_output.csv
    ' and augments the data by collecting
    statistical data about each commit such as total
     number of modified files, as well as number of
    added and deleted lines.

'''

import subprocess
import csv


def patch():
    hash_ = []
    directory = []


    with open("full_output.csv", 'r') as f:
        reader = csv.reader(f, delimiter=',')
        for row in reader:
            hash_.append(row[2])

    with open("full_output.csv", 'r') as f:
        reader = csv.reader(f, delimiter=',')
        for row in reader:
            directory.append(row[1])

    hash_.pop(0)
    directory.pop(0)


#Iterating specific commits collected previously and
     collecting numerical statistics with git show
    --stat corresponding to each commit.
#Output only the last line of the --stat with tail -
    n1

    for h, d in zip(hash_, directory):
        command = 'git -C ./projects/%s show %s --
            stat | tail -n1 >> patch_output.csv'
        subprocess.call(command % (d, h), shell='
            True')

patch()
```

Listing 5: *patch* script

perceived objectivity level for our *ground truth* dataset. To sum it up, we can see the expected process of obtaining dataset for the models in the flowchart Fig. 5.

4.3 Production

4.3.1 Commit Classification

In this section we will describe in detail approach we used in commit classification problem and the process of implementation of two different independent classification models. Our approach to commit classification revolves around the following two key ideas.

First, we classify commits based on their commit log message. Because we cannot feed predictive models with text data, we use conventional machine-learning algorithms and *natural language processing* (NLP) methods to turn the text into vectors of numerical values suitable for statistical analysis. The resulting object is a sparse two-dimensional matrix to which standard NLP methods can be applied.

Second, we use a number of statistical metadata measures, extracted from the source code changes stats such as *total number of modified files, number of added and deleted lines*. By the hypothesis of our study, the commits that implement fixes to vulnerabilities should generally be more compact compared to the ones that introduce some functionality or updates to the software, thus containing smaller quantities than those that are not security-relevant.

To achieve this, we construct and train two different *independent models*, each considering a different aspect of the commit (the log message and the changes in the source code). In the following, we describe how we implemented the two models (*msg* and *patch*)

(A) **Log Message Classifier (msg)**. In this model, we classify commits based on their commit log message.

In order to perform machine learning on log messages, we first need to turn the text into numerical feature vectors. The most intuitive way to do this is to use a *bags of words representation* . The bags of words representation implies that *n_features* will be represented as the number of distinct words in the corpus [2]. In *Scikit-learn*, pre-filtering (text preprocessing, tokenizing, stemming and stopword removal) are all included in CountVectorizer, which builds a dictionary of features and transforms documents to feature vectors s [2].

```
>>> import pandas as pd
>>> from sklearn.svm import LinearSVC
>>> from sklearn.feature_extraction.text import
    CountVectorizer
>>> from sklearn.feature_extraction.text import
    TfidfTransformer

#import the dataset
>>> df = pd.read_csv('full_output_features.csv')

#name the columns
>>> df.columns = ['outcome_class','project_name','
    hash','commit_message','files_changed','
    insertions','deletions']
>>> df.head()

#select the target array
>>> Y_labels = outcome_class

#use CountVectorizer()
>>> count_vect = CountVectorizer()
>>> Xtrain_counts = count_vect.fit_transform(df.
    commit_message)

#use TfidfTransformer()
>>> tfidf_transformer = TfidfTransformer()
>>> X_commit_log = tfidf_transformer.fit_transform(
    Xtrain_counts)
```

Listing 6: Extracting features from text

Once fitted, the vectorizer will build a dictionary of feature indices [2]. In our case, we get a number of 7670 features. The index value of a word in the vocabulary is linked to its frequency in the whole training corpus [2].

While occurrence count is a good start, there is an issue: longer texts will have higher average count values than shorter texts. To avoid these discrepancies it suffices to divide the number of occurrences of each word in a document by the total number of words in the document: these new features are called tf to Term Frequencies [2]. Another refinement on top of tf is to downscale weights for words that occur very frequently and are therefore less informative than those that occur less frequently [2]. This downscaling is called tf-idf for 'Term Frequency times Inverse Document Frequency' [2]. Both tf and tf-idf can be computed as follows using TfidfTransformer: We then use the fit() method to fit our estimator to the data

	outcome_class	project_name	hash	commit_message	files_changed	insertions	deletions
0	1	ant	7bc745a289cf68cb2eba647bbfba9e9ec06eb771	post-process generated javadocs as workaround ...	6	171	3
1	1	ant	08284bc7aa7d066544b30974231240c9b73597eb	[CVE-2012-2098] merge bzip2 edge case improvem...	4	1310	573
2	1	ambari-logsearch	1811d0bad0b5e3691036c027cef24e289640e704	AMBARI-24431. Infra Manager / Log Search: Fix ...	1	1	1
3	1	ambari-logsearch	37db2229747c0e58e95e14a787895af7308f90a2	AMBARI-24422. Log Feeder: upgrade guava versio...	3	3	3
4	1	mod_perl	82b8d8bfe88a8878e1efb793c69548e1eb9eed55	Fix t/perl/hash_attack.t to work with Perl 5.1...	2	29	9
5	1	maven	2b336ff150b12aeae9bd0d2a61f1e8d02504492f	[MNG-6312] Update Maven Wagon dependency o Up...	1	1	1

Fig. 4: Example of the *full_output* dataset.

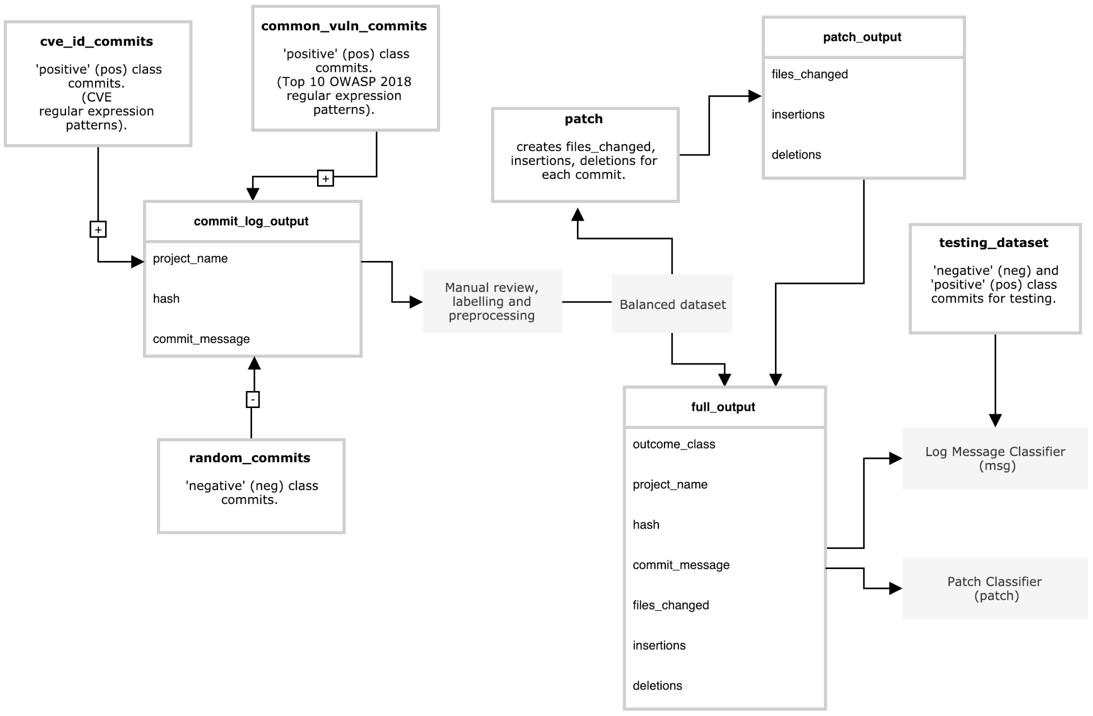

cve_id_commits
'positive' (pos) class commits. (CVE regular expression patterns).
common_vuln_commits
'positive' (pos) class commits. (Top 10 OWASP 2018 regular expression patterns).
patch
creates files_changed, insertions, deletions for each commit.
patch_output
files_changed
insertions
deletions
commit_log_output
project_name
hash
commit_message
Manual review, labelling and preprocessing
Balanced dataset
testing_dataset
'negative' (neg) and 'positive' (pos) class commits for testing.
random_commits
'negative' (neg) class commits.
full_output
outcome_class
project_name
hash
commit_message
files_changed
insertions
deletions
Log Message Classifier (msg)
Patch Classifier (patch)
+
+
-

Fig. 5: Flowchart representing the process of obtaining data

and secondly the `transform()` method to transform our count-matrix to a tf-idf representation [2].

Now that we have our features, we can train a classifier to try to predict the remaining labels. We will use a machine learning algorithm called *Support Vector Machines* (SVMs), a set of supervised learning methods that are commonly used for classification [2]. Specifically, we will use linear SVM model which provides a nice baseline for this task.

```
>>> from sklearn.svm import LinearSVC
>>> from sklearn.model_selection import
    train_test_split

#split x and y into training and testing sets
>>> Xtrain, Xtest, ytrain, ytest = train_test_split(
    X_commit_log, Y_labels, random_state=0)

#train a linear SVM model on the training set
>>> clf_commit_log = LinearSVC().fit(Xtrain, ytrain)
```

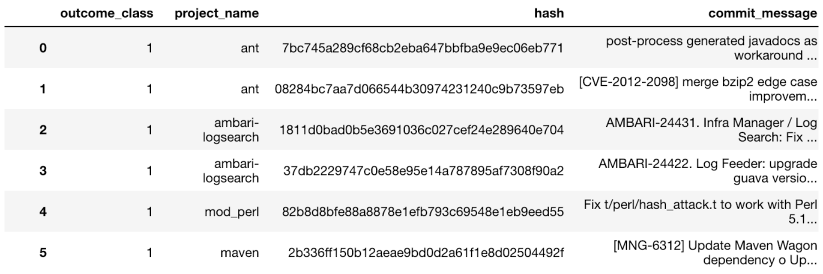
Listing 7: Training of the *msg* classifier

We evaluate the efficiency of our automatic commit identification system in several aspects. First, we split the dataset into training and testing sets to evaluate the predictiveness of the stacking algorithm. We compare it with several individual classifiers, including the linear SVM described in [1].

A common practice in machine learning is to evaluate the model on data it has not seen before is by splitting a data set into *training set*, on which we learn some properties and *testing set*, on which we test the learned properties. This could be done by hand, but it is more convenient to use the `train_test_split` utility function provided by *Scikit-learn*.

In the above example-code, we first import our dataset as a DataFrame object and assign variable `Y_labels` as our target array. Then, we fit the model to the data by calling the `fit()` method on the model of instance `clf_commit_log`. During this step, the `clf_commit_log` object is learning the relationship between `Xtrain` and `ytrain`.

We pass the feature matrix for the testing set `Xtest` to the the `predict()` method for the fitted model. Now that we have our class prediction for every observation, we can use the `accuracy_score` function from `metrics` module to see the fraction of predicted labels that match their true value:



	outcome_class	project_name	hash	commit_message
0	1	ant	7bc745a289cf68cb2eba647bbfba9e9ec06eb771	post-process generated javadocs as workaround ...
1	1	ant	08284bc7aa7d066544b30974231240c9b73597eb	[CVE-2012-2098] merge bzip2 edge case improvem...
2	1	ambari-logsearch	1811d0bad0b5e3691036c027cef24e289640e704	AMBARI-24431. Infra Manager / Log Search: Fix ...
3	1	ambari-logsearch	37db2229747c0e58e95e14a787895af7308f90a2	AMBARI-24422. Log Feeder: upgrade guava versio...
4	1	mod_perl	82b8d8bfe88a8878e1efb793c69548e1eb9eed55	Fix t/perl/hash_attack.t to work with Perl 5.1...
5	1	maven	2b336ff150b12aeae9bd0d2a61f1e8d02504492f	[MNG-6312] Update Maven Wagon dependency o Up...

Fig. 6: Example of the part of the dataset used in *Log Message Classifier (msg)*

```
>>> from sklearn.metrics import accuracy_score

#make class predictions for the testing set
>>> y_commit_log_model = clf_commit_log.predict(
     Xtest)

#calculate accuracy
>>> accuracy_score(ytest, y_commit_log_model)
0.9846406902310048
```

Listing 8: Checking the accuracy of the *msg* model

As we can see, the function shows that approximately 98% predictions in `y_commit_log_model` were correct.

4.3.1.1 *Example of msg classifier commit classification in practice*

Now, we can predict labels for unknown data using the `predict()` method on the model. In our case, we import testing dataset that we have created with *testing_dataset* script previously. Since it is necessary that our unknown data has the same preprocessing type as our known data, we call the `transform()` methods on our testing dataset.

```
#import the testing dataset
>>> df_test = pd.read_csv('testing_ds.csv')

#transform the data
>>> Xtest_counts = count_vect.transform(df_test.
     commit_message)
>>> Xtest_commit_log = tfidf_transformer.transform(
     Xtest_counts)

#predict the data
>>> Y_predict = clf_commit_log.predict(
     Xtest_commit_log)

#create 'predicted class' column with predicted
     classes.
>>> df_test["predicted class"]= Y_predict
>>> df_test[150:500]
```

Listing 9: Testing the *msg* model on unknown data

We get the following results, where *predicted class* column represents the class predicted by the model:

(B) **Patch Classifier (patch)**. Along with our hypothesis, our approach in this model stems from the intuition that the changes of modified lines, deleted and added lines in the in source code can convey information that can be used for classification. Based on the combination of these three features, we treat those as numerical measures to predict the label of the commits (please refer to Figure 9 to see the example of the dataset we use in this model).

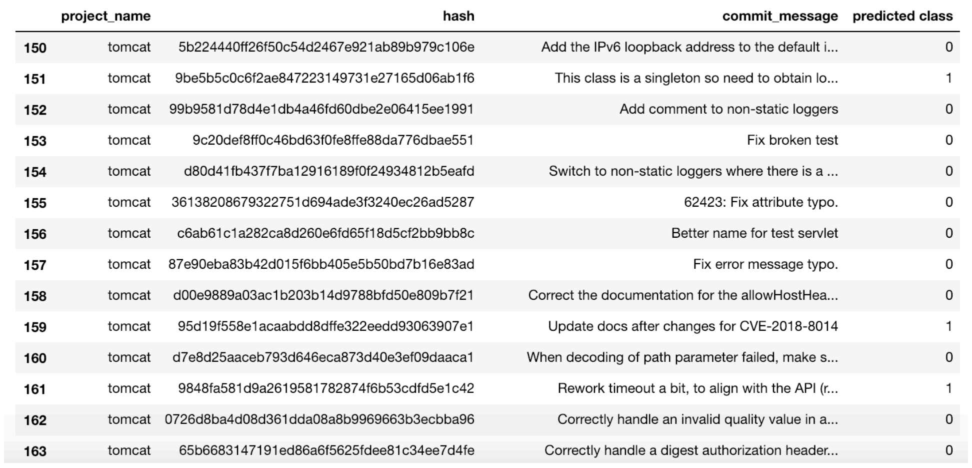
Similarly as in *msg*, for this classification problem, we use linear SVM model. In this dataset, we apply different pre-processing phase. Specifically, we apply different pre-filtering (removal of special characters, filling empty values with 0's). We then normalise our data by calling `preprocessing` function with `normalize()` method.

```
>>> import pandas as pd
>>> from sklearn.svm import LinearSVC
>>> from sklearn import preprocessing
>>> from sklearn.preprocessing import StandardScaler
>>> import numpy as np

#import the dataset
>>> df = pd.read_csv('full_output_features.csv');

#select the columns from the dataset
>>> col = ['files_changed','insertions','deletions']
>>> X_patch = df[col]

#select the target array
>>> Y_labels = df.outcome_class

#fill the empty cells with 0's
>>> X_patch = X_patch.fillna(0)

#remove unwanted charachters
>>> X_patch['files_changed'] = X_patch['
     files_changed'].str.extract('(\d+)')
>>> X_patch['insertions'] = X_patch['insertions'].
     str.extract('(\d+)')
>>> X_patch['deletions'] = X_patch['deletions'].str.
     extract('(\d+)')

#normalise the data
>>> X_patch_normalized = preprocessing.normalize(
     X_patch, norm ='l2')
```

Listing 10: Dataset preprocessing of the *patch classifier*

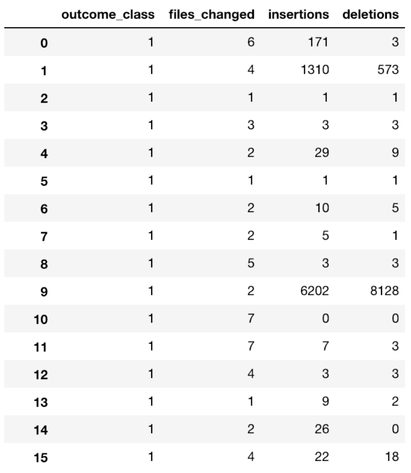
Then, we fit the model to our data by calling the `fit()` method of the model.

In the same way, using `train_split_function`, we split our data into training and testing dataset. In order to check the accuracy of the model, we check the match between predicted labels and their truth value with the `accuracy` function.



	project_name	hash	commit_message	predicted class
150	tomcat	5b224440ff26f50c54d2467e921ab89b979c106e	Add the IPv6 loopback address to the default i...	0
151	tomcat	9be5b5c0c6f2ae847223149731e27165d06ab1f6	This class is a singleton so need to obtain lo...	1
152	tomcat	99b9581d78d4e1db4a46fd60dbe2e06415ee1991	Add comment to non-static loggers	0
153	tomcat	9c20def8ff0c46bd63f0fe8ffe88da776dbae551	Fix broken test	0
154	tomcat	d80d41fb437f7ba12916189f0f24934812b5eafd	Switch to non-static loggers where there is a ...	0
155	tomcat	36138208679322751d694ade3f3240ec26ad5287	62423: Fix attribute typo.	0
156	tomcat	c6ab61c1a282ca8d260e6fd65f18d5cf2bb9bb8c	Better name for test servlet	0
157	tomcat	87e90eba83b42d015f6bb405e5b50bd7b16e83ad	Fix error message typo.	0
158	tomcat	d00e9889a03ac1b203b14d9788bfd50e809b7f21	Correct the documentation for the allowHostHea...	0
159	tomcat	95d19f558e1acaabdd8dffe322eedd93063907e1	Update docs after changes for CVE-2018-8014	1
160	tomcat	d7e8d25aaceb793d646eca873d40e3ef09daaca1	When decoding of path parameter failed, make s...	0
161	tomcat	9848fa581d9a2619581782874f6b53cdfd5e1c42	Rework timeout a bit, to align with the API (r...	1
162	tomcat	0726d8ba4d08d361dda08a8b9969663b3ecbba96	Correctly handle an invalid quality value in a...	0
163	tomcat	65b6683147191ed86a6f5625fdee81c34ee7d4fe	Correctly handle a digest authorization header...	0

Fig. 7: Predicted results for the *testing_dataset*



	outcome_class	files_changed	insertions	deletions
0	1	6	171	3
1	1	4	1310	573
2	1	1	1	1
3	1	3	3	3
4	1	2	29	9
5	1	1	1	1
6	1	2	10	5
7	1	2	5	1
8	1	5	3	3
9	1	2	6202	8128
10	1	7	0	0
11	1	7	7	3
12	1	4	3	3
13	1	1	9	2
14	1	2	26	0
15	1	4	22	18

Fig. 8: Example of the dataset we use for *Patch (patch) classifier*

```
>>> from sklearn.metrics import accuracy_score
>>> from sklearn.model_selection import
    train_test_split

#split x and y into training and testing sets
>>> Xtrain_stat, Xtest_stat, ytrain_stat, ytest_stat
     = train_test_split(X_patch_normalized, Y_labels
    , random_state=0)

#train a linear SVM model on the training set
>>> clf_patch = LinearSVC().fit(Xtrain_stat,
    ytrain_stat)

#make class predictions for the testing set
>>> y_patch_model = clf_patch.predict(Xtest_stat)

#calculate accuracy
>>> accuracy_score(ytest_stat, y_patch_model)
0.5439024390243903
```

Listing 11: Checking the accuracy of *patch* classifier

As we can see, with the *patch* classifier we get the accuracy of approximately 54%.

4.4 Assessment

4.4.1 Evaluation

Similarly as in [1], we experimented with different models (logistic regression, support vector machines, Gaussian Naive Bayes, among others), and with different combinations of parameter settings (e.g., different kernels and regularization methods, and other model-specific parameters). We observed that linear *Support Vector Machine* (SVM) performs better than the other methods in our tests, and provided both good predictive performance and acceptable training time.

In our case, we observed that the impact of choice of the model has a lesser effect on the predictive performance compared to the effect produced when we used different pre-processing strategies. When used separately, the first model produces relatively high recall. On the other hand, we observed that we could not achieve very high predictive performance in the second model, mainly due to the nature of data. To compensate for this effect, we combine the models when making predictions in order to increase the overall predictive performance. As soon as one of the models report commit as security-relevant, we flag the commit as security-relevant.

To measure predictive performance, we use the following metrics: *precision*, *recall* and *f1-score*. Here, we give short definitions:

$$precision = \frac{true\ positive}{true\ positive + false\ positive}$$

$$recall = \frac{true\ positive}{true\ positive + false\ negative}$$

$$f1 = 2 \cdot \frac{precision \text{ x } recall}{precision + recall}$$

4.4.1.1 Model Evaluation

In this section, we will look in detail at the performance of two models:

(A) **Log Message Classifier (msg)**. The `sklearn.metrics` module implements several loss, score, and utility functions to measure classification performance [2]. We use `classification_report` function from `sklearn.metrics` module to build a text report showing the main classification metrics:

```
>>> from sklearn import metrics

#show the classification report
>>> print(metrics.classification_report(ytest,
    y_commit_log_model))
```

Listing 12: Showing the main classification metrics for *msg* model

The outcome of this code is summarised in Figure 9, for *log message* classifier.

Fig. 9: Report showing the main classification metrics for *msg* model

	precision	recall	f1-score
0	0.99	0.99	0.99
1	0.97	0.95	0.96
micro avg	0.99	0.99	0.99
macro avg	0.98	0.97	0.98
weighted avg	0.99	0.99	0.99

(B) **Patch Classifier (patch)**.
Similarly as in *msg* model, we use `classification_report` functions.

```
>>> from sklearn import metrics

#show classification report
>>> print(metrics.classification_report(ytest_stat,
    y_patch_model))
```

Listing 13: Checking the accuracy of *patch* classifier

We get the following report for the *patch* model:

Fig. 10: Report showing the main classification metrics for *patch* model

	precision	recall	f1-score
0	0.56	0.44	0.49
1	0.53	0.65	0.59
micro avg	0.54	0.54	0.54
macro avg	0.55	0.54	0.54
weighted avg	0.55	0.54	0.54

As we can see, *patch* classifier on its own yields relatively low recall compared to *msg* classifier. While trying to improve the accuracy of *patch* model, we have observed that there is simply high invariance between features and class. That is, our initial hypothesis that most of the security fix are quite compact compared to other types of commits is not strong enough in all cases. We can see that the scatter plot in Figure 11 gives us the ability to simultaneously observe three different dimensions of the data: the *(x, y)* location of each point corresponds to the *number of insertions* and *number of deletions*, the size of the point is related to the *number of files changed*, and the color is related to the *labels*.

4.4.2 Further Analysis

In our further analysis, we used cross-validation technique to validate the models as it results in estimates that generally have a lower bias than other methods thus shows a more reliable estimate of the techniques we applied to the dataset.

With cross-validation, we evaluate predictive models by partitioning the samples into a training set to train the model, and a testing set to evaluate it [4].

Fig. 13 illustrates the flow of K-fold stacking model. In k-fold cross-validation, ”the original sample is randomly partitioned into k equal size subsamples” [4]. Then, of the k subsamples, a single subsample is used as the validation data for testing the model, and the remaining $k-1$ subsamples are used as training data [4]. The cross-validation process is then repeated k times (the folds), with each of the k subsamples used exactly once [4]. The k results from the folds can then be averaged to produce a general estimation [4].

As all observations are used for both training and validation, we get a better estimate of model performance and check that the model is not performing differently after being trained on different segments of the labeled data [4].

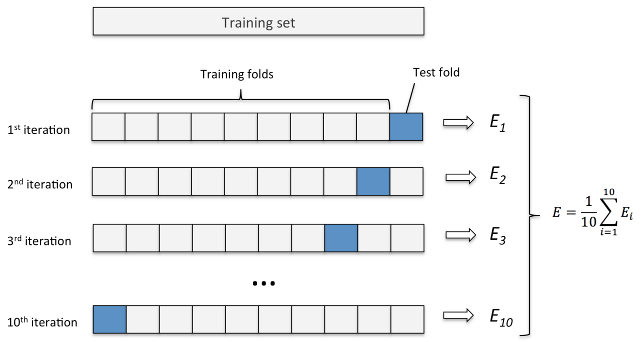


Fig. 13: The flow of K-fold stacking model. Figure source: [24]

Similarly as in [1], we used 10-fold cross-validation to evaluate the predictive performance of our models in terms of precision and recall. The outcome of this experiment is summarised in Fig. 14, for the two models log message classifier (msg) and patch classifier (patch) considered independently.

As both classifiers are trained using the same training instances, we combine both classifiers using unanimous voting, where C *(joint)* predicts the label 1 if either A *(msg)* or B *(patch)* predict 1, else it predicts 0.

Fig. 14: 10-fold cross validation report for *msg* and *patch* classifiers

	precision	recall	f1-score
(msg)	0.98	0.98	0.98
(patch)	0.54	0.53	0.53

We can see that the model trained with commit logs achieves quite high precision and recall rates result than the model trained for source code changes. There are several reasons for this: 1) Commit logs have more descriptive and detailed information. 2) As observed in Fig. 12, statistical measures about source code changes may be not distinct enough to reflect the distinction between security and non-security commits.

5 Conclusion

We learned a lot about machine learning, security engineering and open source software areas during the project.

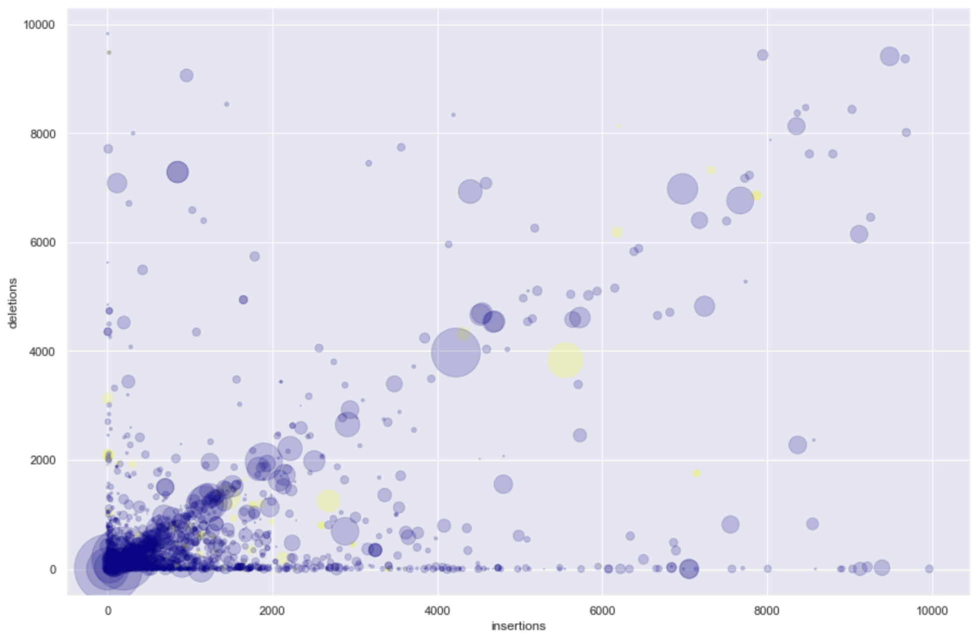


Fig. 11: Using point properties to encode features of the *patch* data

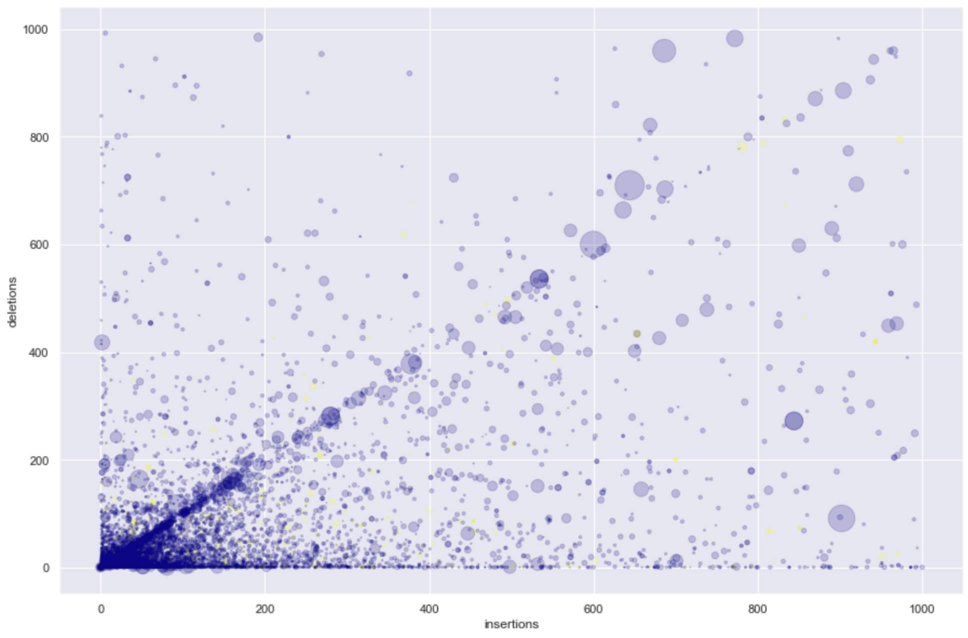


Fig. 12: Using point properties to encode features of the *patch* data. A closer look at the data evaluation

Our experiment demonstrates that our approach can automatically spot security fixes from other types of commits in source code repositories with a relatively high precision and recall rates using machine learning and natural language processing techniques. By using models with quite simple architecture, we trained two independent classifiers based on the commit message and the set of source code changes features and achieved good predictive performance.

While the results are satisfactory, we believe that additional improvements are possible, and further research in data processing, model selection and optimization is needed. Overall, we are satisfied with our project and consider it successful.

References

[1] Sabetta, Antonino, and Michele Bezzi. "A Practical Approach to the Automatic Classification of Security-Relevant Commits." 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2018. https://arxiv.org/abs/1807.02458

[2] scikit-learn user guide. Release 0.20.2. scikit-learn developers. https://scikit-learn.org/stable/_downloads/scikit-learn-docs.pdf

[3] Reis, Sofia, and Rui Abreu. "SECBENCH: A Database of Real Security Vulnerabilities." (2017). http://ceur-ws.org/Vol-1977/paper6.pdf

[4] Vanschoren, Joaquin. ”OpenML.” OpenML: Exploring Machine Learning Better, Together. https://www.openml.org/a/estimation-procedures/1
[5] Buitinck, Lars, et al. ”API design for machine learning software: experiences from the scikit-learn project.” arXiv preprint arXiv:1309.0238 (2013). https://arxiv.org/abs/1309.0238
[6] OWASP Foundation. ”OWASP Top Ten Project.”s https://www.owasp.org/index.php/Category:OWASP_Top_Ten _Project
[7] Shalev-Shwartz, Shai, and Shai Ben-David. Understanding Machine Learning: from Theory to Algorithms. Cambridge University Press, 2017.
[8] Snyk.io, ”The state of open-source security,” 2017. [Online] https://snyk.io/stateofossecurity/pdf/ The%20State%20of%20Open%20Source.pdf
[9] Git. ”Getting Started - About Version Control.” git-scm.com/book/en/v1/Getting-Started-About-Version-Control.
[10] cve-search. ”cve-search.” GitHub, 2018. github.com/cve-search/cve-search.
[11] International Conference on Mining Software Repositories. ”Mining Software Repositories”. http://www.msrconf.org/
[12] Krsul, Ivan Victor. ”Software vulnerability analysis.” West Lafayette, IN: Purdue University, 1998. https://dl.acm.org/citation.cfm?id=927682
[13] Manning, Christopher D., and Hinrich Schutze. ”Foundations of Statistical Natural Language Processing. MIT, 2008.” https://mitpress.mit.edu/books/foundations-statistical-natural-language-processing
[14] Vanderplas, Jacob T. ”Python Data Science Handbook: Essential Tools for Working with Data.” OReilly, 2017.
[15] Appendix: Figure Code https://jakevdp.github.io/ PythonDataScienceHandbook/06.00-figure-code.html#Features-and-Labels-Grid
[16] Shirey, Robert. ”Internet security glossary, version 2. No. RFC 4949. 2007.” https://tools.ietf.org/html/rfc4949
[17] Shull, Forrest, et al. ”Knowledge-sharing issues in experimental software engineering.” Empirical Software Engineering 9.1-2 (2004): 111-137. https://dl.acm.org/citation.cfm?id=966783
[18] Briand, Lionel C. ”A critical analysis of empirical research in software testing.” null. IEEE, 2007. https://ieeexplore.ieee.org/document/4343726
[19] Arthur Samuel, infolab.stanford.edu/pub/voy/museum/samuel.html.
[20] Zhou, Yaqin, and Asankhaya Sharma. ”Automated identification of security issues from commit messages and bug reports.” Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, 2017. https://dl.acm.org/citation.cfm?id=3117771
[21] Bachmann, Adrian, et al. ”The missing links: bugs and bug-fix commits.” Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. ACM, 2010. http://cabird.com/pubs/bachmann2010mlb.pdf
[22] Bachmann, Adrian, and Abraham Bernstein. ”Data retrieval, processing and linking for software process data analysis.” University of Zurich, Technical Report (2009). https://www.merlin.uzh.ch/contributionDocument/download/2156
[23] ”The Open Source Definition” https://opensource.org/osd
[24] Rosaen ”K-fold cross-validation” http://karlrosaen.com/ml/learning-log/2016-06-20/
[25] CVE Details https://www.cvedetails.com/
[26] Open Hub, the open source network https://www.openhub.net/
[27] About Pythonffl https://www.python.org/about/