Empirical Software Engineering (2022) 27: 151
https://doi.org/10.1007/s10664-022-10168-9

SSPCATCHER: Learning to catch security patches

Arthur D. Sawadogo[1] · Tegawendé F. Bissyandé[2] · Naouel Moha[1] · Kevin Allix[2] · Jacques Klein[2] · Li Li[3] · Yves Le Traon[2]

Accepted: 17 March 2022 / Published online: 6 August 2022

Abstract

Timely patching (i.e., the act of applying code changes to a program source code) is paramount to safeguard users and maintainers against dire consequences of malicious attacks. In practice, patching is prioritized following the nature of the code change that is committed in the code repository. When such a change is labeled as being security-relevant, i.e., as fixing a vulnerability, maintainers rapidly spread the change, and users are notified about the need to update to a new version of the library or of the application. Unfortunately, oftentimes, some security-relevant changes go unnoticed as they represent *silent fixes* of vulnerabilities. In this paper, we propose SSPCATCHER, a Co-Training-based approach to catch security patches (i.e., patches that address vulnerable code) as part of an automatic monitoring service of code repositories. Leveraging different classes of features, we empirically show that such automation is feasible and can yield a precision of over 80% in identifying security patches, with an unprecedented recall of over 80%. Beyond such a benchmarking with ground truth data which demonstrates an improvement over the state-of-the-art, we confirmed that SSPCATCHER can help catch security patches that were not reported as such.

1 Introduction

Recently, our digital world was shaken by two of the most widespread malware outbreaks to date, namely WannaCry and Petya. Interestingly, both leveraged a known exploit with an available patch (Trend Micro 2017). Despite the availability of such a patch that could have prevented an infection, a large number of systems around the globe were impacted, leading to a loss of over 4 billion US dollars (Berr 2017). In a typical scenario of vulnerability correction, a developer proposes changes bundled as a software *patch* by pushing a *commit* (i.e., patch + description of changes) to the code repository, which is analyzed by the project maintainer, or a chain of maintainers. The maintainers eventually reject or apply the changes

Communicated by: Hélène Waeselynck

✉ Arthur D. Sawadogo
sawadogo.delwende_donald_arthur@courrier.uqam.ca

Extended author information available on the last page of the article.

to the master branch. When the patch is accepted and released, all users of the relevant code must apply it to limit their exposure to attacks. In reality, for some organizations, there is a time lag between the release of a patch and its application. While in the cases of critical systems, maintainers are hesitant to deploy updates that will hinder operations with downtime, in other cases, the lag can be due to the fact that the proposed change has not been properly advertised as *security-relevant*, and is not thus viewed as critical.

Patching (i.e., the act of applying code changes to a program source code) is an absolute necessity. Timely patching of vulnerabilities in software, however, mainly depends on the tags associated to the change, such as the commit log message, or on the availability of references in public vulnerability databases. For example, nowadays, developers and system maintainers rely on information from the National Vulnerability Database (NIST 2018) to react to all disclosed vulnerabilities. Unfortunately, a recent study on the state of open source security (Snyk.io 2017) revealed that only 9% of maintainers file for a Common Vulnerability Enumeration (CVE) ID after releasing a fix to a vulnerability. The study further reports that 25% of open source software projects completely silently fix vulnerabilities without disclosing them to any official repository.

Silent vulnerability fixes are a concern for third-party developers and users alike. Given the low coverage of official vulnerability repositories, there are initiatives in the software industry to automatically and systematically monitor source code repositories in real-time for identifying security-relevant commits, for example by parsing the commit logs (Zhou and Sharma 2017) or by mining the code of the components (Scandariato et al. 2014). Manual analysis of code changes is indeed heavy in terms of manpower constraints, requires expert knowledge, and can be error-prone. Some other existing works in this area also use the code and logs of commits as inputs to train machine learning models for predicting security-relevant commits. Sabetta et al. (Sabetta and Bezzi 2018) leveraged bag-of-words model to identify security-relevant fixes. They achieved a high precision (at 80%) but face two major problems that we attempt to solve: their features are not explicitly related to security semantics; they do not address the unbalanced dataset problem in real-world scenarios. It is further noteworthy that the literature has also proposed approaches (Zhou and Sharma 2017; Scandariato et al. 2014) for detecting code changes that introduce security vulnerabilities. Conversely, we are focused on *identifying whether a proposed patch is applying code changes to fix an existing vulnerability*.

In this paper, we investigate the possibility to apply machine learning techniques to automate the identification of source code changes that actually represent security patches (i.e., patches that address vulnerable code). To that end, we investigate three different classes of features related to the change metadata (e.g., commit logs), the code change details (e.g., number of lines modified), as well as specific traits that are recurrent in vulnerabilities (e.g., array index change). We then build on the insight that analysts can *independently* rely either on commit logs or on code change details to suspect a patch of addressing a vulnerability. Thus, we propose to build a Co-Training based approach where two classifiers leverage separately text features and code features to eventually learn an effective model. This semi-supervised learning approach further accounts for the reality that the datasets available in practice include a *large portion of samples whose labels (i.e., "security-relevant" or not) are unknown*. We refer to our approach as SSPCATCHER (for "Security Sensitive Patch Catcher").

Our work deals with the automation of the identification of security patches (i.e., patches fixing vulnerabilities) once a code change is presented to be applied to a codebase. To align with realistic constraints[a] of practitioners, we only leverage the information available within the commit.

[a]In practice, identifying security patches must be done at commit time. An approach would be very successful if it could leverage future comments of bug reports and advisories inputs (e.g., CVE). Such information is however not available in reality when the commit is made.

Overall, we make the following contributions:

- We motivate and dissect the problem of identifying security-relevant code changes in Section 2. In particular, we investigate the discriminative power of a variety of features to clarify the possibility of a learning process.
- We propose a semi-supervised approach with Co-Training (Blum and Mitchell 1998) which we demonstrate to yield high precision (95%) and recall (88%). This represents a significant improvement over the state-of-the-art.
- Finally, we show that our approach can help flag patches that were unlabeled until now. We have confirmed our findings by manual analysis, with the help of external expertise.

The implementation, dataset, and results of SSPCatcher are publicly available for the community as a replication package :

http://github.com/vulnCatcher/vulnCatcher

The remainder of this paper is organized as follows. We motivate our study in Section 2 and overview data collection in Section 3. Section 4 describes SSPCatcher while Section 5 presents the experimental study and results. Section 6 discusses threats to validity and future work. We discuss related work in Section 7 and Section 8 concludes this work.

2 Motivation

The urgency of updating a software given a proposed change is assessed at different levels of the software development cycle. We consider the cases of developer-maintainer and maintainer-user communications.

(1) ***Patch processing delays by maintainers.*** We consider the case of the Linux kernel, which is developed according to a hierarchical open source model referred to as Benevolent dictator for life (BDFL) (van Rossum 2008). In this model, anyone can contribute, but ultimately all contributions are integrated by a single person, Linus Torvalds, into the mainline development tree. A Linux kernel maintainer receives patches related to a particular file or subsystem from developers or more specialized maintainers. After evaluating and locally committing them, he/she propagates them upwards in the maintainer hierarchy, eventually up to Linus Torvalds. Since the number of maintainers is significantly lower than that of contributors, there is a delay between a patch authoring date and its commit date. A recent study, however, has shown that author patches for Linux are addressed in a timely manner by maintainers (Koyuncu et al. 2017). Nevertheless, given the critical nature of a security patch, we expect its processing to be even more speedy if the commit message contains relevant information that attracts maintainers' attention.

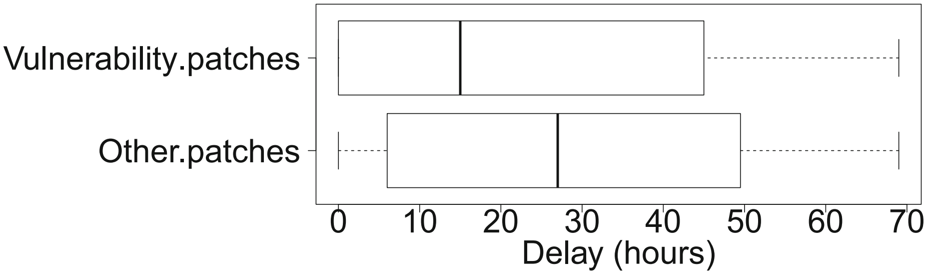


Fig. 1 Delays for validating contributor patches in Linux based on explicit vulnerabilities

Figure 1 illustrates the delay computed on randomly sampled sets of 1 000 commits where the log clearly contained a CVE reference, and 1 000 commits with no such references. These 1 000 commits selected are a part of the negative dataset, identified by the data collection process described in Section 3.1; therefore these commits do not involve vulnerability fixes.

The delay is computed as the difference of time between the contribution date (i.e., Author date in git) and the date it was accepted in the repository (i.e., Commit date in git). The boxplots show how patches that are explicitly related to vulnerabilities are validated faster than other patches: on median average, security patches are validated fifteen hours faster. We confirmed that the difference is statistically significant with MWW tests (Mann and Whitney 1947).

Often, if proper notice is given, maintainers are likely to prioritize the validation of security patches.

(2) ***Version release delays for users.*** In the development cycle of software, versioning allows maintainers to fix milestones with regards to the addition of new features, or the stabilization of a well-tested branch after the application of several bug fixes. However, when a security patch is applied to the code base, it is common to see maintainers release a new version early to protect users against potential attacks. These exceptional cases could then change the versioning cycle to prioritize customer's security and motivate the goal of our paper: identifying silent vulnerability fixes.

We did a study to confirm this assumption. We consider the case of the OpenSSL library and compare the delay between a given commit and the subsequent version release date (which is inferred by checking commits with version tags). The delay was computed for all the 1 550 OpenSSL commits (495 of which carry security patches) collected in our study datasets.

Boxplot representations in Fig. 2 show that many OpenSSL versions are released just after security patches. In contrast, the gap between any other commit and a version release is bigger: releases are made on average seven days after a security patch, but about twenty days after other types of patches.

To reduce user exposure, it is necessary to release new versions when vulnerabilities are patched. To that end, it is critical to identify such security patches.

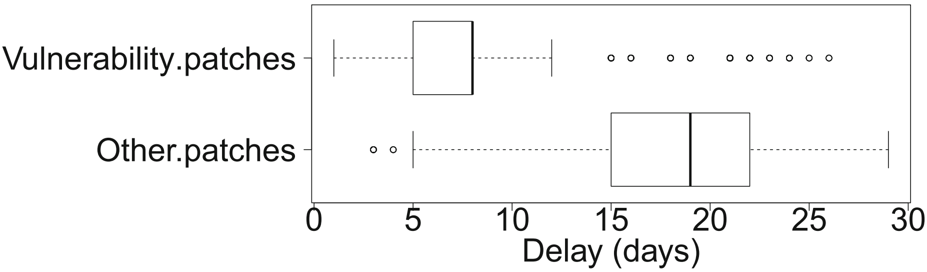


Fig. 2 Comparative delays for OpenSSL release after an explicit security patch vs after any other patch

3 Data Collection

For much modern software, developers rely on the git version control system. Git makes available the history of changes that have been made to the code base in the form of a series of patches. Thus, a patch constitutes a thorough summary of a code change, describing the modification that a developer has made to the source code at the time of a commit. Typically, a patch as depicted in Fig. 3, includes two artifacts: a) the log message in which the developer describes the change in natural language; b) the diff which represents the changes that are to be applied. The illustrated vulnerability, as in many cases, is due to a missing constraint that leaves a window for attackers to exploit.

```
commit 5ebff5337594d690b322078c512eb222d34aaa82
Author: Michal Schmidt <anonymized@redhat.com>
Date: Fri Mar 2 10:39:10 2012 +0100

    util: never follow symlinks in rm_rf_children()
    The function checks if the entry is a directory
    before recursing, but there is a window between
    the check and the open, during which the
    directory could be replaced with a symlink.
    CVE-2012-1174
    https://bugzilla.redhat.com/show_bug.cgi?id=803358

diff --git a/src/util.c b/src/util.c
index 20cbc2b0d..dfc1dc6b8 100644
--- a/src/util.c
+++ b/src/util.c
@@ -3593,7 +3593,8 @@ static int rm_rf_children(int fd,...) {
if (is_dir) {
    int subdir_fd;
- if((subdir_fd = openat(fd, de->d_name, O_RDONLY|...)) < 0){
+ subdir_fd = openat(fd, de->d_name, O_RDONLY|...|O_NOFOLLOW);
+ if (subdir_fd < 0) {
           if (ret == 0 && errno != ENOENT)
               ret = -errno;
           continue;
```

Fig. 3 Example of a security patch in the OpenSSL library

For our experiments, we consider three projects whose code is widespread among IT systems: the **Linux** kernel development project, the **OpenSSL** library project, and the **Wireshark** network protocol analyzer. We also consider the Secbench (Reis and Abreu 2017) dataset, which includes a large number of vulnerability fixing commit samples from a variety of projects using mixed programming languages.

For each of our study projects, we attempt to collect *positive* and *negative* data for the classical binary classification task, as well as the *unlabeled* data for our semi-supervised learning scenario:

- **Positive data** (i.e., *security patches*). We collect patches reported as part of security advisories, and thus known to be addressing a known and reported vulnerability.
- **Negative data** (i.e., *non-security patches*). We use heuristics to build the dataset of negative data. To ensure that it is unbiased and representative, we explicitly consider different cases of non-security patches and transparently collect these sets separately with a clear process to enable replication. Concretely, we consider:
  - *Pure bug fixing patches*. We collect patches that are known to fix bugs in project code, but that are not security-relevant.
  - *Code enhancement patches*. We collect patches that are not about fixing bugs or vulnerabilities. Such patches may be delivered by commits to perform code cleaning, feature addition, performance enhancement, etc.
- **Unlabeled data**. We finally collect patches that are about fixing the code, but for which we do not yet know whether it is about fixing a vulnerability or non-security bugs.

The creation of these datasets is summarized in Fig. 4 and detailed in the following paragraphs.

3.1 Security patches (for positive datasets)

Security patches from study projects We leverage a recent framework proposed by Jimenez et al. (Jimenez et al. 2018) for automated collection of vulnerability instances from software archives. The framework builds upon the National Vulnerability Database information and attempts to connect such information with other sources such as bug tracking systems and git repositories. The data recovered include information, for each item, about the CVE ID, the CVE description, the time of creation, the associated bug ids from the project bug tracking system, the list of impacted software versions, and the list of commits that fixed the vulnerability. Overall, as of July 2018, we managed to retrieve 1 398,

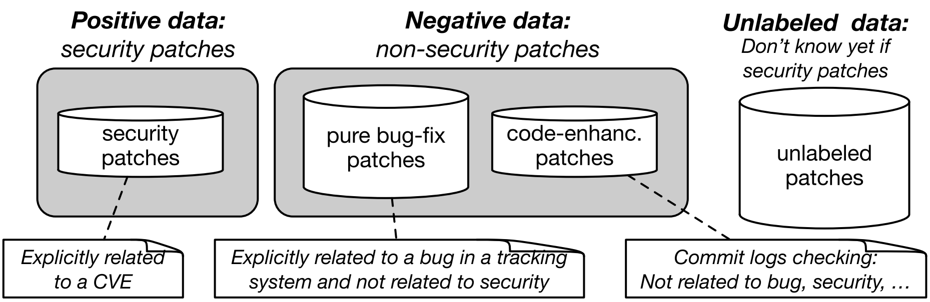


Fig. 4 Distinct subsets of the dataset built for our experiments

986, and 495 security patches for Linux, Wireshark, and OpenSSL respectively for this part. We call this part of the whole dataset ***C-projects dataset*** given the uniform nature of the programming language used.

Security patches from Secbench We consider data from the Secbench (Reis and Abreu 2017) database, which contains 676 reported vulnerability patches from 238 projects. The authors exploited the projects' commits using regular expressions for each vulnerability and then classified the vulnerabilities using the CWE taxonomy. Some vulnerabilities contain score and severity information (CVE). However, some projects are no longer accessible. Overall, we managed to collect a total of 648 security patches within 114 projects. Most vulnerability samples are contributed by only a few number of projects as shown by the long tail distribution in Secbench (cf. Fig. 5).

3.2 Pure bug fixing patches (for negative datasets)

To ensure that SSPCatcher can effectively differentiate security-relevant fixes from other fixes, we set to collect a dataset of non-security-relevant patches following conservative heuristics. First, we consider patches that are not reported in a security advisory, and whose commit logs do not include "vulnerability" or "security" keywords. Then, we focus on those patches whose commits are linked to a bug reported in a bug tracking system.

Finally, we ensure that the bug report itself does not hint at a potential security issue. For that, we follow the approach proposed by security analysts Zhou and Asankhaya (Zhou and Sharma 2017). They proposed a regular expression that yields to catch security-sensitive commits. It, therefore, looks for keywords and combinations of keywords in the commits, for example: "denial.of.service", "directory. traversal", etc. We then applied this approach and drop all cases where the bug report matches the regular expression provided in Table 1. Overall, with this method, we managed to retrieve 1 934, 2 477 and 8 142 pure

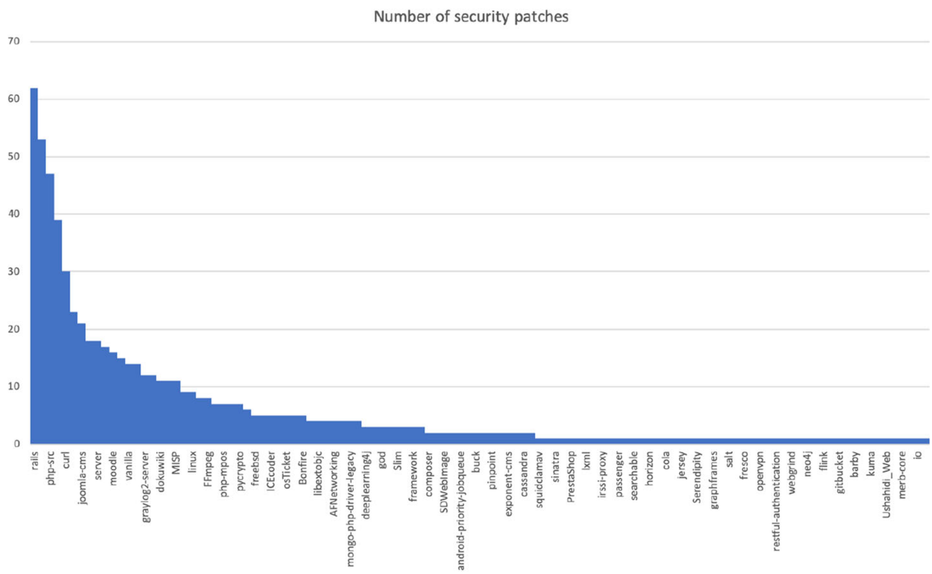


Fig. 5 Secbench dataset distribution

Table 1 Regular expression used to filter out security-related issues described in bug reports

```
(?i)(denial.of.service|\bXXE\b|remote.code.execution
|\bopen.redirect|OSVDB|\vuln|\CVE\b|\bXSS\b|\bReDoS\b
|\bNVD\b|malicious|x-frame-options|attack|cross.site
|exploit|directory.traversal|\bRCE\b|\bdos\b|\bXSRF\b
|clickjack|session.fixation|hijack|advisory|insecure
|security|\bcross-origin\b|unauthori[z|s]ed
|infinite.loop|authenticat(e|ion)|brute force|bypass
|constant.time|crack|credential|\bDoS\b|expos(e|ing)
|hack|harden|injection|lockout|overflow|password
|\bPoC\b|proof.of.concept|poison|privilege
|\b(in)?secur(e|ity)|(de)?serializ|spoof|timing|traversal)
```

bug fixing patches for Linux, Wireshark, and Secbench respectively. Our dataset does not contain any pure bug-fix patches for OpenSSL due to missing links between commits and bug reports of OpenSSL. Future work could consider using state-of-the-art bug linking approaches (Nguyen et al. 2012; Wu et al. 2011; Bissyande et al. 2013).

3.3 Code enhancement patches (for negative datasets)

To ensure that our model will not be overfitted to the cases of fixing patches, we collect noise dataset represented by commits that enhance the code base with new feature additions. The model is aimed at recognizing security fixes vs all others altogether. Thus other types of code enhancement patches are also discriminated against. We considered the case of feature-addition more explicitly in the labeling of the negative set because they are easy to label and also to increase the diversity of the negative set.

We thus set to build a parser of commit logs for identifying such commits. To that end, we first manually investigate a small set of 500 commits over all the projects and attempt to identify what keywords can be leveraged. Given the diversity of fixes and commit log tokens, we eventually decide to focus on keywords recurrent in all commits that are not about feature addition, in order to reduce the search space. These are: *bug, fix, bugzilla, resolve, remove, merge, branch, conflict, crash, debug*. Excluding known security patches, known bug fixes (whether pure or not), and those that match the previous keywords, we consider the remaining patches as the sought noise for the learning process. Overall, we collected 681, 658, 679, 2 527 code enhancement patches for Linux, Wireshark, OpenSSL, and Secbench respectively.

3.4 Unlabeled patches

Ultimately, our goal is to provide researchers and practitioners with an approach for identifying silent security fixing patches. Thus, we hypothesize that some fixing patches are actually unlabeled security patches. To build a dataset of unlabeled patches where security patches may be included, we parse all remaining patches (i.e., patches that are not collected in the previous datasets) and further hone in the subset of unlabeled patches that are more relevant to be caught as security patches. To that end, we focus on commits whose

Table 2 Statistics on the collected datasets

	OpenSSL	Wireshark	Linux	Secbench	Total
Security patches	495	1 398	986	648	3 616
Pure bug fixing patches	(–) [2]	1 934	2 477	8 142	12 553
Code enhancement patches	618	681	658	2 527	4 483
Unlabeled patches	437	18 067	147 746	69 138	235 388

logs match the regular expression `(?i)(bug|vuln`[1]`|fix)`. Eventually, we collected 147 746, 18 067,437 and 69 138 unlabeled patches for Linux, Wireshark, OpenSSL, and Secbench respectively.

Table 2 summarizes the statistics on the collected datasets. We note that, as we postulated, most patches are unlabeled. Security patches are mostly silent (Snyk.io 2017). Even in the case where a patch is present in a security advisory (i.e., the NIST vulnerability database in our case), the associated commit log may not explicitly use terms that hint to a security issue. For example, with respect to the regular expression in Table 1, we note that 15.21% of Wireshark security patches, 37.19% of Linux security patches, and up to 98.78% of OpenSSL security patches do not match security-related tokens.

4 SSPCatcher

Our work addresses a **binary classification problem** of distinguishing security patches from other patches: we consider a combination of *text analysis of commit logs* and *code analysis of commit changes diff* to catch security patches. To that end, we proceed to the extraction of ”facts”(e.g. #Sizeof added, #Sizeof removed, etc.) from text and code, and then perform a feature engineering that we demonstrate to be efficient for discriminating security patches from other patches. Finally, we learn a prediction model using machine learning classification techniques.

In a typical classification task, an appropriately labeled training dataset is available. In our setting, however, this is not the case as introduced earlier: in our dataset, when a commit is attached to a CVE, we can guarantee that it does provide a security patch; when the commit does not mention a CVE, we cannot assume that it does not provide a security patch. Therefore, for positive data, i.e., security patches, we can leverage the limited dataset of patches that have been listed in vulnerability databases (e.g., the NVD). There is, however, no corresponding set of independently labeled negative data, i.e., non-security patches, given that developers may silently fix their vulnerable code. This problem was raised in previous work on the identification of bug fixing patches by Tian et al. (Tian et al. 2012). Nevertheless, our setting requires even more refined analysis since security patches can be easily confused with a mere non-security-relevant bug fix. To address the problem of having a small set of labeled data and a large set of unlabeled data for security patches, we consider a Co-Training (Blum and Mitchell 1998) approach where we combine two models, each trained with features extracted from two disjoint aspects (commit message vs. code diff) of our dataset. This process has been shown to be one of the most effective techniques for semi-supervised learning (Nigam and Ghani 2000).

[1]Commits with logs matching keyword “vuln” cannot be directly considered to be security patches without an audit of the full description and even of the code change.

Concretely, SSPCATCHER considers commit logs, on the one hand, and code diffs, on the other hand, as redundant views of the changes, given that the former describes the latter. Then we train two separate classifiers, one for each view, that are iterated by exchanging labeled data until they agree on classification decisions (cf. Section 4.3).

In this section, we first provide information on feature engineering (cf. Section 4.1) and assessment (cf. Section 4.2). Then, we present the Co-Training approach (cf. Section 4.3).

4.1 Feature Extraction and Engineering

The objective of the feature extraction step is to transform the high-volume raw data that we have previously collected into a reduced dataset that includes only the important facts about the samples. The feature extraction then considers both the textual description of the commits (i.e., the message describing the purpose of the change) and the code diff (i.e., the actual modifications performed). The feature engineering step then deals with the representation of the extracted facts into numerical vectors to be fed to machine learning algorithms.

4.1.1 Commit Text Features

We extract text features by considering all commit logs as a bag of words, excluding stop words (e.g., "as", "is", "would", etc.) which are very frequently appearing in any English document and will not hold any discriminative power. We then reduce each word to its root form using Porter' stemming (Porter 1980) algorithm. Finally, given the large number of rooted words, and to limit the curse of dimensionality, we focus on the top 10 of the most recurring words in commit logs of security patches for the feature engineering step. This number is selected as a reasonable vector size to avoid having a too-sparse vector for each commit, given that commit logs are generally short. We calculate the *inverse document frequency* (*idf*), whose formula is provided in the equation below. It is a measure of how much information the word provides, that is, whether it is common or rare across all commit logs. The feature value for each commit is then computed as the $idf_i = log \frac{|D|}{|\{d_j : t_i \in d_j\}|}$ with $|D|$ being the total number of documents in the corpus and $|\{d_j : t_i \in d_j\}|$ being the number of documents where term t_i appears.

4.1.2 Commit Code Features

Besides description logs, code change details are available in a commit and can contribute to improve the efficiency of the model as demonstrated by Sabetta and Bezzi (Sabetta and Bezzi 2018). Nevertheless in their work, these security researchers considered all code change tokens as a bag of tokens for embedding. In our work, we propose to refine the feature selection by selecting meaningful facts from code to produce an *accurate* and *explainable* model. To that end, on the one hand, we are inspired by the classification study of Tian et al. (Tian et al. 2012), and we extract code facts representing the spread of the patch (e.g., the number of files/lines modified, etc.), the code units involved (e.g., the number of expressions, boolean operators, function calls, etc.). On the other hand, we manually investigated a sample set of 300 security patches and noticed a few recurring code facts: for example, `sizeof` is often called to fix buffer overflow vulnerabilities, while `goto`, `continue` or `break` constructs are frequently involved in security fixes related to loops, etc. Thus, we engineer two sub-categories of features: *code-fix features* and *security-sensitive features*.

Overall, Table 3 provides an enumeration of the exhaustive list of features used in this study.

4.2 Feature Assessment

4.2.1 Statistical Analysis

Before leveraging the features that we have engineered based on manual analysis and intuitive facts, we propose to assess their fitness with respect to discriminating security patches against other types of patches. To that end, we used the Mann-Whitney U test (Mann and Whitney 1947)

in order to compare the distribution of a given feature within the set of security patches against the combined set of pure bug fixing patches and code enhancement patches. The null hypothesis states that the feature is distributed independently from whether the commit fixes a vulnerability 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.

The Mann-Whitney U tests helped discover that a large majority (i.e., 53 out of 67) of the computed features were not meaningful unless we rescaled the feature values according to the size of the patches. Indeed, for example, code enhancement patches that can be huge (e.g., the addition of a new program file) may include a number of loops and sizeof calls,

Table 3 Exhaustive list of features considered for learning

ID	code-fix features	ID	security-sensitive features
F1	#files changed in a commit	F1	#Sizeof added
F2	#Loops added	F2	#Sizeof removed
F3	#Loops removed	F3	F1 - F2
F4	F2 - F3	F4	F1 + F2
F5	F2 + F3	F5-F6	Similar to F1 to F2 for #continue
F6-F9	Similar to F2 to F5 for #ifs	F7-F8	Similar to F1 to F2 for #break
F10-F13	Similar to F2 to F5 for #Lines	F9-F10	Similar to F1 to F2 for #INTMAX
F14-F17	Similar to F2 to F5 for #Parenthesized expressions	F11-F12	Similar to F1 to F2 for #goto
F18-F21	Similar to F2 to F5 for #Boolean operators	F13-F14	Similar to F1 to F2 for #define
F22-F25	Similar to F2 to F5 for #Assignments	F15-F18	Similar to F1 to F4 for #struct
F26-F29	Similar to F2 to F5 for #Functions call	F19-F20	Similar to F1 to F2 for #offset
F30-F33	Similar to F2 to F5 for #Expression	F21-F24	Similar to F1 to F4 for #void
ID	text features		
W1-W10	10 Most recurrent non-stop words		

Table 4 Statistical analysis results for top normalized features with highest discriminative potential

	Code-fix features			sec.-sensitive features			Text features		
	F6	F16	F24	F11	F22	F24	W2	W4	W6
Mean for security patches	0.120	0.038	0.110	0.004	0.006	0.350	0.360	0.360	0.350
Mean for other patches	0.090	0.016	0.050	0.003	0.004	0.330	0.310	0.320	0.330
P-value (MWW)	$5e^{-62}$	$2e^{-40}$	$4e^{-103}$	$1e^{-13}$	$1e^{-15}$	$6e^{-47}$	$2e^{-65}$	$2e^{-66}$	$7e^{-50}$

making related features meaningless, unless their numbers are normalized to the size of code in the patch. We then applied, for each feature value per patch, the following formula:

$$F_{norm} = \frac{F}{\#patch\_added\_lines + \#patch\_removed\_lines} \tag{1}$$

where the normalized value F_{norm} of a feature is computed by taking into account the patch size. Table 4 provides some example cases where the statistical tests were successful against a strict significance level of $\alpha = 0.0005$ for the p-value. Due to space limitations, we show only top-3 features per feature group. For 52 out of 67 features engineered, the statistical analysis shows a high potential of discriminative power. Nevertheless, in the rest of our experiments, and following insights from previous studies (Perl et al. 2015), we keep all features for the learning process as some combinations may contribute to yielding an efficient classifier.

4.2.2 Classification Experiments

The previous statistical analysis assessed the discriminative power of engineered features with respect to security patches and the combined set of bug fixing and code enhancement patches. We propose to further assess the behaviour of one-class classification models with these features applied to the unlabeled patches. Our experiments aim at answering two questions:

- *Can the features help effectively classify unlabeled patches?* We attempt to assess to what extent unlabeled patches that are flagged as security patches would constitute noise or good samples to help augment the training data of a binary classifier.
- *Are the feature categories independent and thus splittable for a Co-Training model learning?* The choice of Co-Training as an approach is based on the hypothesis that the views are redundant. However, another constraint for the efficacy of Co-Training is that the features must be independent (Nigam and Ghani 2000) (i.e., they do not lead to exactly the same classifications).

Features efficiency. Various verification problems in machine learning involve identifying a single class label as a 'target' class during the training process, and at prediction time make a judgement as to whether or not an instance is a member of the target class (Hempstalk and Frank 2008). In many cases, a one-class classifier is used in preference to a multi-class classifiers, mainly because it is inappropriate or challenging to collect or use non-target data for the given situation. In such cases, the one-class classifier is actually an *outlier detector* since it attempts to differentiate between data that appears normal (i.e., from the target class) and abnormal with respect to a training data composed only of normal data. Thus, if the features are not efficient to fully characterize the normal data in the training set, many samples classified as normal will actually be false positives and thus constitute *noise* in an augmented set of normal data.

Given the lack of ground truth (for unlabeled patches), we assess whether unlabeled patches that are flagged as security patches by a one-class classifier are noise (i.e., false positives), and thus deteriorate a binary classification performance when added to a training dataset. The comparison is done following two experiments:

– First, we compute accuracy, precision and recall metrics of a classical SVM **binary classifier** using the existing set of security patches as positive data and other sets of non-security (i.e., bug-fix and code enhancement) patches as negative data.
– Second, we augment the existing set of security patches with automatically labeled patches after applying a **one-class classifier** to the dataset of unlabeled patches. Then we use this augmented set as the positive data and redo the first experiment. This workflow is detailed in Fig. 6.

If the features are not efficient in characterizing security patches, the one-class classifier will yield false positives and false negatives. Thus, when adding false positives to the ground truth positive data, we will be introducing noise which will lead to performance degradation. However, if the features are efficient, we will be increasing the training set and potentially leading to a better classification performance.

Equations (2) and (3) provide the standard formulas for computing performance metrics, where TP is the number of True Positives, TN that of True Negatives, FP that of False Positives and FN that of False Negatives.

$$Precision = \frac{TP}{TP + FP} \;;\; Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \qquad (2)$$

$$Recall = \frac{TP}{TP + FN} \;;\; F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \qquad (3)$$

Our experiments are performed with 10-Fold cross validation and performance is measured for the target class of security patches and only on the initial ground truth samples. Using only the initial set of security patches in the training dataset, we record an average Accuracy of 58% (Recall = 56%, Precision= 71%). However, when we augment the training set with flagged unlabeled patches, we observe a clear improvement of the accuracy to 79% (Recall = 76%, Precision= 85%).

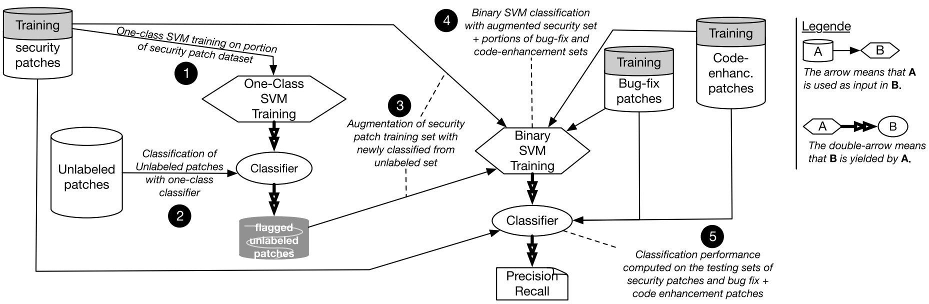
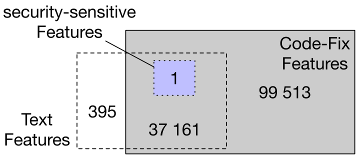
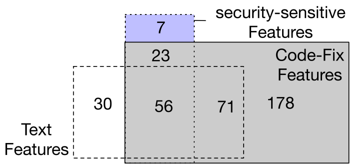


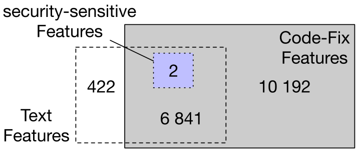
Fig. 6 Workflow for assessing the discriminative power of features



(a) Flagged Linux unlabeled patches



(b) Flagged OpenSSL unlabeled patches



(c) Flagged Wireshark unlabeled patches

Fig. 7 Euler diagrams representing the overlaps between sets of unlabeled patches that are classified as security patches when using One-Class SVM model based on variants of feature sets

> The engineered features are effective for characterizing security patches. They can be used to collect patches for artificially augmenting a training dataset.

Features independence. The two most closely related work in the literature (Zhou and Sharma 2017; Sabetta and Bezzi 2018) rely on commit text or/and code changes that they treat as simple bags of words. Nevertheless, no experiments were performed to assess the contribution and complementarity of the different information parts. We explore these contributions by evaluating the overlap among the unlabeled patch subsets that are flagged when using different feature sets. Figure 7 illustrates these overlaps with Euler diagrams for the different projects considered in our study. We note that although there are overlaps, a large portion of samples are detected exclusively with each feature set (e.g., in Linux, $99,513 + 395 = 99,908$ patches out of $99,513 + 395 + 1 + 37,161 = 137,070$ patches – 73%– are exclusively detected by either code-fix features or text features). Nevertheless, we note that security-sensitive features are more tightly related to code-fix features (except for 7 patches in OpenSSL, all flagged patches with security-sensitive features are also flagged with code-fix features[2], which was to be expected given that security-sensitive features are also about "fixing" code).

We then conclude that *code-fix* features can be merged with *security-sensitive features* to form **code features**, which constitute a feature set that is **independent** from the **text features** set. As Krogel and Schefferd demonstrated, Co-Training is only beneficial if the data sets used in classification are independent (Krogel and Scheffer 2004). This insight on the sets of engineered features serves as the foundation for our model learning detailed in the following paragraphs.

> The engineered features are effective for characterizing security patches. They can be used to collect patches for artificially augmenting a training dataset.

[2]This does not mean that security-sensitive features are useless or redundant. Patches flagged with code-fix features are scarcely flagged with security-sensitive features.

4.3 Co-Training Model Learning

Experimental results described above have established that the different features engineered provide meaningful information for the identification of security patches. Nevertheless, given the large number of these features, manual construction of detection rules is difficult. We propose to apply techniques from the area of machine learning to automatically analyze the code commits and flag those that are most likely to be delivering security patches.

In the construction of our learning-based classifier, we stress on the need for practical usefulness to practitioners. Thus, following recommendations by authors (Perl et al. 2015) proposing automatic machine-learning approaches to support security analysts, we strive to build an approach towards addressing the following challenges:

– Generality: Our feature engineering mixes metadata information from commit logs, which may or may not be explicit, with numerical code metrics. It is thus important that the classifier effectively leverages those heterogeneous features to infer an accurate combined detection model.
– Scalability: Given that most relevant software projects include thousands of commits that must be analyzed, it is necessary for the approach to be able to operate on the large amount of available features in a reasonable time frame.
– Transparency: In practice, to be helpful for analysts, a classifier must provide human-comprehensible explanations with the classification decision. For example, instead of requiring an analyst to blindly trust a black-box decision based on deep features, information gain[3] (InfoGain) scoring values of human-engineered features can be used as hints for manual investigation.

4.3.1 Model Learning

Experiments with one-class classification have already demonstrated that it is possible to build a classifier that fits with the labeled patches in the ground truth data. Unfortunately, in our case, a major problem in building a discriminative classifier is the non-availability of labeled data: the set of unlabeled patches is significantly larger than the limited dataset of labeled patches that we could collect. A classification task for identifying security patches requires examples of both security and security-irrelevant patches. In related work from the security industry (Zhou and Sharma 2017), team members having relevant skills and experience spent several months labeling closed-source data to support the model learning. Since their dataset was not publicly[4] available, we propose to rely on the Co-Training algorithm to solve the non-availability problem. The algorithm was proposed by Blum and Mitchell (Blum and Mitchell 1998), for the problem of semi-supervised learning where there are both labeled and unlabeled examples. The goal of Co-Training is to enhance the performance of the learning algorithm when only a small set of labeled examples is available. The algorithm trains two classifiers separately on two sufficient and redundant views of the examples and lets the two classifiers label unlabeled examples for each other.

Figure 8 illustrates the Co-Training process implemented in this work. It takes labeled and unlabeled patches from a given project or a set of projects and learns a classification

[3]Information gain is a metric based on entropy that allows to tell how important a given attribute of the feature set is.

[4]Our requests to obtain datasets from authors of (Zhou and Sharma 2017) and (Sabetta and Bezzi 2018) remained unresponded.

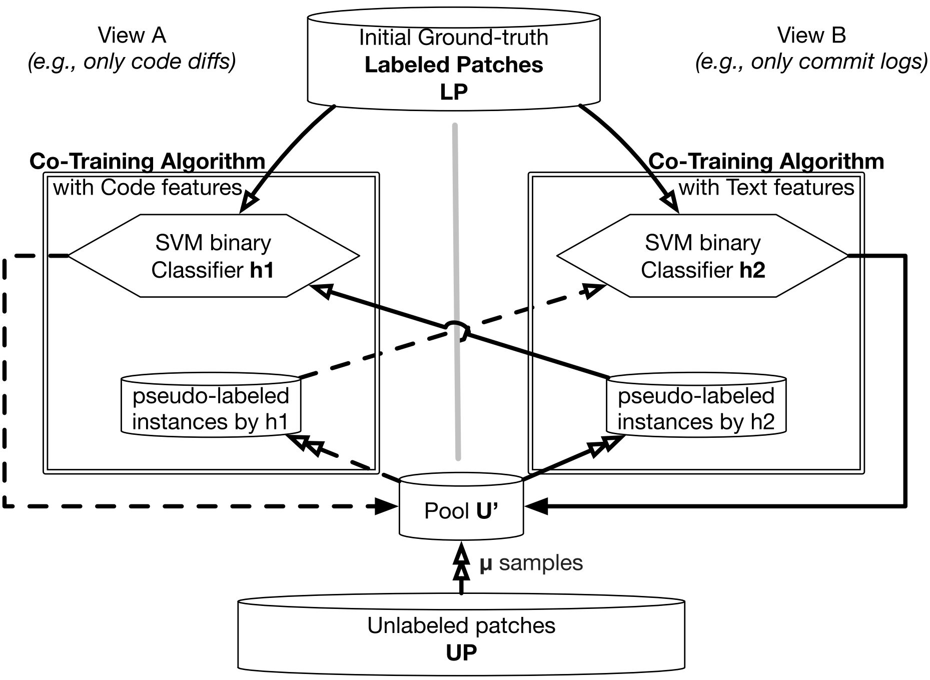


Fig. 8 Co-Training learning model (cf. details in Algorithm 1)

model for predicting patch security relevance. An important assumption in Co-Training is that each view is conditionally independent given the class label. We have demonstrated in Section 4.2.2 that this was the case for the different categories of features explored in this work. Indeed, Co-Training is effective if one of the classifiers correctly labels a sample that the other classifier previously misclassified. If both classifiers agree on all the unlabeled patches, i.e. they are not independent, labeling the data does not create new information.

Concretely, given a training set comprising labeled patches and noted LP, and a set of unlabeled patches UP, the algorithm randomly selects μ samples from UP to create a smaller pool U', then executes the process described in Algorithm 1 during k iterations.

The overall idea behind the Co-Training algorithm steps is that the classifier h_1 adds examples to the labeled set which are in turn used by the classifier h_2 in the next iteration and vice versa. This process should make classifiers h_1 and h_2 to agree with each other after k iterations. In this study, we selected Support Vector Machines (SVM) (Vapnik 2013) as the internal classification algorithm for the Co-Training. SVM indeed provides tractable baseline performance for replication and comparisons against state-of-the-art works.

4.3.2 Identification of Security Patches

Eventually, when the Co-Training is stabilized (i.e., the two internal classifiers agree), the output classifier can be leveraged to classify unlabeled patches. Eventually, in this work, we consider the classifier built on the code view (which has been constantly improved due to the co-training) as the yielded classifier.

Algorithm 1 Steps for each Co-Training iteration.

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

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

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

5 Experimental Study and Results

Our experiments aim at assessing the performance of the overall approach, detailing the impact of the Co-Training algorithm and comparing against the state-of-the-art. We investigate the following research questions:

- **[RQ-1.] What is the effectiveness of the proposed SSPCATCHER Co-Training based patch classification approach?**
  To answer this research question, we perform binary classification experiments and report on Precision, Recall and F-Measure performance metrics of the classifier when discriminating security patches. We also evaluate performance in terms of execution time.
- **[RQ-2.] Can SSPCATCHER be trained to predict security-relevant patches across projects?**
  We investigate the possibility of training a model by leveraging data from a given project and remaining effective on another target project. Firstly, we consider the case when the projects are written in the same programming language (C). Secondly, we consider projects that are written in mixed programming languages.
- **[RQ-3.] How does SSPCATCHER compare against the state-of-the-art?**
  First, we replicate the main components of the approach proposed by Sabetta et al. (Sabetta and Bezzi 2018) (i.e., SVM binary classification with bag-of-words features of code and log) and then compare this approach against SSPCATCHER on our datasets. Second, we conduct dissection study experiments where we evaluate the contribution

of our feature set and the choice of Co-Training by benchmarking against other design choices.

– **[RQ-4.] Can SSPCatcher flag unlabeled patches in the wild?**
  In this research question, we go beyond in-the-lab experiments and propose to assess the performance of SSPCatcher on unseen samples. To that end we propose to split the whole collected dataset based on timeline (instead of the classical ten-fold cross validation). SSPCatcher is trained on all samples except from the last year, and tested only on the last year's data, following experimental procedure by Allix et al. (Allix et al. 2015). We consider the predictions of SSPCatcher on the unlabeled patches in the test set and manually confirm whether the prediction is correct.

5.1 RQ1: Effectiveness of SSPCatcher

We perform binary classification experiments to assess the performance of classifiers in discriminating between security patches (positive class) and non-security patches (negative class). We remind that, as illustrated in Fig. 4, the non-security patches consist in the pure bug-fix patches and code-enhancement patches. These experiments, similarly to past studies (Sabetta and Bezzi 2018; Zhou and Sharma 2017; Tian et al. 2012), report performance based on the ground-truth data (i.e., unlabeled patches are not considered to compute the performance score).

Our first experiment investigates the performance of the Co-Training approach when varying the size of the unlabeled dataset in a uniform programming language environment (C).

In this experiment, we randomly split the labeled patch sets into two equal size subsets: one subset is used in conjunction with the unlabeled dataset for the Co-Training, while the other is used for testing. Precision, Recall, and Accuracy are computed based on the test set. Figure 9 presents the results, showing precision measurements above 90%, and recall measurements between 74% and 91%. We do not show evaluation graphs for OpenSSL dataset and Secbench since this dataset included only 436 unlabeled patches. With this

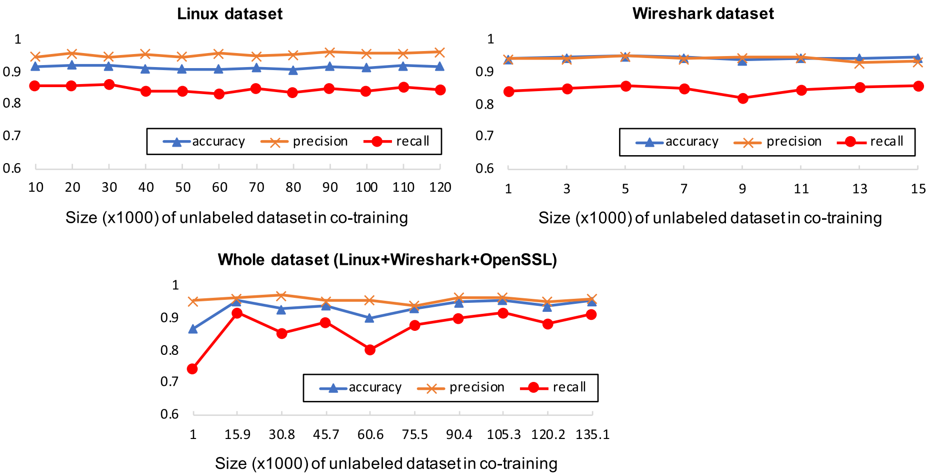


Fig. 9 Precision, Recall and Accuracy metrics in benchmark evaluation with varying sizes for the unlabeled dataset

quantity of unlabeled data, SSPCATCHER yields with OpenSSL the lowest Precision metrics at 74%, but the highest Recall at 93%. About the Secbench dataset, we do not consider it in this experiment because of the mixed nature of the programming languages used. We note that when using C-projects dataset (including Linux, OpenSSL, and Wireshark) the performance remains high. The best performing state-of-the-art approach in the literature for identifying security-relevant commits has reported Precision and Recall metrics at 80% and 43% respectively (Sabetta and Bezzi 2018). Tian et al. have also reported F1-Measure performance around 70% for identifying bug fixing commits (Tian et al. 2012), while the F1-measure performance of SSPCATCHER is 89% on average.

In contrast with OpenSSL, Wireshark, and Linux datasets which represent only samples written in the same programming language (C), the Secbench dataset includes projects whose code is written in various programming languages. Thus, with Secbench we evaluate the possibility of using our feature set and the produced model to predict on any type of project. The results are lower when we consider commits in any project (irrespective of the programming language), but the results are higher (precision: 93%, recall: 89% F1 score: 90%) when we only focus on predicting commits on C files. This (better) performance on C files is expected given that our feature set is partly inspired from the bug-fixing feature set proposed by Tian et al. (Tian et al. 2012) who focused on the C programming language.

Our second experiment estimates the time consumption of the classification approach to ensure that this approach can be executed in a reasonable time. We then evaluate here the time needed for the two classifiers used in the co-training algorithm to label the whole unlabeled dataset. The experiments were done with a computer with these descriptions:

- MacOS: version 10.14.6
- Processor: 2,4 GHz intel core i9
- Memory: 32 GB 2400 MHz-DDR4
- Graphics: Intel UHD Graphics 630 1536 MB

The time value was obtained with the time() function of the standard python library and the value was 125 s for the whole set of unlabeled patches

RQ1►SSPCATCHER (Co-Training + feature set) yields a highly accurate classifier for classifying patches with respect to whether they are security-relevant or not. ◄

5.2 RQ2: Cross-project Evaluation

In the wild of software development projects, as reflected by the case of OpenSSL, there can be limitations in the available labeled data. Thus, it could be beneficial if practitioners can train a model by leveraging data from another project and still obtain reasonable classification performance on a distinct target project. We investigate this possibility on our datasets considering firstly projects that are written in the same programming language (C). Secondly, we consider projects that are written in a mixed programming language (C).

5.2.1 Cross-project Classification on C-projects Dataset

Table 5 shows the classification performance results, in terms of Recall and Precision, when training on one project and applying the model to another. We note that training on Wireshark data yields reasonable (although not optimal) performance on OpenSSL patches, while training on OpenSSL interestingly offers high performance on Linux patches. In both

Table 5 Cross-project classification on projects using programming language C

			Training on
Testing on	OpenSSL precision/recall	Wireshark precision/recall	Linux precision/recall
OpenSSL	(0.93 /0.94)	0.71 / 0.48	0.42 / 0.88
Wireshark	0.53 / 0.88	(0.93 / 0.85)	0.50 / 0.95
Linux	0.89 / 0.78	0.45 / 0.93	(0.95 / 0.84)

cases, the converse is not true. Variations in cross-project performances may be explained by factors such as coding styles differences, code base size, or different security patching policies among projects. Future work will investigate the effects of these factors.

5.2.2 Cross-project Classification on Projects Using Mixed Programming Languages

Table 6 shows the classification performance results, in terms of Recall and Precision, when training on one project and applying the model to another. We first consider the top five projects in Secbench dataset that are written in mixed programming languages. We retain **Rails**(95.4% Ruby), **Php-src** (23.8% php), Mantisbt, **Curl** (7.5% php), **Server** (61.5% php), **Mantisbt** (76.9% php). To these projects, we add the three projects (Linux, OpenSSL, Wireshark) used in section 4.2.1. In particular, we note that training on OpenSSL data yields reasonable performance on Php-src patches, while training on Wireshark offers relatively high performance on Rails patches. Conversely, neither applies. The relatively weak results of this cross-project experiment can be explained by the mixed nature of the projects' programming languages. However, these experiments show that SSPCatcher allows us to classify with relatively acceptable results given the difficulty of the task.

Table 7 illustrates the classification performance, considering Recall and Accuracy when applying the model to all other projects after training on one project. We consider eight projects: Linux, Wireshark, OpenSSL, Curl, Mantisbt, Php-src, Server, and Rails. These projects are the result of adding the top five projects from the Secbench dataset and the three projects obtained from the Jimenez et al. framework. The principle is to train on one project in the batch and predict on all other projects. These experiments allow us to show that training on Linux data yields medium performance on the other patches.

Table 6 Cross-project classification on projects using mixed programming language

		Training on		
Testing on	OpenSSL precision/recall	Wireshark precision/recall	Linux precision/recall	
Rails	0.50 / 0.29	0.60 / 0.44	0.50 / 0.30	
	Curl	0.51 / 0.31	0.52 / 0.75	0.46 / 0.46
	Mantisbt	0.53 / 0.43	0.50 / 0.38	0.56 / 0.36
	php-src	0.77 / 0.68	0.50 / 0.62	0.51/0.51
	Server	0.49 / 0.46	0.57/0.72	0.47/0.44

Table 7 Cross-project classification on projects using mixed programming language: "train on one and predict on all"

	Training without						
	OpenSSL	Wireshark	Linux	rails	Curl	Php-src	Mantisbt	Server
Testing on all	0.51/0.58	0.51/0.58	0.56/0.59	0.50/0.60	0.51/0.51	0.36/0.18	0.54/0.49	0.43/0.13

Table 8 Comparison of F-Measure metrics

	OpenSSL	Wireshark	Linux	Secbench	Whole data
Our Approach	0.93	0.89	0.94	0.76	0.83
Sabetta & Bezzi (Sabetta and Bezzi 2018)	0.45	0.45	0.67	0.44	0.57

RQ2►Cross-project classification can yield comparatively good performance in some cases of combinations, such as when training on OpenSSL to classify Linux patches.◄

5.3 RQ3: How does SSPCatcher Compare Against the State-of-the-art?

While we report a F-Measure performance of around 90%, the most recent state-of-the-art on security commit classification (i.e., (Sabetta and Bezzi 2018)) reports performance metrics around 55%. Our experiments however are performed on different datasets because the dataset used by Sabetta & Bezzi was not made available. Thus, we first replicate the essential components of the best performing approach in their work (Sabetta and Bezzi 2018) (i.e., SVM bi-classification with bag-of-words features of code and log), and can therefore compare[5] their approach and ours in Table 8.

RQ3►Our Co-Training approach outperforms the state-of-the-art in the identification of security-relevant commits.◄

The second experiment assesses the contribution of the feature set on the one hand, and of the choice of Co-Training as learning algorithm on the other hand. We replicate the SVM binary classifier proposed by Sabetta and Bezzi (Sabetta and Bezzi 2018) and apply it on our labeled patches. We also build a similar classifier, however using our own feature set.

We perform 10-fold cross validations for all classifiers and evaluate the performance of the classifier in identifying labeled security patches in the whole dataset. Results in Table 9 indicate that our feature set is more effective than those used by the state-of-the-art, while the Co-Training semi-supervised model is more effective than the classical binary classification model.

Given that our code-fix features overlap with features used by Tian et al. (Tian et al. 2012) for classifying bug fix patches, we present performance comparisons with the different feature sets. Results in Table 10 confirm that our extended feature set (with vulnerability-sensitive features) allows to increase performance by up to 26 percentage points. The performance differences between projects further confirm that the features of Tian et al. (Tian et al. 2012) are indeed very specific to Linux.

5.4 RQ4: Can SSPCatcher Flag Unlabeled Patches in the Wild?

In these experiments, we only consider the C-projects dataset(Linux, OpenSSL, and Wireshark).

Performance computation presented in previous subsections are based on cross validations where training and test data are randomly sampled. Such validations often suffer from

[5] Note that the recorded performance of the replicated approach on our dataset is in line with the performance reported by the authors in their paper (Sabetta and Bezzi 2018).

Table 9 Importance* of Classification method and feature set

	Precision	Recall	F1-measure
SVM binary classification (*with features of Sabetta & Bezzi* (Sabetta and Bezzi 2018))	0.44	0.45	0.44
SVM binary classification (*with our feature set*)	0.87	0.38	0.53
Co-Training + SVM (*with our feature set*)	0.85	0.81	0.83

*Performance metrics are for classifying 'security patches'. Due to space limitation, we refer the reader to the replication package for all evaluation data.

the data leakage problem (Ribeiro et al. 2016), which leads to the construction overly optimistic models that are practically useless and cannot be used in production. For example, in our case, data leakage can happen if the training set includes security patches that should actually only be available in the testing set (i.e., we would be learning from the future). We thus propose to divide our whole dataset, with patches from all projects, following the commits timeline, and select the last year's commits as test set. The previous commits are all used as training set. We then train a classifier using SSPCATCHER approach and apply it to the 475 commits of the test set. To ensure confidence in our conclusions, we focus on automatically measuring the performance based only on the last year patches for which the labels are known (i.e., the patches coming from the security patches dataset, the pure bug fix patches dataset, and the code enhancement patches dataset as illustrated in Fig. 4). Overall, we recorded precision and recall metrics of 0.64 and 0.67 respectively.

In a final experiment, we propose to audit 10 unlabeled patches flagged as security patches by a Co-Training classifier built by learning on the whole data. We focus on the top-10 unlabeled patches that are flagged by the classifier with the highest prediction probabilities. Two authors manually cross-examine the patches to assess the plausibility of the classification. We further solicit the opinion of two researchers (who are not authors of this paper) to audit the flagged security patches. For each presented patch, patch auditors must indicate whether yes or no they accept it as a security patch. Auditors must further indicate in a Likert scale to what extent the associated details on the features with highest InfoGain was relevant to the reason why they would confirm the classification. Among the 10 considered patches, 5 happen to be for Linux, 3 for OpenSSL and 2 are for Wireshark.

We compute $Precision@10$ following the formula :

$$Precision@k = \frac{1}{\#auditors} \sum_{i=1}^{\#auditors} \frac{\#confirmed\ patches}{k}$$

Ideally, a security patch should be confirmed experimentally by attempting an exploit. Nevertheless, this requires extremely high expertise for our subjects (Linux, OpenSSL and Wireshark) and significant time. Instead, and to limit experimenter bias, auditors were asked to check at least whether issues fixed by the patches have similar occurrences in line with known potential vulnerabilities. For example, one of the flagged security patches is "fixing a memory leak" in OpenSSL (cf. commit `9ee1c83`). The literature indicates this as a known category of vulnerability which is easily exploitable (Szekeres et al. 2013).

At the end of the auditing process, we record a Precision@10 metric of 0.55. Although this performance *in the wild* may seem limited, it is actually comparable to the performance

Table 10 F1-Measure Comparison: Our features vs features in (Tian et al. 2012)*

	OpenSSL	Wireshark	Linux	Secbench	Whole data
Co-Training + SVM (*with our feature set*)	0.93	0.89	0.94	0.76	0.83
Co-Training + SVM (*with feature set of Tian & al.* (Tian et al. 2012))	0.65	0.71	0.96	56	0.67
SVM binary classification (*with features of Tian & al.* (Tian et al. 2012))	0.69	0.77	0.99	0.48	0.61

This comparison serves to assess the impact of our security-sensitive features

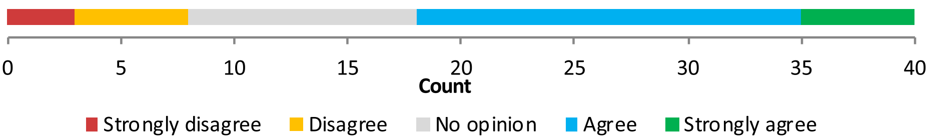


Fig. 10 Do the highlighted features provide relevant hints for manual review of flagged patches?

recorded *in the lab* by the state-of-the-art, and is a very significant improvement over a random classifier that, given the small proportion of security patches (Ponta et al. 2019), would almost always be wrong.

Figure 10 indicates the distribution of the Likert scale values for the satisfaction rates indicated by the auditors for the usefulness of leveraging the features with highest InfoGain to confirm the classification.

RQ4►The approach helps to catch some silent security patches. Features with high InfoGain can be useful to guide auditors.◄

6 Insights, Threats to Validity, and Limitations

6.1 Discussion

The Deep learning panacea. Co-attention is an interesting deep-learning approach that could actually be relevant for accurately classifying code changes. Unfortunately, neural network based approaches have one constraint and one limitation in the context of our work: (1) they require large datasets to train (when pre-trained models are unavailable as is the case here). Datasets on security patches are not only scarce but also highly imbalanced; (2) they are generally not sufficiently explainable, which is a strong limitation as we need a trade-off between accuracy and interpretability of results (i.e., to provide hints to the analyst as to why the patch is predicted as being security-related). Our focus in this work was to deal with dataset imbalance, hence we did not aim for a deep learning approach. Future work could investigate the possibility of leveraging models that were pre-trained for bug fixes and fine-tune them for security fix detection.

Excluded features. During feature extraction, we have opted to ignore information related to the author of a commit or the file where the commit occurs, as such information can lead to an overfitted model. Furthermore, we expect our classifier to be useful across projects, and thus we should not include project-specific features. In contrast, although we found that some selected features have, individually, little discriminative power, we keep them for the learning as, in combinations, they may help yield efficient classifiers.

Benefit of unlabeled data. Generally, labeling is expensive and time-consuming, while unlabeled data is often freely available on large scales. Our Co-Training approach successfully leverages such data and turns a weakness in our problem setting into an essential part of the solution. Furthermore, it should be noted that, by construction, our dataset is highly imbalanced. Although some data balancing techniques (e.g., SMOTE (Chawla et al. 2002)) could be used, we chose to focus our experiments on validating the suitability of our feature set with the Co-Training for semi-supervised learning. Future work could investigate other optimizations.

6.2 SSPCatcher and the Practice of Software Development

SSPCatcher was designed to be readily integrated into a real-world pipeline of collaborative software development. First, in terms of inputs, we consider information that is readily available and relevant for the purpose of security patch prediction. Second, the features for representing patch samples are extracted only based on the sample patch, without leveraging external information. This design choice contributes to reducing the computation time: simple features are considered based on patch information, instead of building on complex code features such as cyclomatic complexity metrics. Third, we envision SSPCatcher to be deployed in a typical code management system. In such systems which implement pre-commit tasks such as with "Git hook", it is possible to perform a set of processing actions on a commit before adding it to the repository. Our approach is expected to be leveraged in such scenarios where a security relevance warning can be made before the commit is made publicly visible or even accepted.

On the other hand, SSPCatcher was developed in python and written in the form of a library so that it can be easily integrated into an existing pipeline. It could directly incorporate inputs from a pipeline and produce the necessary outputs.

Finally, we note that SSPCatcher performs very well on patches applied to C program files but also reasonably well on patches for other programming languages. This opens the door to the identification of security patches in large projects where code from different programming languages co-exist

6.3 Threats to Validity

As with most empirical studies, our study carries some threats to validity. An important threat to *internal validity* in our study is the experimenter bias when we personally labeled code enhancement commits. However, we have indicated the systematic steps for making the decisions in order to minimize bias. As a threat to *external validity*, the generalizability of the results can be questioned since we could only manually assess a small sample set of flagged unlabeled patches. Given that our ranking is based on prediction probability, assessment of top results is highly indicative of the approach performance. Finally, threats to *construct validity* concern our evaluation criteria. Nevertheless, we used standard metrics such as Precision, Recall, F-Measure, and Likert scale to evaluate the effectiveness of the SSPCatcher approach.

6.4 Limitations

Our approach exhibits a number of limitations in terms of:

- *Programming language support:* SSPCatcher applies to code changes, i.e., diffs. While we do not require any programming language-specific parser to extract feature values, our feature engineering is partly inspired from the bug-fix identification task for C programs by Tian et al (Tian et al. 2012). Consequently, and as shown by the performance results on Secbench, our approach works best on C language. Nevertheless, the results that we obtain overall, including other programming languages, remain acceptable (i.e., largely over 50% Precision score).
- *Expressiveness and interpretability of the feature set:* our feature set is limited to our manually engineering effort based on 300 vulnerability fixes. We acknowledge the limitation that this feature set is not exhaustive and that they remain high-level hints that

cannot systematically be used to explain the security relevance. This later limitation, which we share with the state of the art, makes it necessary to rely on human expertise to document the security aspect of the patch.

– *Sensitiveness to project types:* Our experimental results show that SSPCATCHER performance differs across projects. The learned model is further influenced by coding styles, dataset size, and security patching policies which affect the inter-project application. Due to limitations in the collected dataset size, the produced model may not be used in the wild without re-training.
– *Exploitation of commit metadata:* SSPCATCHER does not exploit commit metadata, which is a relevant source of information for learning a more accurate model for security patch identification. We have made such a design choice by considering that some metadata, such as the commit author, may lead to overfitting due to the fact that some projects have designated security maintainers.

6.5 Future Work

We plan to apply SSPCATCHER to security patch identification to Java projects after collecting the necessary training data (e.g., from (Ponta et al. 2019)). Such a classifier could then help the open source community report more vulnerabilities and their patches (those address vulnerabilities) to security advisories. Besides SVM, which was used to ensure tractable performance comparisons with the state-of-the-art, we will investigate some Boosting algorithms. Finally, we will consider adapting other security-sensitive features (e.g., stall ratio, coupling propagation, etc. from (Chowdhury et al. 2008)) to the cases of code differences to assess their impact on the classification performance.

7 Related Work

The identification of security-relevant commits has applications for various stakeholders in software development. The literature includes a number of related works that we summarize in this section.

Our work is related to several research directions in the literature, most notably studies on 1) Security commit identification, 2) vulnerability management and 3) change analysis.

7.1 Security Commit Identification

Recently, researchers from the security industry (Zhou and Sharma 2017; Sabetta and Bezzi 2018) (from SourceClear, Inc., and SAP respectively)

have presented early investigations on the prediction of security issues in relation with commit changes. Zhou and Asankhaya (Zhou and Sharma 2017) focus on commit logs, commit metadata, and associated bug reports, and leverage regular expressions to identify features for predicting security-relevant commits. The authors use embedding ($word2vec$) to learn the features, which leads to an opaque decision-making system (Pontin 2018; Knight 2017) when it comes to guiding a security analyst in his/her auditing tasks. The approach is further limited since experimental data show that not all fixes are linked to reported bugs, and not all developers know (or want to disclose in logs) that they are fixing vulnerabilities. Sabetta and Bezzi (Sabetta and Bezzi 2018) improve over the work of Zhou and Asankhaya by considering code changes as well. Their approach is fully-supervised (thus, assuming that the labeled dataset is perfect and sufficient).

Closely related work in identifying security patches is contributed so far by the industry. Nevertheless, various academic works rely on scarce data to train machine learning models for vulnerability detection, exploitation, or patching. Our work will enable the amplification of such datasets (beyond the disclosed security patches), to include silent fixes, thus increasing the coverage and reliability of the state-of-the-art.

7.2 Vulnerability Management

Recently, the topic of Autonomous Cyber Reasoning Systems (Ji et al. 2018) has attracted extensive attention from both industry and academia, with the development of new techniques to automate the detection, exploitation, and patching of software vulnerabilities in a scalable and cost-effective way. Static analysis approaches such as the code property graph by Yamaguchi et al. (Yamaguchi et al. 2014b) require a built model of vulnerabilities based on expert knowledge. Dynamic approaches leverage fuzzing to test a software with intentionally invalid inputs to discover unknown vulnerabilities (Godefroid et al. 2008; Sutton et al. 2007), or exploit taint analyses to track marked information flow through a program as it executes in order to detect most types of vulnerabilities (Newsome and Song 2005), including leaks (Li et al. 2015). Such approaches, although very precise, are known to be expensive, and achieve a limited code coverage (Brooks 2017). Recently, researchers have been investigating concolic analysis (Cadar et al. 2008) tools for software security. Mayhem (Cha et al. 2012) is an example of such a system.

The literature includes a number of approaches that use software metrics to highlight code regions that are more likely to contain vulnerabilities. Metrics such as code churn and code complexity along with organizational measures (e.g., team size, working hours) allowed to achieve high precision in a large scale empirical study of vulnerabilities in Windows Vista (Zimmermann et al. 2010). However, Jay et al. (Jay et al. 2009) have warned that many of these metrics may be highly correlated with lines of code, suggesting that such detection techniques are not helpful in reducing the amount of code to read to discover the actual vulnerable piece of code.

Nowadays, researchers are exploring machine learning techniques to improve the performance of automatic software vulnerability detection, exploitation, and patching (Ji et al. 2018; Li et al. 2018). For example, Scandariato et al. (Scandariato et al. 2014) have trained a classifier on textual features extracted from source code to determine vulnerable software components. Xiaoning Du et al. (Du et al. 2019) also propose an approach named LEOPARD that uses code metrics features for the identification of vulnerable functions in projects. Their feature extraction process was mainly based on code complexity instead of Yang Xiao et al. (Xiao et al. 2020) work that used function signatures. These approaches yield good predictions results with several machine learning algorithms. However, it's challenging to train automatic learning models without an available and suitable vulnerable code data set. Jimenez et al. proposed VulData7, an extensible framework and dataset of real vulnerabilities, automatically collected from software archives. VulData7 retrieves patches for 1,600 of the 2,800 reported vulnerabilities from the four systems available on GitHub for analysis and predictive vulnerability studies.

Several unsupervised learning approaches have been presented to assist in the discovery of vulnerabilities (Yamaguchi et al. 2013; Chang et al. 2008). We differ from these approaches both in terms of objectives and in the use of a combination of features from code and metadata. With respect to feature learning, new deep learning-based approaches

(Li et al. 2018) are being proposed since they do not require expert intervention to generate features. The models are however mostly opaque (Pontin 2018) for analysts who require explainability of decisions during audits. Capturing code semantics and properties for feature engineering is one of the most effective approaches to unsupervised learning (Yamaguchi et al. 2014a). Yaqin Zhou et al. (Zhou et al. 2019) propose an automatic feature extraction approach based on graph properties for accurate predictions of vulnerabilities. Finally, it is noteworthy that the industry is starting to share with the research community some datasets yielded by manual curation efforts of security experts (Ponta et al. 2019).

7.3 Change Analysis

Software change is a fundamental ingredient of software maintenance (Li et al. 2013). Software changes are often applied to comply to new requirements, to fix bugs, to address change requests, and so on. When such changes are made, inevitably, some expected and unexpected effects may ensue, even beyond the software code. Software change impact analysis has been studied in the literature as a collection of techniques for determining the effects of the proposed changes on other parts of the software (Arnold 1996).

Researchers have further investigated a number of prediction approaches related to software changes, including by analysing co-change patterns to predict source code changes (Ying et al. 2004). Closely related to ours is the work of Tian et al. (Tian et al. 2012) who propose a learning model to identify Linux bug fixing patches. The motivation of their work is to improve the propagation of fixes upwards the mainline tree.

SSPCatcher, however, is substantially different regarding: (1) *Objective*:. (Tian et al. 2012) targets Linux development, and identifies bug fixes. We are focused on security patches. (2) *Method*: (Tian et al. 2012) leverages the classification algorithm named Learning from Positive and Unlabeled Examples (LPU) (Li and Liu 2003). In contrast, we explore Co-Training which requires two independent views of the data. We also include a more security-sensitive set of features. We explore a combination of latent (e.g., #sizeof) and explicit (e.g., keyword) features. (3) *Evaluation*: (Tian et al. 2012) was evaluated against a keyword-based approach. We evaluate against the state-of-the-art and based on manual audit. All data is released and made available for replication. Following up on the work of Tian et al. (Tian et al. 2012), Hoang et al. have proposed a deep learning-based tool for classifying bug fix commits (Hoang et al. 2018).

Security analysis of commits has been investigated by Perl et al. (Perl et al. 2015) who presented VCCFinder for flagging suspicious commits by using an SVM classifier. In contrast to our work, VCCFinder aims at identifying vulnerability-introducing changes, while, conversely, we aim for identifying those changes that fix vulnerabilities.

8 Conclusion

We have investigated the problem of identifying security patches, i.e., patches that address security issues in a code base. Our study explores a Co-Training approach which we demonstrate to be effective. Concretely, we proposed to consider the commit log and the code change diff as two independent views of a patch. The Co-Training algorithm then iteratively converges on a classifier that outperforms the state-of-the-art. We further show experimentally that this performance is due to the suitability of our feature set as well as the effectiveness of the Co-Training algorithm. Finally, experiments on unlabeled patches show that our model can help uncover silent fixes of vulnerabilities.

Data Availability We provide the dataset, scripts, and results as a replication package at http://github.com/vulnCatcher/vulnCatcher. Our implementation of SSPCATCHER is further open sourced for the entire research to build on our results.

References

Allix K, Bissyandé TF, Klein J, Le Traon Y (2015) Are your training datasets yet relevant? In: International Symposium on Engineering Secure Software and Systems. Springer, pp 51–67

Arnold RS (1996) Software change impact analysis. IEEE Computer Society Press, California

Berr J (2017) "wannacry" ransomware attack losses could reach $4 billion. https://www.cbsnews.com/news/wannacry-ransomware-attacks-wannacry-virus-losses/, Available: August 2018

Bissyande TF, Thung F, Wang S, Lo D, Jiang L, Reveillere L (2013) Empirical evaluation of bug linking. In: Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on. IEEE, pp 89–98

Blum A, Mitchell T (1998) Combining labeled and unlabeled data with co-training. In: Proceedings of the eleventh annual conference on Computational learning theory. ACM, pp 92–100

Brooks TN (2017) Survey of automated vulnerability detection and exploit generation techniques in cyber reasoning systems. arXiv preprint arXiv:1702.06162

Cadar C, Dunbar D, Engler DR et al (2008) Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, vol 8, pp 209–224

Cha SK, Avgerinos T, Rebert A, Brumley D (2012) Unleashing mayhem on binary code. In: Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, pp 380–394

Chang R-Y, Podgurski A, Yang J (2008) Discovering neglected conditions in software by mining dependence graphs. IEEE Trans Softw Eng 34(5):579–596

Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) Smote: synthetic minority over-sampling technique. Journal of artificial intelligence research 16:321–357

Chowdhury I, Chan B, Zulkernine M (2008) Security metrics for source code structures. In: Proceedings of the fourth international workshop on Software engineering for secure systems. ACM, pp 57–64

Du X, Chen B, Li Y, Guo J, Zhou Y, Liu Y, Jiang Y (2019) Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 60–71

Godefroid P, Levin MY, Molnar DA et al (2008) Automated whitebox fuzz testing. In: NDSS, vol 8, pp 151–166

Hempstalk K, Frank E (2008) Discriminating against new classes: One-class versus multi-class classification. In: Australasian Joint Conference on Artificial Intelligence. Springer, pp 325–336

Hoang T, Lawall J, Oentaryo RJ, Tian Y, Lo D (2018) Patchnet: A tool for deep patch classification. In: Tool Demonstrations of International Conference on Software Engineering

Jay G, Hale JE, Smith RK, Hale DP, Kraft NA, Ward C (2009) Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship. JSEA 2(3):137–143

Ji T, Wu Y, Wang C, Zhang X, Wang Z (2018) The coming era of alphahacking?: A survey of automatic software vulnerability detection, exploitation and patching techniques. In: 2018 IEEE Third International Conference on Data Science in Cyberspace (DSC). IEEE

Jimenez M, Le Traon Y, Papadakis M (2018) Enabling the continous analysis of security vulnerabilities with vuldata7. In: IEEE International Working Conference on Source Code Analysis and Manipulation

Knight W (2017) The dark secret at the heart of ai. MIT Technology Review https://www.technologyreview.com/s/604087/the-dark-secret-at-the-heart-of-ai/

Koyuncu A, Bissyandé TF, Kim D, Klein J, Monperrus M, Le Traon Y (2017) Impact of tool support in patch construction. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, pp 237–248

Krogel M-A, Scheffer T (2004) Multi-relational learning, text mining, and semi-supervised learning for functional genomics. Mach Learn 57(1-2):61–81

Li B, Sun X, Leung H, Zhang S (2013) A survey of code-based change impact analysis techniques. Softw Test Verification Reliab 23(8):613–646

Li L, Bartel A, Bissyandé TF, Klein J, Le Traon Y, Arzt S, Rasthofer S, Bodden E, Octeau D, McDaniel P (2015) Iccta: Detecting inter-component privacy leaks in android apps. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, pp 280–291

Li X, Liu B (2003) Learning to classify text using positive and unlabeled data. In: IJCAI. ACM, pp 587–592

Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, Deng Z, Zhong Y (2018) Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv:1801.01681
Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. The annals of mathematical statistics, pp. 50–60
Newsome J, Song DX (2005) Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In: NDSS, vol 5. Citeseer, pp 3–4
Nguyen AT, Nguyen TT, Nguyen HA, Nguyen TN (2012) Multi-layered approach for recovering links between bug reports and fixes. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM, p 63
Nigam K, Ghani R (2000) Analyzing the effectiveness and applicability of co-training. In: Proceedings of the ninth international conference on Information and knowledge management. ACM, pp 86–93
NIST (2018) National vulnerability database. https://nvd.nist.gov
Perl H, Dechand S, Smith M, Arp D, Yamaguchi F, Rieck K, Fahl S, Acar Y (2015) Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, pp 426–437
Ponta SE, Plate H, Sabetta A, Bezzi M, Dangremont C (2019) A manually-curated dataset of fixes to vulnerabilities of open-source software. arXiv preprint arXiv:1902.02595
Pontin J (2018) Greedy, brittle, opaque, and shallow: The downsides to deep learning. https://www.wired.com/story/greedy-brittle-opaque-and-shallow-the-downsides-to-deep-learning/
Porter MF (1980) An algorithm for suffix stripping. Program 14(3):130–137
Reis S, Abreu R (2017) SECBENCH: A database of real security vulnerabilities. In: Jaatun MG, Cruzes DS (eds) Proceedings of the International Workshop on Secure Software Engineering in DevOps and Agile Development co-located with the 22nd European Symposium on Research in Computer Security (ESORICS 2017), Oslo, Norway, September 14, 2017. CEUR Workshop Proceedings, vol 1977, CEUR-WS.org, pp 69–85
Ribeiro MT, Singh S, Guestrin C (2016) Why should i trust you?: Explaining the predictions of any classifier. In: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 1135–1144
Sabetta A, Bezzi M (2018) A practical approach to the automatic classification of security-relevant commits. In: 34th IEEE International Conference on Software Maintenance and Evolution (ICSME)
Scandariato R, Walden J, Hovsepyan A, Joosen W (2014) Predicting vulnerable software components via text mining. IEEE Trans Softw Eng 40(10):993–1006
Snyk.io (2017) The state of open-source security. https://snyk.io/stateofossecurity/pdf/The%20State%20of%20Open%20S%ource.pdf, Available: August 2018
Sutton M, Greene A, Amini P (2007) Fuzzing: brute force vulnerability discovery. Pearson Education, California
Szekeres L, Payer M, Wei T, Song D (2013) Sok: Eternal war in memory. In: Security and Privacy (SP), 2013 IEEE Symposium on. IEEE, pp 48–62
Tian Y, Lawall J, Lo D (2012) Identifying linux bug fixing patches. In: Proceedings of the 34th International Conference on Software Engineering. IEEE Press, pp 386–396
Trend Micro (2017) Patching problems and how to solve them. https://www.trendmicro.com/vinfo/us/security/news/vulnerabilities-and-exploits/patching-problems-and-how-to-solve-them, Available: August 2018
van Rossum G (2008) Origin of bdfl. All Things Pythonic Weblogs. http://www.artima.com/weblogs/viewpost.jsp
Vapnik V (2013) The nature of statistical learning theory. Springer, New York
Wu R, Zhang H, Kim S, Cheung S-C (2011) Relink: recovering links between bugs and changes. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM, pp 15–25
Xiao Y, Chen B, Yu C, Xu Z, Yuan Z, Li F, Liu B, Liu Y, Huo W, Zou W, Shi W (2020) Mvp: Detecting vulnerabilities using patch-enhanced vulnerability signatures. In: USENIX Security Symposium
Yamaguchi F, Golde N, Arp D, Rieck K (2014) Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy, pp 590–604
Yamaguchi F, Golde N, Arp D, Rieck K (2014) Modeling and discovering vulnerabilities with code property graphs. In: Security and Privacy (SP), 2014 IEEE Symposium on. IEEE, pp 590–604
Yamaguchi F, Wressnegger C, Gascon H, Rieck K (2013) Chucky: Exposing missing checks in source code for vulnerability discovery. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, pp 499–510
Ying AnnieTT, Murphy GC, Ng R, Chu-Carroll MC (2004) Predicting source code changes by mining change history. IEEE Trans Softw Eng 30(9):574–586

Zhou Y, Liu S, Siow J, Du X, Liu Y (2019) Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: NeurIPS
Zhou Y, Sharma A (2017) Automated identification of security issues from commit messages and bug reports. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, pp 914–919
Zimmermann T, Nagappan N, Williams L (2010) 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. IEEE, pp 421–428

Affiliations

Arthur D. Sawadogo[1] · Tegawendé F. Bissyandé[2] · Naouel Moha[1] · Kevin Allix[2] · Jacques Klein[2] · Li Li[3] · Yves Le Traon[2]

Tegawendé F. Bissyandé
tegawende.f.bissyande@uni.lu

Naouel Moha
moha.naouel@uqam.ca

Kevin Allix
kevin.allix@uni.lu

Jacques Klein
jacques.klein@uni.lu

Li Li
li.li@monash.edu

Yves Le Traon
yves.le.traon@uni.lu

1 Université du Québec à Montréal, Montréal, QC Canada
2 SnT, University of Luxembourg, 2 Av. de l'Universite, 4365 Esch-sur-Alzette, Luxembourg
3 Monash University, Melbourne, Victoria, Australia