UNIVERSITÉ DU QUÉBEC À MONTRÉAL

TOWARDS OVERCOMING ZERO-DAY VULNERABILITIES IN OPEN SOURCE SOFTWARE : AN AUTOMATIC APPROACH FOR SECURITY PATCHES IDENTIFICATION

DISSERTATION

PRESENTED

AS PARTIAL FULFILLMENT

OF THE DOCTORATE IN COMPUTER SCIENCE

BY

DELWENDE DONALD ARTHUR SAWADOGO

AUGUST 2022

UNIVERSITÉ DU QUÉBEC À MONTRÉAL

VERS UNE RÉDUCTION DES VULNÉRABILITÉS "ZERO-JOUR" À TRAVERS DES APPROCHES AUTOMATIQUES DE GESTION DES CORRECTIFS DE SÉCURITÉ

THÈSE

PRÉSENTÉE

COMME EXIGENCE PARTIELLE

DU DOCTORAT EN INFORMATIQUE

PAR

DELWENDE DONALD ARTHUR SAWADOGO

AOÛT 2022

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the support of many people who, in one way or another, have contributed and extended their precious knowledge and experience in my PhD studies. It is my pleasure to express my gratitude to them.

First of all, I would like to express my deepest thanks to my supervisors Prof. Naouel Moha and Prof. Tegawendé Bissyandé, who have given me this great opportunity to come across continents to pursue my doctoral degree. They showed me passion, opportunity, intuition, and the future. I would like to thank them for their guidance, passion for research, and social support in my difficult moments. Since then, working in this field has been just joyful for me. They have taught me how to perform research, write technical papers, and conduct fascinating presentations. Their dedicated guidance has made my PhD journey a fruitful and fulfilling experience. I am pleased about the friendship we have built up during the years.

Second, I am equally grateful to Prof. Jacques Klein (University of Luxembourg), who did not spare any effort to assist me during all the steps of this thesis.

Third, I would like to extend my thanks to all my co-authors, including Prof. Yves Le Traon, Dr. Kevin Alix, Timothé Riom, Abdoul-Kader Kaboré, and all the team members of TruX and SerVal at SnT (Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg) for their valuable discussions and collaborations. It was always passionate, insightful, and pleasant to converse with them.

I would like to also express my great thanks to all of my friends in Montreal and Luxembourg for our memorable moments.

Moreover, I would like to thanks my family, brothers, and sisters for always giving me the strength to achieve my goals, being present in all steps of my life, and giving me this unconditional love. May God bless you.

Finally, I am as ever indebted to God, who made all things possible. "Father, I thank you because you have heard me" (John 11 :41)

DEDICATION

To my parents, who unfortunately passed away before seeing this stage of my life. In particular to my mother, whom I thank for always having motivated and supported me in my life and especially during the beginning of this thesis. I am sure that from there you are proud of me.

Table des matières

LISTE DES FIGURES

RÉSUMÉ

Les attaques de sécurité logicielle peuvent avoir un impact considérable : elles peuvent porter atteinte à la vie privée par la fuite de données, entraîner des pertes financières par l'indisponibilité des services, corrompre l'intégrité de données sensibles, etc. De nombreux efforts sont déployés par les développeurs et les équipes de recherche pour réduire l'exposition des logiciels aux attaques de parties malveillantes. Les équipes de développement de logiciels propriétaires sont généralement très organisées, avec des revues de code régulières et des analyses statiques ainsi que des tests dynamiques continus. Dans le monde du logiciel libre, ces ressources sont rares et les procédures de contribution au code source sont ouvertes. Des vulnérabilités jour-zéro peuvent passer inaperçues. Une vulnérabilité de type jour-zéro est une vulnérabilité dans un système ou un dispositif qui a été divulguée, mais n'a pas encore été corrigée. Ce type de vulnérabilité peut rester inconnu des parties légitimes pendant de longues durées, augmentant ainsi les risques d'attaques. L'objectif de cette thèse est de proposer une approche générique et automatique utilisant des techniques d'apprentissage automatique pour détecter le plus tôt possible les vulnérabilités dans le code des logiciels libres en contribuant aux deux blocs suivants :

— *L'identification des commits corrigeant les vulnérabilités :* lorsqu'un changement de code (commit) est étiqueté comme étant pertinent pour la sécurité, c'est-à-dire comme corrigeant une vulnérabilité, les mainteneurs diffusent rapidement le changement, et les utilisateurs sont informés de la nécessité de mettre à jour la bibliothèque l'application. Malheureusement, certains changements pertinents pour la sécurité passent souvent inaperçus car ils représentent des correctifs silencieux de vulnérabilités. Nous proposons SSPCatcher, une approche basée sur le co-entraînement pour détecter les correctifs de sécurité (c'est-à-dire les correctifs qui corrigent le code vulnérable) dans le cadre d'un service de surveillance automatique des dépôts de code. En s'appuyant sur différentes classes de caractéristiques, nous montrons empiriquement qu'une telle automatisation est réalisable et peut donner une précision de plus de 80% dans l'identification des correctifs de sécurité, avec un rappel de plus de 80%. Au-delà d'une telle évaluation comparative avec des données de base qui démontre une amélioration par rapport à l'état de l'art, nous avons confirmé que SSPCatcher peut aider à capturer des correctifs de sécurité qui n'ont pas été signalés comme tels.

— *L'identification des commits introduisant les vulnérabilités :* la détection des vulnérabilités dans les logiciels est une course constante entre les équipes de développement et les attaquants potentiels. Tandis que de nombreuses approches statiques et dynamiques se sont concentrées sur l'analyse régulière du logiciel dans son intégralité, une direction de recherche récente s'est concentrée sur l'analyse des changements appliqués au code. Nous proposons dans cette partie une nouvelle approche pour identifier les commits contribuant à la vulnérabilité, basée sur une technique d'apprentissage semi-supervisée avec un ensemble de caractéristiques spécifiques. En outre, étant donné l'influence de VCCFinder (Perl *et al.*, 2015) dans cette direction de recherche, nous entreprenons une enquête sur ses performances en tant que système de pointe. À cette fin, nous proposons également une étude de réplication de l'approche d'apprentissage supervisé VCCFinder.

Ce document présente notre problématique, nos contributions ainsi que les travaux réalisés dans cette thèse.

Mots-clés : correctif de sécurité, vulnérabilités "jour zéro", attaque de sécurité logicielle, co-entraînement, vulnérabilité logicielle, apprentissage automatique, logiciel libre.

ABSTRACT

Attacks on software security can have a significant impact : they can damage privacy through data leakage, cause financial losses through unavailability of services, corrupt the integrity of sensitive data, etc. Software developers and teams are making many efforts to reduce software exposure to attacks by malicious parties. Proprietary software development teams are usually very organized, with regular code reviews, static analysis, and dynamic testing. In the open source world, these resources are scarce, and the procedures for contributing source code are more open. Zero-day vulnerabilities can go unnoticed. A zero-day vulnerability is a computer security flaw that the software or service provider is not yet aware of or that has not yet been patched. This type of vulnerability could easily go unnoticed by legitimate parties, thus increasing the risk of attacks. This thesis aims to propose a generic and automatic approach using machine learning techniques to detect vulnerabilities in open source software code as early as possible by contributing to the following two blocks :

— *Vulnerability-fixing patch identification :* when fixing 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. 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.

— *Vulnerability-introducing patch identification :*
Detecting vulnerabilities in software is a constant race between development teams and potential attackers. While many static and dynamic approaches have focused on regularly analyzing the software in its entirety, a recent research direction has focused on the analysis of changes that are applied to

the code. In this part, we design a new approach to identify vulnerability-contributing commits based on a semi-supervised learning technique with a specific feature set. In addition, given the influence of VCCFinder (Perl *et al.*, 2015) in this research direction, we undertake an investigation into its performance as a state-of-the-art system. To that end, we also propose a replication study on the VCCFinder supervised learning approach.

This document presents problems, contributions, and the work done in this thesis.

Keywords : security patches, zero-day vulnerabilities, security attacks, co-training, software vulnerabilities, machine learning, open source software.

CHAPITRE I

INTRODUCTION

Software is an essential part of our daily lives. Nowadays, we are witnessing the integration of software into every aspect of human activities, from simple mobile phones to vehicles, homes (e.g., Google Home), etc. They are also increasingly involved in more sensitive areas such as medicine, assistance for the elderly, natural disaster management (Catarci *et al.*, 2008), etc. Such software is developed by software companies (proprietary software) or may have been developed by companies or independent programmers who collaborate on the Internet and could publicly open the source code of their work (open source software). Once mainly leveraged by a few researchers, engineers, and other technology professionals, open source software has rapidly gained the interest and respect of information technology (IT) professionals in many industries as well as user confidence. This change can be explained by the openness of the code and the contribution to the development of such software. For example, any person or entity can adapt and customize the code of another given project. However, the use of such software raises concerns about support and assistance in case of bugs and the impact that modifying the code could have, such as the possibility of creating new bugs.

Detecting and fixing bugs in software is a priority activity for companies, as their presence in software directly impacts the user experience and, therefore, the credibility of the company. Moreover, if the bug introduces a security hole, the issue-level is even higher because it exposes users and/or the company to potential attacks.

Vulnerabilities are therefore the most sensitive category of bugs. Many efforts are made by companies and the open source communities to detect vulnerabilities in software and to quickly propose patches to avoid attacks. Detection tools such as VCCFinder (Perl *et al.*, 2015) and the Buffer Overflow Detection Tool (Larochelle et Evans, 2001) have been developed to detect vulnerabilities in open source software (OSS). However, despite the existence of these detection/correction tools, the efforts of companies and the open source communities, large-scale attacks are still being carried out (Berr, 2017). Recent successful attacks have focused on vulnerabilities that were not yet known by any of the stakeholders in the software projects (Farwell et Rohozinski, 2011). This type of vulnerabilities exploited by attackers constitutes the category of zero-day vulnerabilities (Bilge et Dumitraş, 2012). A zero-day vulnerability is a computer security flaw that the software or service provider is not yet aware of or that has not yet been patched.

For some years now, we have been witnessing sophisticated, excellently planned, organized, and executed attacks by attackers exploiting these zero-day vulnerabilities. These attacks affect many significant companies and thus leading to significant financial and other losses (Farwell et Rohozinski, 2011).

The overall objective of this thesis is to propose some comprehensive approaches using machine learning to reduce the presence of zero-day vulnerabilities in open source software.

1.1 Motivation

Attacks that exploit zero-day vulnerabilities are becoming increasingly common. They are the source of financial and technological damage. Moreover, we even see the creation of online markets for these vulnerabilities (Egelman *et al.*, 2013), which leads to the rapid expansion of exploitable vulnerabilities worldwide and creates an economy around the problem making the situation more complex. A famous example of zero-day vulnerability exploitation is the attack on the Iranian centrifuges (Farwell et Rohozinski, 2011). This confirms the urgency of in-time detection and fixing of zero-day vulnerabilities and shows that areas, even the seemingly most secure, are affected. Several approaches (Goseva-Popstojanova et Tyo, 2018a; Wijayasekara *et al.*, 2014; Wijayasekara *et al.*, 2012; Perl *et al.*, 2015) have been tried to reduce these vulnerabilities.

In the industry context, giants such as Google and Microsoft have proposed free resources based on the following principles :

— *full disclosure* : it consists of publishing the vulnerability so that all stakeholders know its existence. In this scheme, the attackers, the software developers, and the potential victims have the same information ;

— *responsible disclosure :* it aims to consult the stakeholders affected by the vulnerability (companies or the free software community) and offers them a time to correct the vulnerability before full disclosure. For example : Google announced since 2014 its zero-project[1] team that focuses on zero-day vulnerability detection. This team discloses vulnerabilities responsibly to encourage affected companies to fix their vulnerabilities as soon as possible ;

— *non-disclosure* : non-disclosure proposes that discovered vulnerabilities are not published.

1. https ://googleprojectzero.blogspot.com

In the research context, the zero-day vulnerabilities are addressed through various different ways. Zero-day vulnerabilities are consequences of certain specific flaws. To address the existence of silent vulnerabilities and the long delay in fixing these vulnerabilities, many studies have been carried out to correct these flaws and can be grouped into two main categories :

— Identification of vulnerability-fixing patches (Sun *et al.*, 2019; Zhou et Sharma, 2017; Sabetta et Bezzi, 2018; Meneely *et al.*, 2013; Scandariato *et al.*, 2014; Ji *et al.*, 2018; Yamaguchi *et al.*, 2013). The second axis proposes approaches that speeds up the processing of security bug reports fixing and provides tools to detect vulnerability-fixing patches. The **contribution #1** of this dissertation proposes works in this area to improve state-of-the-art and designs a new approach for the detection of vulnerability-fixing patches.

— Identification of vulnerability-introducing changes and vulnerable code (Wijayasekara *et al.*, 2014; Wijayasekara *et al.*, 2012; Ponta *et al.*, 2019; Shin et Williams, 2008; Chowdhury *et al.*, 2008; Li *et al.*, 2018; Perl *et al.*, 2015; Neuhaus *et al.*, 2007). The first research axis investigates the possibility of a given change (patch) being security-relevant or not. Automatic approaches proposed leverage machine learning algorithms to detect security-sensitive patches. The **contribution #2** of this thesis proposes approaches that improve the state-of-the-art and allow to vulnerability-introducing patches detection.

1.2 Thesis statement

In this thesis, we explore the use of machine learning approaches to detect zero-day vulnerabilities and reduce delays in the patching (i.e., the act of applying code changes to a program source code) process to mitigate those in open source software. Reducing these vulnerabilities in open source software consists of considering several aspects. Several artifacts are involved in a typical scenario of detecting and fixing vulnerabilities. This ranges, for example, from the bug report describing a detected vulnerability in the software, to the patch that fixes the vulnerability. Proposing a comprehensive approach for reducing zero-day vulnerabilities cannot be done effectively without considering all these artifacts and their interactions. To that end, we investigated the analysis the two main artifacts of software development process (Figure : 1.1) :

— *Detect and disclose errors in the patching process :* **patches** that are fixing a bug could contain non-explicit vulnerability-fixing patches, i.e. patches that fix vulnerability but that aren't labeled as. When a silent vulnerability-fixing patch is not identified as, with the right priority by maintainers, it could increase the duration of the zero-day vulnerability presence. To reduce it, we implement approaches to predict security-sensitive fixes to accelerate the patching process (contribution #1).

— *Detect and disclose zero-day vulnerabilities :* considering the **patches** as contributions from software teams to create functionalities. These patches, instead of just adding new features and fixing bugs, could also introduce vulnerabilities. When these vulnerabilities are not identified at the time by legitimate parts, it may lead to the introduction of zero day vulnerability. To fix it, we design a new approach to automatically detect whether an incoming patch will introduce some vulnerabilities (contribution #2).

1.2.1 Existing lines of research

Many approaches were proposed to reduce the exposition of vulnerabilities in source code. These approaches succeeded depending on their application scenario but are still limited due to the complex life cycle of vulnerabilities introduction, the missing of a comprehensive representation of vulnerabilities sensitives patches (i.e, security-sensitive patches) and the problem of unbalanced datasets in this area. We highlight three main research axes based on state-of-the-art reviews in this section.

— Static code analysis : using static analysis for software vulnerabilities detection is the first existing axis that yields those detection. It is based on software code parsing to highlight code-snippets that could lead to vulnerable actions. However, these static approaches are limited because of the rapid evolution of vulnerability patterns. Therefore, it is still challenging to keep up with the detection of new vulnerabilities.

— Dynamic execution analysis : the dynamic analysis of software allows catching some vulnerabilities that are not necessarily visible through static approaches. Dynamic taint analysis or fuzzing analyze the applications to find some execution faults that can be used as a backdoor for malicious attacks. However, this type of approach is part of the downstream detection approaches, as it allows the presence or absence of the vulnerability to be detected in the code and requires the code to be compiled each time. This is not evident in the time optimization view due to the massive amount of code set to execute at each time.

  Static code and dynamic execution analysis allow the identification of vulnerable code and vulnerable behavior of code by analyzing the code or its execution directly. However, these approaches are limited by the rapid evolution of vulnerabilities and the massive scale of code.

— Machine learning : automatic learning approaches are recognized for predicting future behavior better than the other methods for most of cases. They allow for a given problem to train models that will predict with excellent efficiency whether or not an element belongs to a given class. Moreover, these approaches will enable the evolution of the models according to the data to adapt interactively to the growth of the nature of the problem. These approaches handle rapid pattern growth well through their prediction models that can learn from existing pattern-sets and predict a new vulnerability pattern based on the trained model. They are also reputed to work better with vast sets of data. This could be the alternative to the scaling problem of fuzzing approaches.

1.2.2 Thesis map

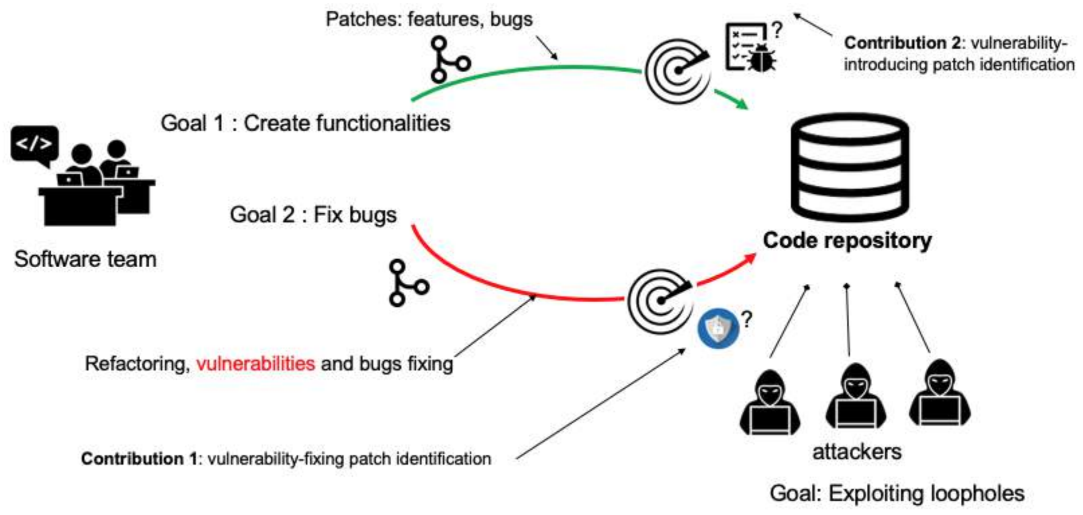

Patches: features, bugs
Contribution 2: vulnerability-introducing patch identification
Goal 1 : Create functionalities
</>
Software team
Goal 2 : Fix bugs
Code repository
Refactoring, vulnerabilities and bugs fixing
Contribution 1: vulnerability-fixing patch identification
attackers
Goal: Exploiting loopholes

Figure 1.1 : Thesis map

Figure 1.1, presents the thesis map. We adopted a method that analyze primary artifacts in the software development cycle to allow legitimate parts whose main objective is to develop features to compete with attackers. The main steps of this method can be grouped into the following points :

— *upstream vulnerability detection problem.* Unlike existing classical vulnerability detection approaches (static, dynamic, etc.), which try to detect vulnerabilities in software once they have been introduced, we are interested in predicting the vulnerable commits through machine learning techniques.

— *analysis of artifacts.* To avoid the problems of scale with the rapid evolution of open source code (For example : chromium project from 13/02 to 20/02 2020 : 2,069 commits on the master branch, 175,343 files were modified, including 175,343 additions and 157,809 deletions), we propose a set of approaches by analyzing each commit and bug report in order to be able to predict their nature (security impact or not) before they are validated in the code repository.

— *Enable legitimate parties to be alert once a security-sensitive commit is predicted.* We propose automatic approaches that will act as sentinels and could be useful for :

  — developers when a predicted commit may introduce or fix a vulnerability or when a predicted bug report may contain security-related information. This overcomes the problem of silent vulnerability patches on the one hand, and on the other hand, reduces the time taken to expose and triage security-related bug reports.

  — the users when predicting a commit that may fix a vulnerability. This avoids exposed vulnerability attacks on users who have not updated their version during a silent fix.

1.2.3 Thesis problems

Problem #1 : Lack of High-quality labeled dataset for zero-day vulnerabilities

Dataset quality plays an essential role in prediction performance for machine learning approaches. The principal limit is that the common existing approaches generally use some binary dataset. This splitting approach doesn't reflect real-world problems because the practical cases are not typically binary. For example the assumption of silent vulnerbilities means that a bug labeled as non-security related can be security-relevant without the legitimate parties knowing.

Problem #2 : Dataset Imbalance

A balanced dataset is crucial for creating a good training set (Orriols et Bernadó-Mansilla, 2005). Most existing classification methods do not perform well on minority class examples when the dataset is highly imbalanced. They aim to optimize the overall accuracy without considering the relative distribution of each class (Liu *et al.*, 2011). Typically real-world data are unbalanced, and it is one of the leading causes of the decrease in generalization in machine learning algorithms (Kim, 2007). Most of existing learning algorithms do not take into account the imbalance of class. They give the same attention to the majority class and the minority class. It is hard to build a good classifier in these conditions (Zhang *et al.*, 2010). The cost in miss predicting minority classes is higher than that of the majority class for many unbalanced datasets ; this is mainly the case in security-sensitive datasets where tagged vulnerabilities and vulnerability fixes tend to be the minority class.

Problem #3 : Absence of the relevant feature-sets that are specifically suited to the vulnerability management task.

Feature engineering plays an important role in all machine learning tasks (Rohrhofer *et al.*, 2021). With the exception of the work by Tian *et al.* (Tian *et al.*, 2012) proposing a set of features to predict bugs better, there is a lack of works on security-related feature extraction. This prevents security experts from identifying after predictions the most relevant features to adopt preventive solutions. For example, based on information gain [2], it is possible to identify the set of features

2. Information gain is a metric based on entropy that allows telling how important a given

that influenced the model prediction.

Problem #4 : Limitations in terms of model explainability

Instead of having only the high predictions performance, a vulnerability prediction task should also be comprehensive and explainable. A comprehensive approach can help developers and security-teams to avoid bad practices and flags causes of vulnerabilities presence. In this respect, many deep learning approaches are limited by explainability despite the high prediction performance they achieve. Additionally, black-box problems (and explainable AI), non-detection of exceptional cases, lack of prioritization, and confirmation bias problems limit these deep learning approaches in zero-day vulnerabilities mitigation problems.

1.2.4 Research methodology

The Figure 1.2 illustrates the research methodology. We split the approach into two significant works : i) work 1 : Automatic identification of security-sensitive fixes and ii) work 2 : Automatic identification of vulnerability-introducing patches. We describe the details in the overview section.

Common steps

The part entitled shared steps contains the steps of our approach shared by the two major works carried out (work 1 and work 2).

The purpose of this thesis is propose an automatic approach to reduce zero-day vulnerabilities in open source software.This includes also the identification of silent

attribute of the feature set is.

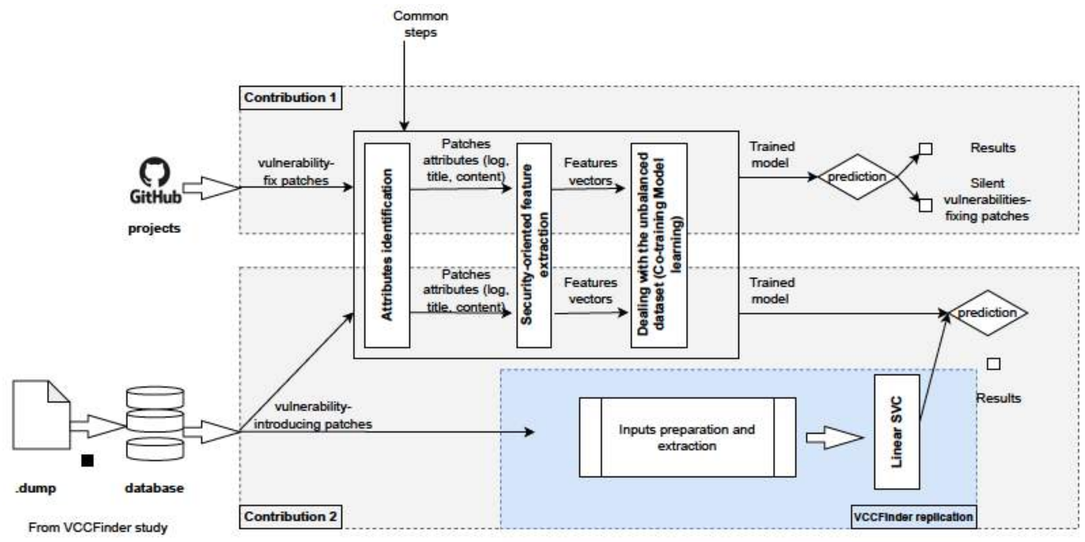

Common steps
Contribution 1
GitHub projects
vulnerability-fix patches
Attributes identification
Patches attributes (log, title, content)
Security-oriented feature extraction
Features vectors
Dealing with the unbalanced dataset (Co-training Model learning)
Trained model
prediction
Results
Silent vulnerabilities-fixing patches
Contribution 2
.dump
database
From VCCFinder study
vulnerability-introducing patches
Inputs preparation and extraction
Linear SVC
VCCFinder replication

Figure 1.2 : Research methodology

security-fixes patches 1.2. Considering a set of security-related patches, we first sought to identify attributes of each patch (e.g., patch diffs, commit messages, authors information, meta-information, etc.). Once attributes are identified with respect to our need to have a one-time predicting approach, we then proceed to security-related feature engineering. We extracted and assessed features that represent "facts" of the patch (e.g., lines_sizeof, lines_added, tf-IDF of commits messages, etc.) to generate relevant features vectors. In the final step, we propose a model learning approach that deals with unbalanced data sets for better prediction performance.

Contribution #1.

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 part, 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 (cf. Chapter 3.

Contribution #2.

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

cible baseline for advancing research on machine learning-based identification of vulnerability-introducing commits (cf. Section 4).

Summary.

We ensure to answer the thesis problems highlighted in the section 1.2.3 through this research methodology.

To address ***Problem #1 : Lack of high-quality labeled dataset for zero-day vulnerabilities***, we build and share with the community a qualitative and split dataset based on artifacts contents (patches attributes).

To address ***Problem #2 : Dataset Imbalance***, we propose a semi-supervised learning approach based on co-training that deals with the unbalanced datasets.

To address ***Problem #3 : Absence of the relevant feature-sets that are specifically suited to the vulnerability management task***, we propose an explainable and feature-engineering approach that extracts relevant and representative feature vectors.

To address ***Problem #4 : Limitations in terms of model explainability***, we propose a specific feature engineering setup which enables analysts to track down the high-level vulnerability-relevant reasons why the model predicts a patch to be security-relevant.

1.3 Thesis contributions

In this thesis, we propose an automatic learning approaches to predict the future behavior of development artifacts and thus reduce zero-day vulnerabilities. The main contributions are listed as follow :

— We motivate and dissect the problem of identifying security-relevant code changes. In particular, we investigate the discriminative power of various features to clarify the possibility of a learning process.
— We propose a semi-supervised approach with Co-Training (Blum et Mitchell, 1998) which we demonstrate to yield high precision (80%) and recall (80%). This represents a significant improvement over the state-of-the-art.
— 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.
— We perform a replication study of VCCFinder, highlighting the different steps of the methodology and assessing to what extent our results conform with the author's published findings.
— We rebuild and share a clean, fully reproducible pipeline, including artifacts, for facilitating performance assessment and comparisons against the VCCFinder's state-of-the-art approach. This new baseline might help unlock the field.
— We explore the feasibility of assembling a new state of the art in vulnerability-contributing commit identification by assessing a new feature set.
— We leveraged co-training to resolve the issue of lacking labeled data.

1.4 Roadmap

The remainder of this thesis is structured as follows. Chapter 2 sets out the thesis background, focusing on the main axis of the statement. Chapter 3 presents the first contribution in this thesis. Chapter 4 presents the second contribution. The last Chapter highlights possible future works and the conclusion of this thesis.

CHAPITRE II

BACKGROUND AND RELATED WORK

In this chapter, we discuss main concepts related to vulnerability management, in particular 1) zero-day vulnerabilities, 2) security-related bug reports, 3) vulnerability-fixing patches identification, and 4)vulnerability-introducing patches identification.

2.1 Zero-day vulnerabilities

A zero-day attack is a malicious attack that exploits a vulnerability that has not been publicly disclosed (Bilge et Dumitraş, 2012). There is virtually no defense against a zero-day attack. As long as the vulnerability remains unknown, the affected software cannot be patched, and anti-virus products cannot detect the attack through signature-based scanning. For attackers, unpatched vulnerabilities in popular software represent an open door for any target they wish to attack.

The National Vulnerability Database (NVD [1]) maintains a database with extensive information about vulnerabilities, including technical details and disclosure dates. The NVD defines a vulnerability as a software bug that allows attackers to execute commands as other users, access data that have access restrictions, behave as

1. https ://nvd.nist.gov

another user or launch denial of service attack, etc. In general, a zero-day attack is an attack that exploits vulnerabilities not yet disclosed to the public. However, the life cycle of exposure vulnerabilities is more complex. Indeed, until a vulnerability ceases to affect end-hosts after several years, there can be a race between these attacks and the remediation measures deployed by the security community. This race contains these steps :

1. A programming security-related bug that evades testing
2. Attacker sometimes discover the vulnerability before legitimate parties, exploit it, and package the exploit with a malicious payload to conduct zero-day attacks against the selected target
3. After the vulnerability or the exploits are discovered by the security community and described in a public advisory, the vendor of the affected software releases a patch for the vulnerability and security vendors update anti-virus signatures to detect the exploit or the specific attacks
4. However, the exploit is then reused, and in some cases, additional exploits are created based on the patch (Brumley *et al.*, 2008), for attacks on a larger scale, targeting Internet hosts that have not yet applied the patch.

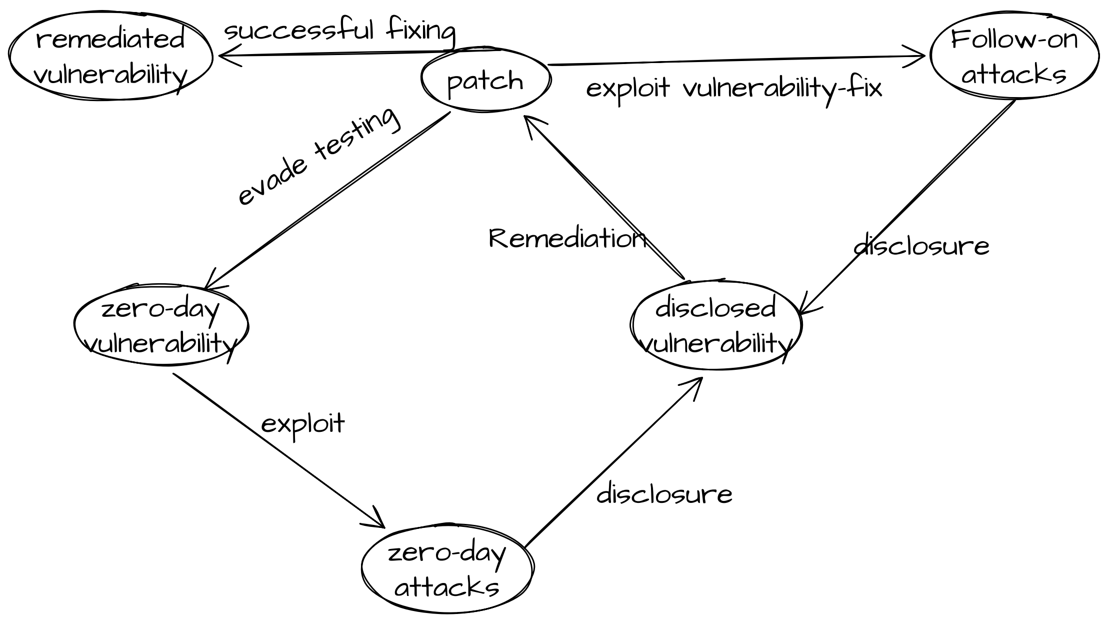


Figure 2.1 : Zero-day vulnerabilities life cycle

Improving the disclosure and remediation process is the best way to reduce zero-day vulnerabilities in software (Figure 2.1). This starts by proposing automatic approaches to identify a patch's characteristics that introduce vulnerabilities to facilitate the testing process in time. The second area is the identification of patches to reduce the time between disclosure and remediation and also to take advantage of the relevant nature of these patches to remediate vulnerabilities. This can also be achieved by learning the difference between simple bug fixes and vulnerability fixes which can be achieved by carefully studying the representation of a security-related patch. The final area that could reduce exposed vulnerabilities is to facilitate disclosure by providing approaches that identify and prioritize security-related bug reports.

2.2 Security-related bug reports

Prompt patching is essential to protect users and developers from the disastrous consequences of malicious attacks. In practice, developers fix bugs by priority. Bugs that affect the security of the entity and its users (security bug reports) are given higher priority than those that do not directly impact security. When such a bug is labeled as affecting security, developers quickly fix the bug, and users are informed of the need to upgrade to a new version of the library or a new version of the library or application. Unfortunately, very often, some security-related bugs go unnoticed. Identifying security-related bugs is then essential to limit the exposure of vulnerabilities. Many works in the literature have proposed approaches to automate the detection of commits that introduce vulnerabilities. We will focus here only on approaches that use machine learning techniques.

Supervised learning is the most widespread approach leveraged in the literature for security bug report identification. Wijayasekara *et al.* (Wijayasekara *et al.*, 2014) have presented a seminal work on detecting security bug reports using machine learning. They rely on text mining to extract syntactical information bug reports and compress them before generating feature vectors fed to Naive Bayes classifiers. Gegick *et al.* (Gegick *et al.*, 2010) used a term-by-document frequency matrix from words in the natural language descriptions of bug reports to training a statistical model. Similarly, Behl *et al.* (Behl *et al.*, 2014), later compared term frequency-inverse document frequency (TF-IDF) against a probabilistic learning approach like Naives Bayes.

Zou *et al.* (Zou *et al.*, 2018) proposed to use a combination of text-mining features and meta-data (e.g., time, severity, and priority) for improving the identification of security bugs reports. They trained a supervised approach (SVM) with Radial Basis Function(RBF) and improved previous work by over 20 percentage points.

More recently, Das *et al.* (Das et Rahman, 2019) and Pereira*et al.* (Pereira *et al.*, 2019) proposed an approach based on class imbalance sampling and TF-IDF vectors to improve security-relevant bug report detection using Naive Bayes Multinomial classification. Following up on these state-of-the-art investigations, Peters *et al.* (Peters *et al.*, 2019) proposed FARSEC, a framework for filtering and ranking bug reports to reduce the presence of security-related keywords and improve text-based prediction models for security bug fixes.

Semi-supervised and unsupervised learning approaches have been experimented with by Mostafa *et al.* (Mostafa *et al.*, 2019) and Goseva-Popstojanova *et al.* (Goseva-Popstojanova et Tyo, 2018b). The first work presented an evolutive and realistic approach for the identification of security bug reports which considers the evolution of security vocabulary on NVD database and practical constraints like small training set for security bugs reports prediction, and the second assesses the impact of algorithms and features in the detection of security bug reports.

2.3 Vulnerability-fixing patch identification

Identifying fixing changes that are labeled as being security-relevant, i.e., as fixing a vulnerability is related to several research directions in the literature, most notably studies on 1) security commit identification, 2) vulnerability management and 3) change analysis.

2.3.1 Security commit identification

Recently, researchers from the security industry (Zhou et Sharma, 2017; Sabetta et 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 et 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 et 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).

2.3.2 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.*, 2014a) 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 et 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. (Jimenez *et al.*, 2018) 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.*, 2014b). 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).

2.3.3 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). Another related 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.

2.4 Vulnerability-introducing patch identification

The possibility of automatically finding vulnerabilities in code bases has long been identified by researchers as a worthy investigation target. In this section, we present a selection of significant prior works that we group by families of approaches, most notably studies on 1) static analysis for vulnerability detection, 2) vulnerability detection with symbolic execution, 3) vulnerability detection with dynamic analysis, 4) vulnerability detection with code metadata, 5) machine learning application for vulnerability analysis and 6) vulnerability detection at commit level

2.4.1 Static analysis for vulnerability detection

First released in May 2001, Flawfinder performs static analysis of C and C++ programs and detects calls to a manually curated list of sensitive APIs (Ferschke *et al.*, 2012). Examples of such APIs widely recognised as sensitive are `strcpy`, `random` or `syslog`.

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

Find-Sec-Bugs[2] targets Web applications written in Java, and searches for potential vulnerabilities by matching high-level patterns that model problematic code pieces. Find-Sec-Bugs was made available to developers through a convenient IDE plugin.

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

— **Frama-C** (Signoles *et al.*, 2012), that leverages Static- and Dynamic-Analysis, Formal verification, and Testing ;

— **Clang**[3], that can find bugs such as memory leaks, 'use after free' errors, and dangerous (though valid) type casting ;

— **Oclint**[4], that performs analyses of Abstract Syntax Trees to find known patterns of dangerous code constructs ;

— **Cppcheck**[5], that specialises in finding undefined behaviours, and that strives to produce very few False Positives ;

— **Infer**[6], that catches memory safety errors by trying to build formal proofs of programs, and then interpreting failures of proof as bugs ;

— **Uno** (Holzmann, 2002), that offers an approach aiming at detecting a limited number of errors, but with high precision ;

— **Sparse**, that was developed by (Torvalds *et al.*, 2003) specifically for the Linux kernel and thus can detect low-level errors in (among other things) bitfields operations or endianness ;

2. `https://find-sec-bugs.github.io`
3. `https://clang-analyzer.llvm.org`
4. `http://oclint.org`
5. `http://cppcheck.sourceforge.net`
6. `https://fbinfer.com`

— **Flint++** [7], that can detect and warn developers about dangerous coding practices.
— **git-vuln-finder** [8], that is based on C/C++ pattern matching.

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

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

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

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

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

7. `https://github.com/JossWhittle/FlintPlusPlus`
8. `https://github.com/cve-search/git-vuln-finder`

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

2.4.2 Vulnerability detection with symbolic execution

Symbolic execution has also long been identified by researchers as a promising technique to detect vulnerabilities in software. It enables some flexibility on the testing by using unknown symbolic variables rather than hard-coded-like asserting tests. Symbolic execution methods were notably experimented in cadar2008klee by the tool KLEE that found 56 new bugs, including 3 in COREUTILS (Cadar *et al.*, 2008).

A good review of the use of Symbolic execution for software security was published in cadar2013symbolic by (Cadar et Sen, 2013).

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

2.4.3 Vulnerability detection with dynamic analysis

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

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

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

2.4.4 Vulnerability detection with code metadata

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

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

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

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

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

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

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

2.4.5 Machine learning application for vulnerability analysis

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

Literature approaches have employed learning techniques on diverse programming languages and software systems : (Chang *et al.*, 2008) have applied a HMFSM (Heuristic Maximal Frequent Subgraph Mining) to four C programs (make, openssl, procmail and amaya). Their approach uses a a mix of static analysis and data mining to extract patterns that were then associated with their frequency : the more frequent a pattern, the safer it is considered. In their evaluation, they managed to find 3800 violations of well-known patterns. (Zimmermann *et al.*, 2010) proposed to use a measure of code complexity (McCabe, 1976) to predict the presence of vulnerabilities in Windows Vista. Using Linear Regression, they manage to have a precision below 64% for a relatively low recall of 21% on a ten-fold validation process. (Yamaguchi *et al.*, 2013) have presented CHUCKY, an approach to identify anomalous or missing checks on C programs. It is a combination of taint analysis and machine learning that results in finding up to 96% of missing checks by comparing a piece of code to the most similar ones. (Scandariato *et al.*, 2014) extracted text from 182 releases of 20 Android applications to generate feature vectors, using a feature discretisation method proposed by (Kononenko, 1995). This approach achieved good performance for detecting vulnerabilities within a project, but lower performance for inter-project detection. DEKANT was proposed to generate a model out of sliced pieces of PHP applications and WordPress plugins (Medeiros *et al.*, 2016). This model, based on a set of annotated source code, serves as the basis for the discovery of new vulnerabilities.

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

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

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

2.4.6 Vulnerability detection at commit level

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

(Yang *et al.*, 2017) focus on automatically detecting vulnerability-contributing changes in the Mozilla Firefox project. The tool extracts features from commits and uses a random forest classifier to detect VCCs. By first using an estimated number of potential VCCs present in the code under analysis, they claim to produce fewer False Positives than VCCFinder. (Sabetta et Bezzi, 2018) consider the code modified by a commit as a text document, and then leverage Natural Language Processing techniques to feed multiple machine learning classifiers. One of (Wang, 2019) contribution is to filter commits by excluding or including those matching a list of keywords. For example, their filtering step can discard up to 92% of commits, hence vastly reducing the effort needed to analyse the suspicious commits.

Other works have directly mentioned and inherited from VCCFinder. Directly trying to improve on VCCFinder, in a 5 pages technical report, (Yamamoto, 2018) aims at decreasing the number of false-positive results yielded by VCCFinder. To that end, he proposes to separate additions from deletions in the commits to extract code-related features. The results presented in this technical paper are claimed to be slightly better than those of VCCFinder. (Zhou et Sharma, 2017) compare different algorithms for automatically discovering security issues. Albeit mentioning that VCCFinder uses LinearSVM, they only consider information from the commit message, gathered using regular expressions, and from bug reports.

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

CHAPITRE III

VULNERABILITY-FIXING PATCH IDENTIFICATION

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 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 case 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 et 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 et 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 et 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 chapter, 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 et 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 chapter is organized as follows. We motivate our study in Section 3.1 and overview data collection in Section 3.2. Section 3.3 describes SSPCatcher while Section 3.4 presents the experimental study and results. Section 3.5 discusses threats to validity and future work and Section 4.4 summarise our contributions in this work.

3.1 Motivation

The urgency of updating a software given a proposed change is assessed at different levels of the software development cycle. The stakeholders here are (1) Developers that are using third libraries parts, (2) maintainers that validate developers' code, and (3) the user that use the software and make some updates. We then 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.

Figure 3.1 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 3.2 ; 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 et Whitney, 1947).

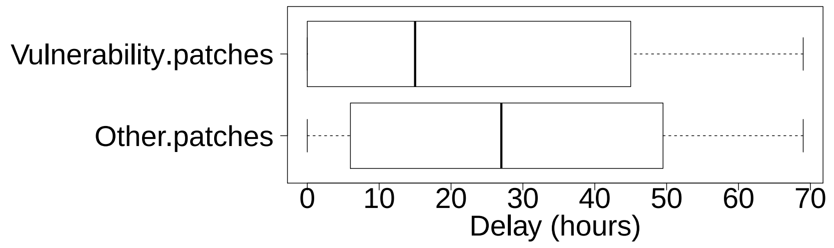


Figure 3.1 : Delays for validating contributor patches in Linux based on explicit vulnerabilities

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 work : 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 Figure 3.2 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.

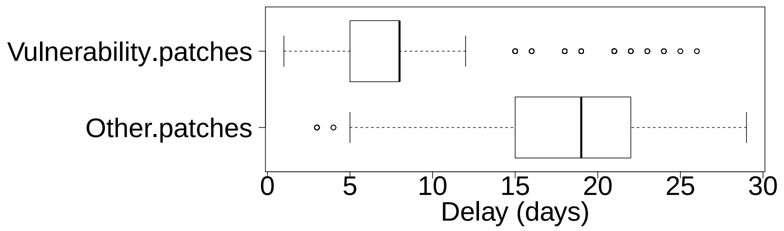


Figure 3.2 : Comparative delays for OpenSSL release after an explicit security patch vs after any other patch

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.

3.2 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 Figure 3.3, 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;
```

Figure 3.3 : 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 (**?**) 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 recognized 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 Figure 3.4 and detailed in the following paragraphs.

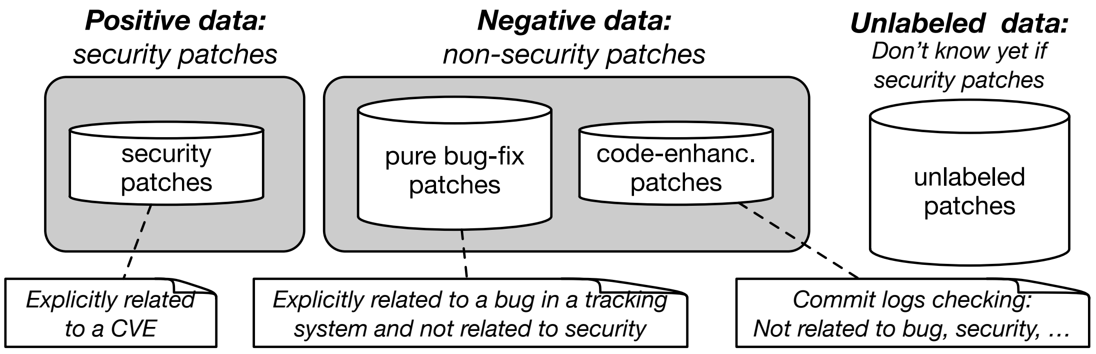

Positive data:
security patches
Negative data:
non-security patches
Unlabeled data:
Don't know yet if security patches
security patches
pure bug-fix patches
code-enhanc. patches
unlabeled patches
Explicitly related to a CVE
Explicitly related to a bug in a tracking system and not related to security
Commit logs checking: Not related to bug, security, ...

Figure 3.4 : Distinct subsets of the dataset built for our experiments

3.2.1 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, 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 (**?**) 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 pro-

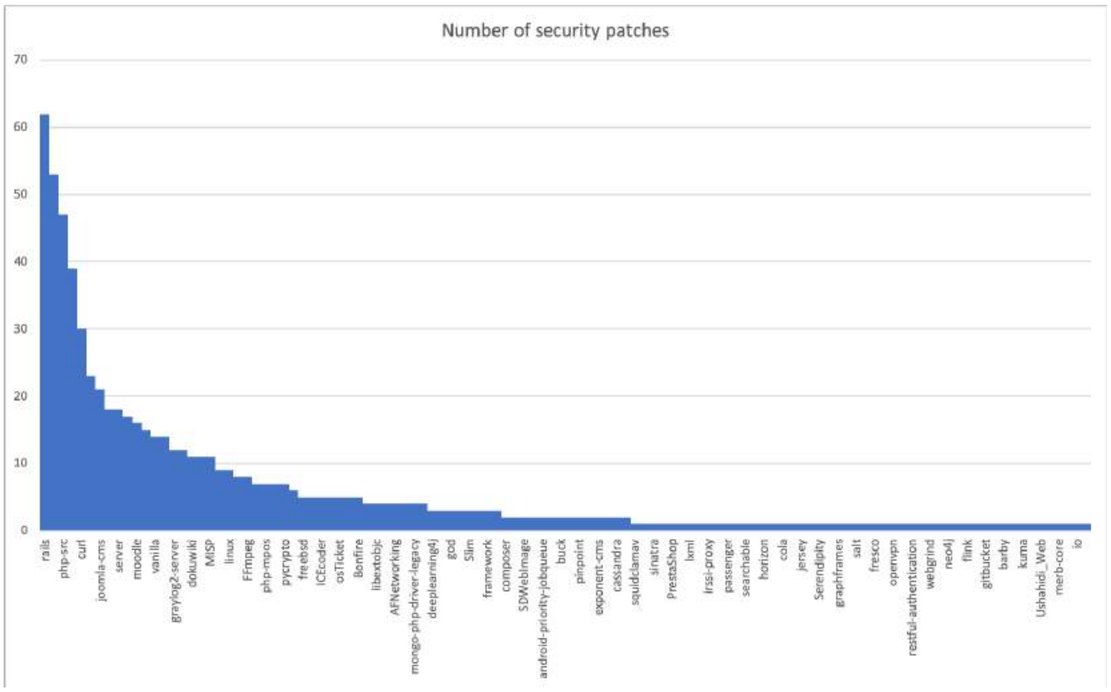


Figure 3.5 : Secbench dataset distribution

jects 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. Figure 3.5).

3.2.2 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 Sharma (Zhou et 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 3.1. Overall, with this method, we managed to retrieve 1 934, 2 477 and 8 142 pure 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).

Table 3.1 : 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)
```

3.2.3 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.

3.2.4 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 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 3.2 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

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.

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 3.1, 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.

Table 3.2 : 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

3.3 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 correspon-

2. No pure bug fixing dataset because of links missing between bugs and commits.

ding 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 et 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 et Ghani, 2000).

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 3.3.3).

In this section, we first provide information on feature engineering (cf. Section 3.3.1) and assessment (cf. Section 3.3.2). Then, we present the Co-Training approach (cf. Section 3.3.3).

3.3.1 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.

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.

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 et 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 3.3 provides an enumeration of the exhaustive list of features used in this study.

Table 3.3 : 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		

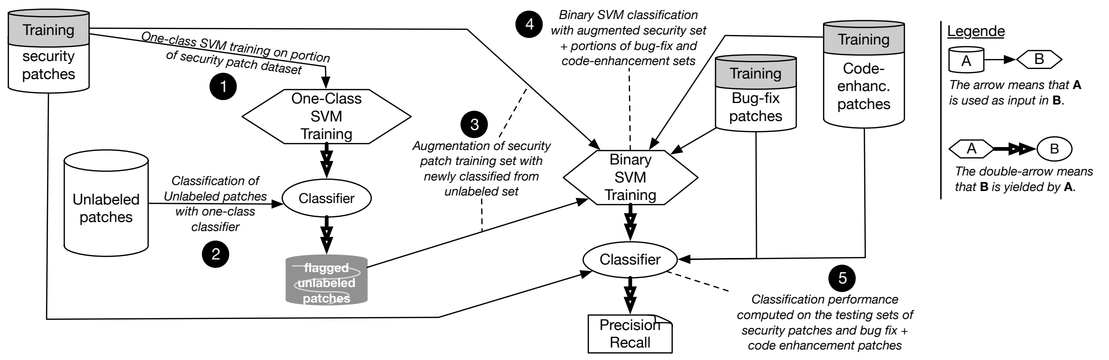

Training
security patches
One-class SVM training on portion of security patch dataset
1
One-Class SVM Training
Unlabeled patches
Classification of Unlabeled patches with one-class classifier
2
Classifier
flagged unlabeled patches
3
Augmentation of security patch training set with newly classified from unlabeled set
4
Binary SVM classification with augmented security set + portions of bug-fix and code-enhancement sets
Binary SVM Training
Training
Bug-fix patches
Training
Code-enhanc. patches
Classifier
Precision Recall
5
Classification performance computed on the testing sets of security patches and bug fix + code enhancement patches
Legende
A
B
The arrow means that **A** is used as input in **B**.
A
B
The double-arrow means that **B** is yielded by **A**.

Figure 3.6 : Workflow for assessing the discriminative power of features

3.3.2 Feature Assessment

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 et 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, 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 :

$$F_{norm} = \frac{F}{\#patch\_added\_lines + \#patch\_removed\_lines} \tag{3.1}$$

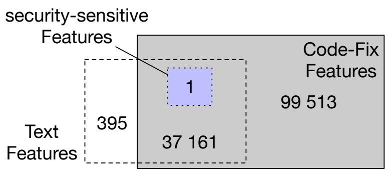
where the normalized value F_{norm} of a feature is computed by taking into account the patch size. Table 3.4 provides some example cases where the statistical tests were successful against a strict significance level of $\alpha = 0.0005$ for the p-value. Due to space limitations, we show only top-3 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.

Table 3.4 : Statistical analysis results for top normalized features with highest discriminative potential.

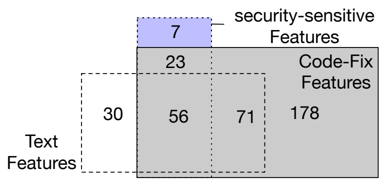
	Code-fix features			sec.-sensitive features			Text features		
	F6	F16	F24	F11	F22	F24	W2	W4	W6
Mean for security patches	0.120	0.038	0.110	0.004	0.006	0.350	0.360	0.360	0.350
Mean for other patches	0.090	0.016	0.050	0.003	0.004	0.330	0.310	0.320	0.330
P-value (MWW)	$5e^{-62}$	$2e^{-40}$	$4e^{-103}$	$1e^{-13}$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

Classification experiments

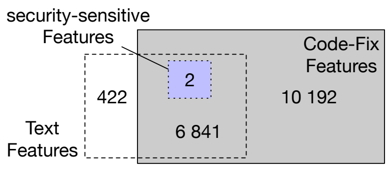
The previous statistical analysis assessed the discriminative power of engineered features with respect to security patches and the combined set of bug fixing and code enhancement patches. We propose to further assess the behaviour of one-class classification models with these features applied to the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



(a) Flagged Linux unlabeled patches



(b) Flagged OpenSSL unlabeled patches



(c) Flagged Wireshark unlabeled patches

Figure 3.7 : 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.

— *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 et 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 et Frank, 2008). In many cases, a one-class classifier is used in preference to a multi-class classifier, 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 Figure 3.6.

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 (3.2) and (3.3) 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.

$$Precision = \frac{TP}{TP + FP} \ ; \ Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.2}$$

$$Recall = \frac{TP}{TP + FN} \ ; \ F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3.3}$$

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%).

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 et Sharma, 2017; Sabetta et 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 3.7 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[3], which was to be expected given that security-

3. This does not mean that security-sensitive features are useless or redundant. Patches

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 et Scheffer, 2004). This insight on the sets of engineered features serves as the foundation for our model learning detailed in the following paragraphs.

Code features (*formed by security-sensitive features + code-fix features*) and **Text features** are independent. They will represent two distinct views of the data, an essential requirement for Co-Training.

3.3.3 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

flagged with code-fix features are scarcely flagged with security-sensitive features.

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[4] (InfoGain) scoring values of human-engineered features can be used as hints for manual investigation.

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 et 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[5] 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 et Mitchell, 1998), for the problem of

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

5. Our requests to obtain datasets from authors of (Zhou et Sharma, 2017) and (Sabetta et Bezzi, 2018) remained unresponded.

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 3.8 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 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 3.3.2 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.

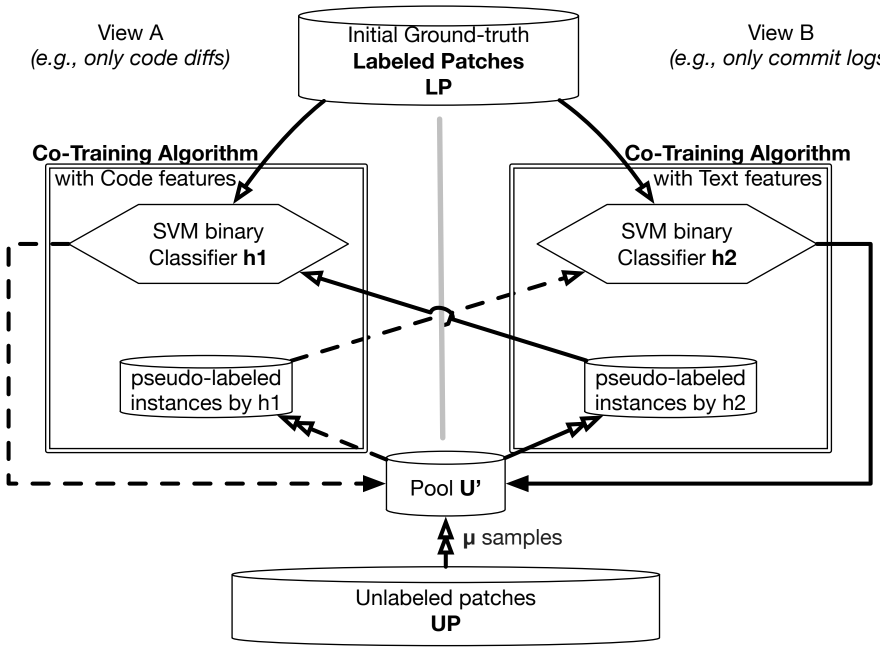


Figure 3.8 : Co-Training learning model (cf. details in Algorithm 1)

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.

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.

3.4 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

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 $classifier$;
$h_1 \leftarrow buildClassifier(True)$ ;      $h_2 \leftarrow buildClassifier(False)$;
$(P_1, N_1) \leftarrow classify(h_1, U')$ ;      $(P_2, N_2) \leftarrow classify(h_2, U')$;
$LP \leftarrow LP \cup random\_subset(\#p, P_1) \cup random\_subset(\#p, P_2)$;
$LP \leftarrow LP \cup random\_subset(\#n, N_1) \cup random\_subset(\#n, N_2)$;
$U' \leftarrow U' \cup random\_subset(\#2 * (p + n), UP)$;
```

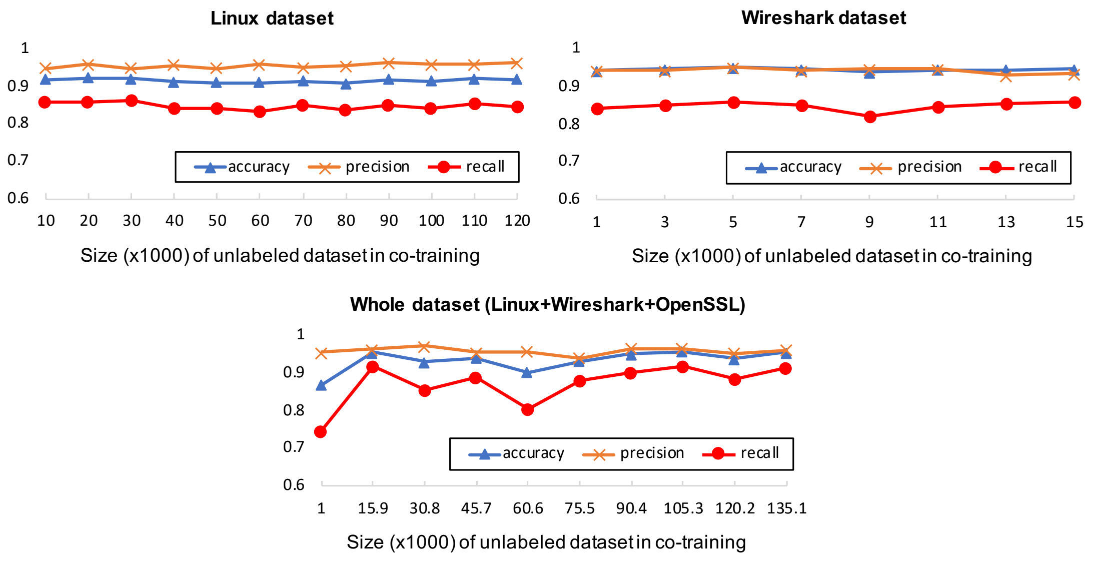


Figure 3.9 : Precision, Recall and Accuracy metrics in benchmark evaluation with varying sizes for the unlabeled dataset.

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 et 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.

3.4.1 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 Figure 3.4, the non-security patches consist in the pure bug-fix patches and code-enhancement patches. These experiments, similarly to past studies (Sabetta et Bezzi, 2018; Zhou et 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 3.9 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 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 et 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. ◄

3.4.2 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).

Cross-project classification on C-projects dataset

Table 3.5 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 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.

Table 3.5 : Cross-project classification on projects using programming language C

		Training on		
		OpenSSL precision/recall	Wireshark precision/recall	Linux precision/recall
Testing on	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)

Cross-project classification on projects using mixed programming languages

Table 3.6 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 3.7 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.

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

Table 3.6 : Cross-project classification on projects using mixed programming language

		Training on		
		OpenSSL precision/recall	Wireshark precision/recall	Linux precision/recall
Testing on	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 3.7 : 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

3.4.3 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 et 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 et Bezzi, 2018) (i.e., SVM bi-classification with bag-of-words features of code and log), and can therefore compare [6] their approach and ours in Table 3.8.

Table 3.8 : 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 et Bezzi, 2018)	0.45	0.45	0.67	0.44	0.57

6. 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 et Bezzi, 2018).

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 et 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 validation for all classifiers and evaluate the performance of the classifier in identifying labeled security patches in the whole dataset. Results in Table 3.9 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.

Table 3.9 : Importance* of Classification method and feature set

	Precision	Recall	F1-measure
SVM binary classification (*with features of Sabetta & Bezzi (Sabetta et 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.

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 3.10 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.

Table 3.10 : 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

3.4.4 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 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 Figure 3.4). 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 work) 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 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 3.10 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.

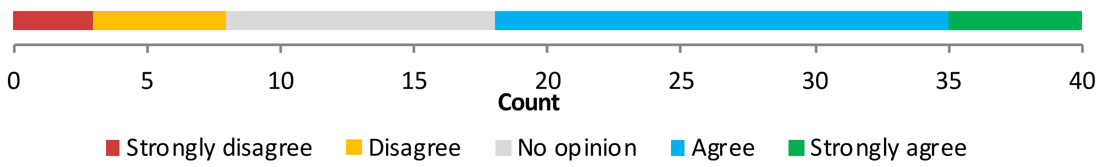


Figure 3.10 : Do the highlighted features provide relevant hints for manual review of flagged patches ?

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

3.5 Insights, Threats to Validity, and Limitations

3.5.1 SSPCatcher and the related work

Many works are related to this study (Chapter 4), as we have already explored the related works, in this part, we highlight the main differences between our work and the most closest others.

In the software change research axis, 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 of 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 et 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).

3.5.2 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.

3.5.3 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.

3.5.4 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.

3.5.5 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.

3.5.6 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.

3.6 Summary

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.

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.

CHAPITRE IV

VULNERABILITY-INTRODUCING PATCH IDENTIFICATION

Software development is a complex engineering activity. At any stage of the software lifecycle, developers will introduce bugs, some of which will lead to failures that violate security policies. Such bugs are commonly known as *software vulnerabilities* (Krsul, 1998) and are one of the main concerns that our ever-increasingly digitalised world is facing. Detecting software vulnerabilities as early as possible has thus become a key endeavour for software engineering and security research communities (Zhu *et al.*, 2019; Cadar *et al.*, 2008; Livshits et Lam, 2005; Larochelle et Evans, 2001). Typically, software vulnerabilities are tracked during code reviews, often with the help of analysis tools that narrow down the focus scope by flagging potentially dangerous code. On the one hand, when such tools build on static analysis (either deciding based on code metrics or matching detection rules), the number of false positives can be a deterrent to their adoption. On the other hand, when the tools build on dynamic analysis (e.g., for pinpointing invalid memory address), they are operated on the entire software which may not scale to the frequent evolutions of software.

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

VCCFinder is a machine learning approach that trains a classification model, which can discriminate between safe commits and commits that lead to the code being vulnerable. The experimental assessment presented by the authors has shown great promise for wide adoption. Indeed, by training a classifier on vulnerable commits made in 2011 on open source projects, VCCFinder was demonstrated to be capable of precisely flagging a majority of vulnerable commits that were made between 2011 until 2014. VCCFinder further produced 99% less false positives than the tool the authors decided to compare their implementation to, namely FlawFinder (Ferschke *et al.*, 2012). Finally, the authors reported that VCCFinder flagged some 36 commits to which no CVE was attached, one of which has been indeed confirmed as a vulnerability introducing commit.

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

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

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

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

- We perform a replication study of VCCFinder, highlighting the different steps of the methodology and assessing to what extent our results conform with the authors published findings.
- We rebuild and share a clean, fully reproducible pipeline, including artefacts, for facilitating performance assessment and comparisons against the VCCFinder state-of-the-art approach. This new baseline might help unlock the field.
- We explore the feasibility of assembling a new state of the art in vulnerability-contributing commit identification, by assessing a new feature set.

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

— We identify one issue to be the lack of labelled data, and we explore the possibility to leverage a specialised technique, namely co-training, to mitigate that issue.

The main findings of this work are as follows :

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

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

The rest of this chapter is organised as follows :

— We first focus on describing the VCCFinder approach : what resources are available, what we had to guess, and how we reimplemented it (Section 4.1). We compare the achieved results with the originally presented ones.
— We then propose and evaluate in Section 4.2 a new approach, built with

another feature set, and co-training.

— We finally summarise our contributions in Section 4.4.

4.1 Replication study of VCCFinder

The first objective of our work is to investigate to what extent the VCCFinder (Perl *et al.*, 2015) state-of-the-art approach can be replicated (different team, different experimental setup) and/or reproduced (different team, same experimental setup). VCCFinder [2] is a machine learning-based approach aiming at detecting commits which contribute to the introduction of vulnerabilities into a C/C++ code base.

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

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

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

2. VCCFinder means Vulnerability-Contributing Commit Finder

4.1.1 Datasets

Datasets - VCCFinder Paper

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

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

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

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

To overcome this difficulty, the authors follow an approach defined by (Śliwerski *et al.*, 2005) and called SZZ. The principle is to start from vulnerable lines of code. Such vulnerable lines of code are identified thanks to the vulnerability fixing

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

commits : indeed, it is reasonable to assume that the lines that have been fixed were previously vulnerable. Then the `git blame` command is used on these identified lines of code. The `git blame` command allows finding the last commit that modified a given line. The assumption here is that the last modification made on a vulnerable line of code is the modification that introduced the vulnerability.

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

In the VCCFinder paper, both datasets have been divided into a training set and a testing set (following a two-third, one-third ratio). All commits created before January, 1st 2011 are put in the training set, and the remaining in the test set. The numbers of commits of each dataset are presented in the left part of Table 4.1. Note that among the whole dataset of 170 860 commits, only 1258 (640 + 718) commits have been classified. The 468 (219 + 249) labelled commits in the test set is used as ground truth, notably to compute Precision and Recall performance metrics.

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

Datasets - availability

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

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

Datasets - our Replication Study

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

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

— A table listing 179 public repositories of C/C++ projects ;
— A table listing 351 400 commits, each commit being linked to a repository thanks to the use of a repository id ;

4. `https://github.com/hperl/vccfinder`

— A table listing the CVEs used to identify the vulnerability fixing commits.

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

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

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

Table 4.1 presents a comparison between a) the number of commits that have been involved in our replication attempt, and b) the dataset described in VCCFinder original paper.

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

Table 4.1 : Datasets comparisons

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

* Vulnerability-Contributing Commit

As shown in Table 4.1, the datasets used in the VCCFinder paper and the ones used in our replication study are not identical. Even if the number of positive and negative samples in the training and test sets are close (same order of magnitude), we can notice significant differences regarding : (1) the number of repositories presenting a fixing commit (66 vs 38), (2) the number of negative samples (i.e. fix commits) in the Test sets (249 in the VCCFinder paper, 879 in our replication study).
This fact alone guarantees that we will not be able to obtain exactly identical results. Given *how much* the datasets are different, we even *expect* our results to be potentially significantly different.

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

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

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

We note that we cannot definitely affirm which of the first or the second setup VCCFinder original paper used, as both are coherent with the figures reported. The repartition is presented in Table 4.2, and detailed in the following paragraphs :

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

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

Table 4.2 : Dataset repartition scenarios

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

treated similarly in the training than in the testing.

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

4.1.2 Features

Features - VCCFinder paper

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

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

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

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

— The number of structural keywords of C/C++ programs (such as `if`, `int`, `struct`, `return`, `void`, `unsigned`, `goto`, or `sizeof`, etc) present in m. Overall, 62 keywords are referenced ;

— The number of hunks[6] in m ;

— The number of additions in m ;

— The number of files changed in R.

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

— The total number of commits in R ;

— The percentage of commits in R performed by the author of m ;

— The number of changes performed on the files modified by m after m was applied ;

— The number of changes performed on the files modified by m before m was applied ;

— The number of authors altering the files impacted by m ;

— The number of stargazers, forks, subscribers, open issues and others, including the commit message itself.

Features - availability

The earlier mentioned git repository ends up registering commits in a database, though as already stated (Section 4.1.1), we are unsure whether the resulting database would have all the information needed, in particular, we have been unable

6. A hunk is a block of continuous added lines

to locate code that would compute all the features required. Furthermore, the original paper does not contain enough details to fully re-implement the full feature extraction ourselves.

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

Features - our replication study

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

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

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

4.1.3 Machine learning algorithm

Machine learning algorithm - VCCFinder paper

The VCCFinder approach leverages an SVM algorithm (through its LibLinear (Fan *et al.*, 2008) implementation) to learn discriminating vulnerability introducing commits from other commits.

This algorithm builds a hyper-plan that would separate, in our case, vulnerability introducing commits from others. To classify a given commit, a distance is computed between the feature vector of this commit (i.e., a point in the hyper-space) and this hyper-plan.

The sign of this distance determines whether this commit contributes to a vulnerability or not.

Given a commit and the extracted features, we describe now the generation of the feature vector of this commit that is used as input of the machine learning algorithm.

This process follows a generalised bag-of-words approach that normalises the features' values into boolean vectors. Regarding the normalisation, for each feature, commits are categorised into bins based on the occurrences of the feature. Then a string is built by concatenating the name of the feature and the bin identifier.

Finally, joining all these newly created strings together with the texts formed by the patch code and/or commit message, a considerable string is built and fed to a tool named SALLY (Rieck *et al.*, 2012). SALLY is a binary tokenisation tool which generates a high-dimensional sparse vector of booleans from a string, computing a hash for each split-on-space sub-string. At the end of this process, each commit is represented now by, first, a Boolean, indicating its class (vulnerability-contributing commit or not) and a succession of pairs `(feature_hash/binary value)` that represent a sparse vector of the features.

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

Eventually, the authors present their results on the test set with a Recall-to-Precision curve for which the actual parameter is the threshold in Figure 4.1.

After computing the distance from the hyperplane for each commit in the test set and by incrementally lowering the threshold, the commits the closest to the hyperplane will be classified as VCCs. Lowering the threshold results in increasing the number of True Positives, but might also quickly bring more False Positives.

The higher the Recall-to-Precision curve, the more precise, and the more horizontal, the more the model is not sacrificing precision for recall.

Machine learning algorithm - VCCFinder availability

As already explained, VCCFinder authors did not release code that perform all the required steps of their approach. Even in the repository found on the Internet (but not mentioned in the VCCFinder paper), the code that orchestrates the training of the classifier and its usage is absent.

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

Machine learning algorithm - our replication study

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

Regarding the construction of the feature vectors, and more specifically regar-

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

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

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

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

4.1.4 Results

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

VCCFinder Paper

To assess the performance of their machine learning-based approach, the authors keep about two-thirds of their datasets for training, and use one-third of the datasets for testing. Table 4.1 presents the exact numbers. Note that, as explained in SubSection 4.1.1, we are not sure about what the training and testing sets are composed of.

The original results are presented in Figure 4.1, which is directly extracted from the paper (Perl *et al.*, 2015). The plot is obtained by measuring/computing pre-

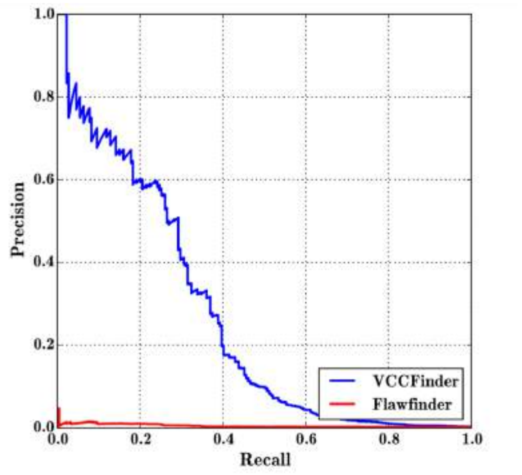


Figure 4.1 : Extracted from the VCCFinder paper : precision/recall performance profile of VCCFinders

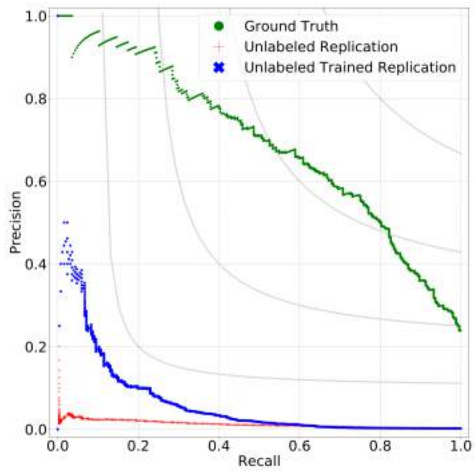


Figure 4.2 : Precision/recall performance profile of VCCFinder's Replication

cision and recall values when varying the threshold.

In the original paper, the authors compare VCCFinder against a then-state-of-the-art tool named flawfinder(in red in Figure 4.1). Flawfinder is a static analyser tool that looks for dangerous calls to sensitive C/C++ APIs in the code as `strcpy` and flags them.

Figure 4.1 shows that VCCFinder greatly outperforms Flawfinder. The authors also set their tool to the same level of recall that Flawfinder is capable of for this dataset, 24%, and show that their approach presents then a precision of 60%. In comparison, Flawfinder can only achieve 1% in such conditions. For a recall of 84%, VCCFinder has a precision of 1%.

With precision and recall values extracted from Figure 4.1, an F1-score can be computed thanks to the following formula :

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

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

Table 4.3 describes several metrics (extracted from the original paper) such as True Positive, False Positive, etc computed on the test set. VCCFinder flagged 53 commits that are, according to the ground truth, actually introducing a known vulnerability. Applying VCCFinder to the larger set of unclassified commits, 36 commits were flagged as suspicious. Among those 36 potential VCCs, one was described by authors as confirmed by the project maintainers, who had already patched this vulnerability. Authors opted not to comment on the other 35 commits, invoking "responsible disclosure".

These 36 commits are presented as belonging all to the post-January 2011 unclassified set. Thus, on what they define themselves as the ground truth, no false positive is met.

Our Replication Study

The results presented in Figure 4.2 show the precision per recall we obtain on the 3 different test sets while diminishing the threshold. One can understand the threshold as the minimum distance from the hyperplane for a commit to be considered as VCC. The grey curves represent the lines for a constant F1-score at 0.2, 0.4, 0.6 and 0.8. We now details the results for each of the 3 test sets presented in 4.1.1 :

Ground Truth Replication :

The replication achieves a maximum F1-score of 0.63 for a recall of 0.76 and a precision of 0.54 (see line 2 of Table 4.3 and green dots in Figure 4.2). We

Table 4.3 : Results of replication on updated test set

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

* VCC : Vulnerability-Contributing Commit

† Vulnerability-Fixing Commit and post-2011 Unlabelled

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

Unlabelled Replication :

This attempt trains on the ground truth but is tested on both ground truth and beyond 2011 unclassified is drawn in red in Figure 4.2. We can see it perform very poorly, presenting more than three thousand false positives, once set to the same recall of 0.24. The precision is then barely of 2% and the F1 score of 0.037.

Unlabelled Train Replication :

It is after assessing how poorly the last experiments performed that we decided to include unclassified in the training, forcing them as non-VCCs. The results are illustrated thanks to the blue curve in Figure 4.2 and the last row of Table 4.3. It improves sensibly the performances without reaching the level of the original. The precision for fixed recall is of 8%, leading to an F1-score of 0.12.

Parameters Exploration

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

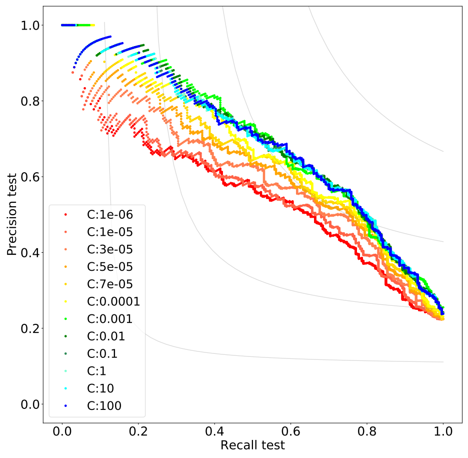


Figure 4.3 : Precision/recall performance profile of VCCFinder's replication for varying values of C parameter

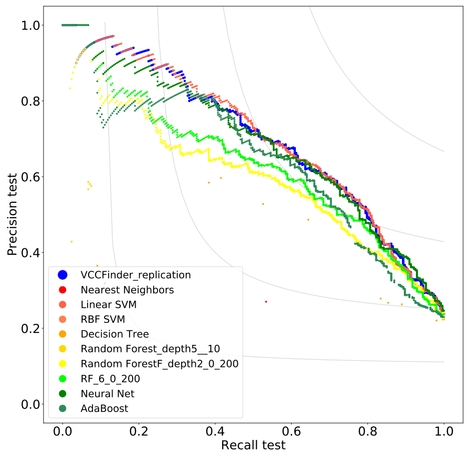


Figure 4.4 : Precision/recall performance profile for comparing classifying algorithms

Exploration over parameter C :

In the original paper it is just stated that the optimal conditions are for a cost parameter C of 1. We experiment for different values of C on the basis of the Ground Truth Replication. We experiment for values from $C = 10^{-6}$ to 100, and obtain the values presented in Figure 4.3.

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

Exploration over class weight parameter :

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

Exploration with other algorithms :

We also experimented with a variety of different machine learning algorithms.

Results are presented in Figure 4.4. We note that SVM—that is used by the original VCCFinder paper—is among the algorithms that produce the best results.

4.1.5 Analysis

We discuss the experimental results of our replication attempt of the VCCFinder approach.

RQ 1 : Is our reproduction of VCCFinder successful ?

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

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

We have been unable to *Reproduce* VCCFinder.

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

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

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

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

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

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

4.2 Research for improvement

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

Considering our released artefacts of **a new replicable baseline**, we propose to

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

investigate some seemingly-appealing variations of the VCCFinder approach to offer insights to the community. Thus, in this section, we go beyond a traditional replication paper by :

(1) Studying the impact of leveraging a different feature set that was claimed to be relevant to vulnerabilities (Sawadogo *et al.*, 2020), thus proposing a new approach to compare against VCCFinder (in Section 4.2.1) ;
(2) Trying to overcome the problem of unbalanced datasets, i.e., the fact that there are much more unlabelled samples than labelled ones (in Section 4.2.2).

4.2.1 Using an alternate feature set

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

In this section, we investigate the use of an alternate feature set, described in a recent publication sawadogo2020learning that is targeting **the detection of vulnerability fix commits, rather than the detection of VCC**. To reduce ambiguity when needed, we refer to this alternate feature set as *New Features*, while the VCCFinder feature set is denoted *VCC Features*.

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

Table 4.4 : Alternate set of features (adapted from (Sawadogo *et al.*, 2020))

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

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

New Feature Set

The *New Feature set* is made of three types of features : Text-based features, Security-Sensitive features and Code-Fix features. They are all shown in Table 4.4

— Code metrics : A difference between the two feature sets concerning the code is that the new feature set focuses on 17 characteristics of the code, while VCCFinder collects 62 keywords. Though, for each, it also computes whether they are added, removed, the difference of those two factors and their addition.
  Taken individually, most of them are common to the two feature sets. Except for the count of elements under parenthesis, function calls, keywords : `INTMAX`, `define` and `offset`, VCCFinder's feature set includes them all and beyond.

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

Table 4.5 : Confusion Table for New Features

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

Note that we tried to normalise the features (as recommended in hsu2003practical). The results of detection along the test set were the same or slightly worse with this normalisation step. Thus we decided not to normalise the features.

Results

Figure 4.5 and Table 4.5 present the performances with the New Feature Set.

By considering the Ground Truth only (second line of Table 4.5 and green curve in Figure 4.5), the New Features are less performant than VCC Features. For, still, a recall of 0.24, the precision is only 67% while it used to top at 92% in such a case.

Here again, because of the doubt on what is the actual test set in the original paper (cf. Section 4.1.1), we also tested on both the ground truth and the unclassified commits post January, 1st 2011 (red curve in Figure 4.5 and last row in Table 4.5).

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

4.2.2 Adding Co-Training

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

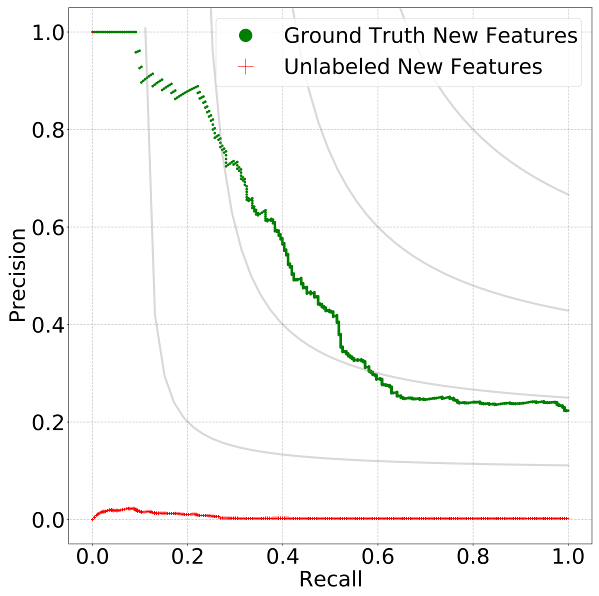


Figure 4.5 : Precision-recall performances using New Features

One field of machine learning focuses on the usability of the unlabelled data. The study by (Castelli et Cover, 1995) states that it is possible, in some case, to leverage unlabelled samples to improve a machine learning model. (Zhang et Oles, 2000) investigated the potential for gaining information from unlabelled data. This last study concludes that so called active-methods have already proven theoretical efficiency.

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

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

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

Co-Training Principle

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

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

Description of the algorithm

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

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

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

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

return $Text\_features(x)$

$vectors = \emptyset$

Implementation

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

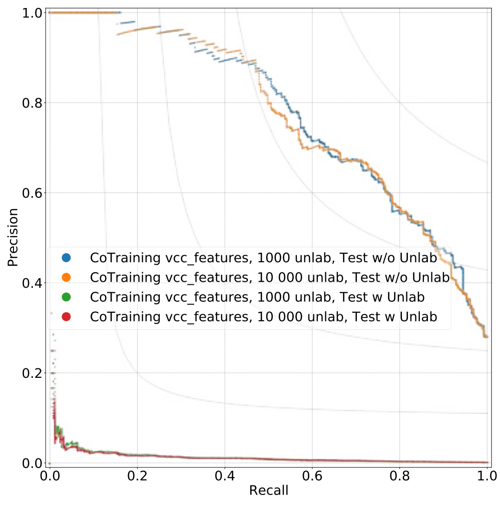


Figure 4.6 : Co-Training Performance using VCC Features' set

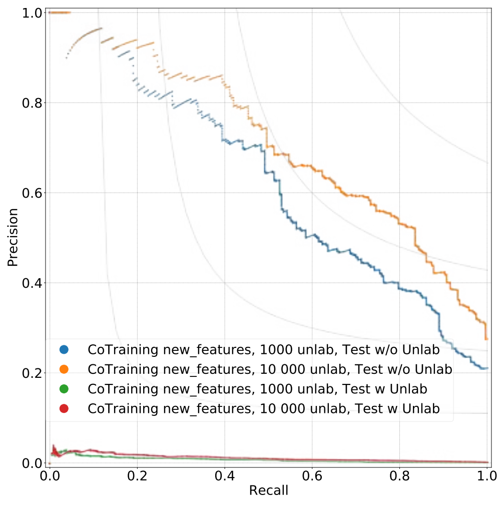


Figure 4.7 : Co-Training Performance using New Features set

unlabelled commits added.

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

Co-Training Results

Co-Training with VCC Features

Performance is improved slightly (cf. Figure 4.6 vs Figure 4.2) when Co-Training is used in conjunction with VCC Features. This improvement, however, does not appear to change with the size increase of the training set (whether 1000 or 10 000).

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

Co-Training with New Features

Figure 4.7 presents the results for a Co-Training process based on New Features. It includes variations for the training set (with 1000 and 10 000 unclassified commits) and, tests with and without the unclassified commits. On testing without the unlabelled Test set, one can conclude that the increase of 1000 unlabelled already helps perform better than the baseline green curve of Figure 4.5. An increase of the dataset by 10 000 is further contributing to detect more VCCs.

Co-Training Analysis

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

This finding is clear when we consider the unclassified commits, in which cases the performance metrics dramatically drop. There seems to be an effect, though, for the New Features when only considering the Ground Truth.

4.3 Difference with related work

Many works are related to this study (Chapter 4), as we have already explored the related works, in this part, we highlight the main differences between our work and the closest others. Predicting the suspicious commits through filtering them by excluding or including those matching a list of keywords is an axis in the vulnerability-introducing detection area. For example, (Wang, 2019) proposes an approach that falls in this axis and their filtering step can discard up to 92% of commits. However, in the main cases, the artifacts of their works are not available for comparison and then, cannot be used as a baseline for the research community. In contrast, our reproduction/replication work proposes a new baseline that might help unlock the field and share a clean, fully reproducible pipeline including artifacts.

Another closest work is (Zhou et Sharma, 2017) that compare different algorithms for automatically discovering security issues. They consider information from the commit message, gathered using regular expressions, and from bug reports. In opposition to VCCFinder and our baseline approach, no information is taken from the patch code itself.

4.4 Summary

Vulnerability detection is a key challenge in software development projects. Ideally, vulnerabilities should be discovered when they are being introduced, i.e., by flagging the suspicious vulnerability-contributing commits. VCCFinder, presented in 2015 at the CCS flagship security conference held the promise of detecting vulnerability-contributing commits at scale using machine learning. Since the research direction that this approach initiated has not boomed since then, we have proposed to revisit it. First, we attempted (and failed) to replicate the approach and to replicate the results. Then, we propose to build an alternative approach for the detection of vulnerability-contributing commits using a new feature sets (whose extraction is clearly replicable) and a semi-supervised learning technique based on co-training to account for the existence of a large set of unlabelled commits. Our experimental results indicate that the proposed approach does not yield as good performance as the ones reported in the VCC-Finder publication. Nevertheless, it constitutes a strong and reproducible baseline for the research community. Our artefacts are publicly available at `https://github.com/Trustworthy-Software/RevisitingVCCFinder`

CHAPITRE V

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

Ensuring software security is of the utmost importance during software evolution, and developers are indeed increasingly concerned about secure development. Yet, there are regular reports of successful attacks on applications. Typically, these attacks exploit vulnerabilities (also referred to as "security-sensitive bugs") in the application or the operating system code. Such attacks are further feasible when they are due to zero-day vulnerabilities, i.e., computer security flaws that the software or service provider is not yet aware of or that have not yet been patched. Such vulnerabilities can indeed easily go unnoticed by legitimate parties, thus increasing the risk of attacks.

This thesis works addresses challenges towards the reduction of zero-day vulnerabilities by investigating the automation of the detection of patches that introduce or fix vulnerabilities : (1) we proposed SSPCatcher, a semi-supervised approach for vulnerability-fixing patches detection by coping with the insufficiency of labeled data ; (2) we replicate and comprehensively assess the state of the art VCCFinder in vulnerability-introducing patch detection and propose an alternative approach.

Vulnerability-fixing patch identification : In our first contribution, we have designed an automated approach that predicts when a fixing change should be as being security-relevant. Concretely, we have investigated the problem of identifying security patches, i.e., patches that address security issues in a codebase. We have proposed SSPCatcher, a Co-Training-based approach to catch security patches as part of an automatic monitoring service of code repositories. The Co-Training approach is developed to cope with the fact that most samples in our training set is unlabeled (we do not know whether they are security-relevant or not). It has been demonstrated experimentally to be more effective than prior state-of-the-art works. We further show experimentally that this performance is not only due to the suitability of our feature set but also to the effectiveness of the Co-Training algorithm. Finally, experiments on unlabeled patches show that our model can help uncover silent fixes of vulnerabilities.

This contribution is beneficial for both practitioners and researchers interested in vulnerability-fixing patches detection because we propose an automatic approach that (1) facilitates maintainers' work by alerting when a given fixing patch is security relevant or not ; (2) avoids "follow-up" attacks on silent fixing patches ; and (3) alerts legitimate stakeholders of a given project to take the right actions or to perform more investigation about a flagged vulnerability-fixing patch. In addition, by proposing an accurate approach that outperforms the state of the art, we believe that this could give more confidence to both practitioners and researchers in the using SSPCatcher.

Vulnerability-introducing patch identification : Our second contribution deals with the problem of vulnerability-introducing patch identification. We have first proposed to revisit VCCFinder, an influential approach for detecting vulnerability-contributing commits at scale using machine learning. After discussing the practical challenges in replicating VCCFinder, we have designed a new approach to identify vulnerability-contributing commits based on a semi-supervised learning technique with a new specialized feature set. While our results do not show that we outperform VCCFinder, the proposed approach constitutes a strong and reproducible baseline for the research community.

This contribution is useful for both practitioners and researchers interested in detecting vulnerability-introducing patches because (1) we reproduce an automatic approach that helps security or bug-fixing teams to grant the appropriate privilege to a given bug, regardless of whether it is security-sensitive or not ; (2) we prove, in line with the original VCCFinder paper (Perl *et al.*, 2015), that this approach helps to avoid security attacks based on software vulnerabilities by identifying patches that are flagged as security-relevant ; (3) we propose an alternative and explainable approach for better interpretation of the results and to facilitate further investigation by security teams ; and due to the lack of available approaches in this area, (4) we share a fully reproducible approach for vulnerability introduction patches available to practitioners and researchers.

5.2 Discussions

Improvement/non-improvement. In vulnerability-fixing patches identification, we note that our contribution outperforms the state of the art by empirically showing that such automation is feasible and can yield a precision of over 80% in identifying security patches, with an unprecedented recall of over 80%. However, in vulnerability-introducing patches identification, our experimental results indicate that the proposed approach does not yield as good performance as the ones reported in the SSPCatcher. This could be explained by (1) the difference between the characteristics of a vulnerability-fixing and introducing patch in terms of feature set relevance ; and (2) the difference in terms of datasets constructions between these two experiments.

Finally, we note that our approaches perform 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.

Practical implications. Our contributions were designed to be readily integrated into a real-world pipeline of collaborative software development. We envision these contributions to be deployed in a typical code management system. In such systems which implement pre-commit tasks, such as "Git hook", it is possible to perform a set of processing actions on a commit before adding it to the repository. Our approaches are 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. In addition, all approaches were developed in Python and written in the form of a library so that they can be easily integrated into an existing pipeline. It could directly incorporate inputs from a pipeline and produce the necessary outputs.

Finally, we recommend practitioners that want to use these proposed approaches to (1) build a security-relevant dataset by considering the data set collection process proposed ; (2) implement the feature engineering process and train the semi-supervised model for more accuracy ; and (3) use a versioning system that implements pre-commit tasks such as with "Git hook" to perform the prediction on a commit before adding it to the repository.

5.3 Future works

For future research, the topics that could be investigated include :

— **Automated feature engineering.** Existing works on automatic vulnerability-introducing/fixing patch detection using machine learning often relies on features engineered explicitly for the task. Therefore, these works, including ours, require substantial human effort to identify relevant features. Due to the typical massive size of the datasets, manual feature engineering is an arduous task, and sometimes the easy way to leverage it is to sample. Automated feature engineering is one relevant approach that fixes the limits of manual feature engineering. Automated feature engineering suits the huge size of the datasets by automatically catching representative vectors from raw data. For example, recent research has shown that neural networks can be leveraged to learn the semantic representation of code. However, it is still hard to understand the explainability of the features after using these approaches. In future work, we could investigate how to leverage such representations for predicting the security relevance of code changes.

— **Towards a more extensive detection approach.** We have proposed approaches that aim to reduce zero-day vulnerabilities by contributing to the automated identification of vulnerability-introducing and fixing patches. We have experimented with these approaches on mainly C programming languages projects due to the lack of large-scale labeled datasets. In future work, we will investigate how to build datasets for an extended set of programming languages as well as whether transfer learning can help apply a learned detector from one language to the other.

— **Automatic vulnerability detection with a focus on categories.** Reducing patching process time is crucial when addressing security-sensitive bugs. Existing works have proposed approaches that detect as many vulnerabilities as possible, generally, without considering their severity and category. However, the scope of vulnerabilities is large, and some categories (e.g., identified by the Common Weakness Enumeration criterion) require more attention than others due to their severity or complex structure. Our future work idea is to investigate the identification of specific sets of vulnerabilities grouped according to their CWE (Common Weakness Enumeration).

For example, proposing an automatic approach that allows predicting *improper Authentication (CWE-287*[1]*)* could enable a better understanding of the problem, consider all factors and characteristics and thus lead to more refined and effective models.

.

1. https ://cwe.mitre.org/data/definitions/287.html

REFERENCES

Allix, K., Bissyandé, T. F., Klein, J. et Le Traon, Y. (2015). Are your training datasets yet relevant? Dans *International Symposium on Engineering Secure Software and Systems*, 51–67. Springer.

Alohaly, M. et Takabi, H. (2017). When do changes induce software vulnerabilities ? Dans 2017 *IEEE 3rd International Conference on Collaboration and Internet Computing (CIC)*, 59–66. IEEE.

Arnold, R. S. (1996). Software change impact analysis. IEEE Computer Society Press.

Arusoaie, A., Ciobâca, S., Craciun, V., Gavrilut, D. et Lucanu, D. (2017). A comparison of open-source static analysis tools for vulnerability detection in c/c++ code. Dans *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 161–168. IEEE.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D. et McDaniel, P. (2014). Flowdroid : Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Dans *Acm Sigplan Notices, 259–269*. ACM.

Association for Computer Machinery (2020). Artifact review and badging. Accessed November 27, 2020. Récupéré de https://www.acm.org/publications/policies/artifact-review-badging-current

Balcan, M.-F. et Blum, A. (2005). A pac-style model for learning from labeled and unlabeled data. Dans *International Conference on Computational Learning Theory*, 111–126. Springer.

Ban, X., Liu, S., Chen, C. et Chua, C. (2019). A performance evaluation of deep-learnt features for software vulnerability detection. Concurrency and Computation : Practice and Experience.

Behl, D., Handa, S. et Arora, A. (2014). A bug Mining tool to identify and analyze security bugs using Naive Bayes and TF-IDF : A Comparative Analysis. *ICROIT 2014 - Proc. 2014 Int. Conf. Reliab. Optim. Inf. Technol*.

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.

Bilge, L. et Dumitraş, T. (2012). Before we knew it : an empirical study of zero-day attacks in the real world. Dans *Proceedings of the 2012 ACM conference on Computer and communications security*, 833–844. ACM.

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

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

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

Brumley, D., Poosankam, P., Song, D. et Zheng, J. (2008). Automatic patch-based exploit generation is possible : Techniques and implications. Dans *2008 IEEE Symposium on Security and Privacy (sp 2008),* 143–157. IEEE.

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

Cadar, C. et Sen, K. (2013). Symbolic execution for software testing : three decades later. *Commun*. ACM.

Castelli, V. et Cover, T. M. (1995). On the exponential value of labeled samples. *Pattern Recognition Letters,* 16(1), 105–111.

Catarci, T., de Leoni, M., Marrella, A., Mecella, M., Salvatore, B., Vetere, G., Dustdar, S., Juszczyk, L., Manzoor, A. et Truong, H.-L. (2008). Pervasive software environments for supporting disaster responses. *IEEE Internet computing*, 12(1).

Cha, S. K., Avgerinos, T., Rebert, A. et Brumley, D. (2012). Unleashing mayhem on binary code. Dans *Security and Privacy (SP), 2012 IEEE Symposium on*, 380–394. IEEE.

Chang, R.-Y., Podgurski, A. et Yang, J. (2008). Discovering neglected conditions in software by mining dependence graphs. IEEE Transactions on Software Engineering.

Chawla, N. V., Bowyer, K. W., Hall, L. O. et Kegelmeyer, W. P. (2002). Smote : synthetic minority over-sampling technique. *Journal of artificial intelligence research.*

Cho, C. Y., Babic, D., Poosankam, P., Chen, K. Z., Wu, E. X. et Song, D. (2011). Mace : Model-inference-assisted concolic exploration for protocol and vulnerability discovery. Dans *USENIX Security Symposium, volume 139*.

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

Das, D. C. et Rahman, M. R. (2019). Security and performance bug reports identification with class-imbalance sampling and and Feature Selection . *2018 Jt. 7th Int. Conf. Informatics, Electron. Vis. 2nd Int. Conf. Imaging, Vis. Pattern Recognition*, ICIEV-IVPR 2018.

Du, X., Chen, B., Li, Y., Guo, J., Zhou, Y., Liu, Y. et 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.

Egelman, S., Herley, C. et Van Oorschot, P. C. (2013). Markets for zero-day exploits : Ethics and implications. Dans *Proceedings of the 2013 New Security Paradigms Workshop, 41–46.* ACM.

Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R. et Lin, C.-J. (2008). Liblinear : A library for large linear classification. *Journal of machine learning research*.

Farwell, J. P. et Rohozinski, R. (2011). Stuxnet and the future of cyber war.

Feng, Q., Zhou, R., Xu, C., Cheng, Y., Testa, B. et Yin, H. (2016). Scalable graph-based bug search for firmware images. Dans *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 480–491. ACM.

Ferschke, O., Gurevych, I. et Rittberger, M. (2012). Flawfinder : A modular system for predicting quality flaws in wikipedia. Dans *CLEF (Online Working Notes/Labs/Workshop*), 1–10.

Gegick, M., Rotella, P. et Xie, T. (2010). Identifying security bug reports via text mining : An industrial case study. *Proc. - Int. Conf. Softw. Eng.*

Ghaffarian, S. M. et Shahriari, H. R. (2017). Software vulnerability analysis and discovery using machine-learning and data-mining techniques : a survey, *ACM Computing Surveys (CSUR).*

Godefroid, P., Levin, M. Y., Molnar, D. A. et al. (2008). Automated whitebox fuzz testing. Dans *NDSS, volume 8, 151–166*.

Goseva-Popstojanova, K. et Tyo, J. (2018b). Identification of security related bug reports via text mining using supervised and unsupervised classification. Dans *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 344–355. IEEE.

Groß, S. (2018). FuzzIL : Coverage Guided Fuzzing for JavaScript Engines. (Mémoire de maîtrise). *Karlsruhe Institute of Technology*.

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

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

Hogan, K., Warford, N., Morrison, R., Miller, D., Malone, S. et Purtilo, J. (2019). The challenges of labeling vulnerability-contributing commits. Dans *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW),* 270–275. IEEE.

Holzmann, G. J. (2002). Uno : Static source code checking for userdefined properties. Dans *In 6th World Conf. on Integrated Design and Process Technology*, IDPT '02.

Jay, G., Hale, J. E., Smith, R. K., Hale, D. P., Kraft, N. A. et Ward, C. (2009). Cyclomatic complexity and lines of code : *Empirical evidence of a stable linear relationship. Journal of Software Engineering and Applications*.

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

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

Kersten, R., Luckow, K. S. et Pasareanu, C. S. (2017). Poster : Afl-based fuzzing for java with kelinci. Dans *ACM Conference on Computer and Communications Security*.

Kim, M. (2007). An effective under-sampling method for class imbalance data problem. Dans *Proceedings of the 8th Symposium on Advanced Intelligent Systems, 825–829*.

Kim, S., Whitehead Jr, E. J. et Zhang, Y. (2008). Classifying software changes : Clean or buggy ? *IEEE Transactions on Software Engineering*.

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

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

Kononenko, I. (1995). On biases in estimating multi-valued attributes. Dans *Ijcai,* 1034–1040. Citeseer.

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

Krogel, M.-A. et Scheffer, T. (2004). Multi-relational learning, text mining, and semi-supervised learning for functional genomics. Machine Learning. Krsul, I. V. (1998). *Software vulnerability analysis. Purdue University West Lafayette, IN.*

Larochelle, D. et Evans, D. (2001). Statically detecting likely buffer overflow vulnerabilities. Dans *10th USENIX Security Symposium*.

Levin, A., Viola, P. et Freund, Y. (2003). Unsupervised improvement of visual detectors using co-training. Dans *null, p. 626*. IEEE.

Li, B., Sun, X., Leung, H. et Zhang, S. (2013). A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*.

Li, H., Oh, J., Oh, H. et Lee, H. (2016a). Automated source code instrumentation for verifying potential vulnerabilities. Dans *IFIP International Conference on ICT Systems Security and Privacy Protection, 211–226.* Springer.

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

Li, X. et Liu, B. (2003). Learning to classify text using positive and unlabeled data. Dans *IJCAI*, 587–592. ACM.

Li, Z., Zou, D., Xu, S., Jin, H., Qi, H. et Hu, J. (2016b). Vulpecker : an automated vulnerability detection system based on code similarity analysis. Dans *Proceedings of the 32nd Annual Conference on Computer Security Applications, 201–213*. ACM.

Lin, G., Zhang, J., Luo, W., Pan, L., Xiang, Y., De Vel, O. et Montague, P. (2018). Cross-project transfer representation learning for vulnerable function discovery. *IEEE Transactions on Industrial Informatics*.

Liu, Y., Yu, X., Huang, J. X. et An, A. (2011). Combining integrated sampling with svm ensembles for learning from imbalanced datasets. *Information Processing & Management, 47(4), 617–631.*

Livshits, V. B. et Lam, M. S. (2005). Finding security vulnerabilities in java applications with static analysis. Dans *USENIX Security Symposium,* 18–18.

Mann, H. B. et Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics,* 50–60.

Martin, M., Livshits, B. et Lam, M. S. (2005). Finding application errors and security flaws using pql : a program query language. Dans *Acm Sigplan Notices, 365–383*. ACM.

McCabe, T. J. (1976). A complexity measure. IEEE Transactions on software Engineering.

Medeiros, I., Neves, N. et Correia, M. (2016). Dekant : a static analysis tool that learns to detect web application vulnerabilities. Dans *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 1–11. ACM.

Meneely, A., Srinivasan, H., Musa, A., Tejeda, A. R., Mokary, M. et Spates, B. (2013). When a patch goes bad : Exploring the properties of vulnerability-contributing commits. Dans *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement,* 65–74 IEEE.

Moshtari, S., Sami, A. et Azimi, M. (2013). Using complexity metrics to improve software security. *Computer Fraud & Security, 2013(5)*, 8–17.

Mostafa, S., Findley, B., Meng, N. et Wang, X. (2019). SAIS : Self-Adaptive Identification of Security Bug Reports. *IEEE Trans. Dependable Secur. Comput.*

Neuhaus, S., Zimmermann, T., Holler, C. et Zeller, A. (2007). Predicting vulnerable software components. Dans *ACM Conference on computer and communications security,* 529–540. Citeseer.

Newsome, J. et Song, D. X. (2005). Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. Dans *NDSS, volume 5, 3–4. Citeseer. Nguyen, A. T., Nguyen, T. T., Nguyen, H. A. et Nguyen, T. N. (2012).*

Multi-layered approach for recovering links between bug reports and fixes. Dans *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 63–71. ACM.

Nigam, K. et Ghani, R. (2000). Analyzing the effectiveness and applicability of co-training. *Dans Proceedings of the ninth international conference on Information and knowledge management, 86–93*. ACM.

NIST (2018). National vulnerability database. https://nvd.nist.gov. Orriols, A. et Bernadó-Mansilla, E. (2005). The class imbalance problem in learning classifier systems : a preliminary study. *Dans Proceedings of the 7th annual workshop on Genetic and evolutionary computation, 74–78*.

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

Pereira, M., Kumar, A. et Cristiansen, S. (2019). Identifying Security Bug Reports Based Solely on Report Titles and Noisy Data. 39–44.

Perl, H., Dechand, S., Smith, M., Arp, D., Yamaguchi, F., Rieck, K., Fahl, S. et Acar, Y. (2015). Vccfinder : Finding potential vulnerabilities in open-source projects to assist code audits. *Dans Proceedings of the 22nd ACM SIGSAC, Conference on Computer and Communications Security, 426–437*. ACM.

Peters, F., Tun, T. T., Yu, Y. et Nuseibeh, B. (2019). Text Filtering and Ranking for Security Bug Report Prediction. *IEEE Trans. Softw. Eng.*

Ponta, S. E., Plate, H., Sabetta, A., Bezzi, M. et Dangremont, C. (2019). A manually-curated dataset of fixes to vulnerabilities of open-source software. Dans *Proceedings of the 16th International Conference on Mining Software Repositories, 383–387*. IEEE Press.

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, M. F. (1980). An algorithm for suffix stripping. Program.

Ribeiro, M. T., Singh, S. et Guestrin, C. (2016). Why should i trust you ? : Explaining the predictions of any classifier. Dans *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 1135–1144. ACM.

Rieck, K., Wressnegger, C. et Bikadorov, A. (2012). Sally : A tool for embedding strings in vector spaces. Journal of Machine Learning Research. Rohrhofer, F. M., Saha, S., Di Cataldo, S.

Geiger, B. C., von der Linden, W.et Boeri, L. (2021). Importance of feature engineering and database selection in a machine learning model : A case study on carbon crystal structures. *arXiv preprint arXiv* :2102.00191.

Sabetta, A. et Bezzi, M. (2018). A practical approach to the automatic classification of security-relevant commits. Dans *34th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE.

Sawadogo, A. D., Bissyandé, T. F., Moha, N., Allix, K., Klein, J., Li, L. et Le Traon, Y. (2020). Learning to catch security patches.

Scandariato, R., Walden, J., Hovsepyan, A. et Joosen, W. (2014). Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*.

Shin, Y. et Williams, L. (2008). An empirical model to predict security vulnerabilities using code complexity metrics. *ESEM'08 Proc. 2008 ACM-IEEE Int. Symp. Empir. Softw. Eng. Meas.*

Shin, Y. et Williams, L. (2011). An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. *Dans Proceedings of the 7th International Workshop on Software Engineering for Secure Systems, 1–7*.

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

Śliwerski, J., Zimmermann, T. et Zeller, A. (2005). When do changes induce fixes ? Dans *ACM sigsoft software engineering notes, 1–5*. ACM.

Snyk.io (2017). The state of open-source security. https://snyk.io/stateofossecurity/pdf/The%20State%20of%20Open%20Source.pdf, Available : August 2018.

Sun, X., Peng, X., Zhang, K., Liu, Y. et Cai, Y. (2019). How security bugs are fixed and what can be improved : an empirical study with Mozilla. *Sci. China Inf. Sci.*

Sutton, M., Greene, A. et Amini, P. (2007). Fuzzing : brute force vulnerability discovery. Pearson Education.

Szekeres, L., Payer, M., Wei, T. et Song, D. (2013). Sok : Eternal war in memory. Dans *Security and Privacy (SP), 2013 IEEE Symposium on, 48–62*. IEEE.

Tian, Y., Lawall, J. et Lo, D. (2012). Identifying linux bug fixing patches. Dans *Proceedings of the 34th International Conference on Software Engineering, 386–396*. IEEE Press.

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

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 science & business media.

Wang, S. (2019). Leveraging machine learning to improve software reliability.

Wang, S., Chollak, D., Movshovitz-Attias, D. et Tan, L. (2016). Bugram : bug detection with n-gram language models. Dans *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 708–719*. ACM.

Wijayasekara, D., Manic, M. et McQueen, M. (2014). Vulnerability identification and classification via text mining bug databases. Dans *IECON 2014-40th Annual Conference of the IEEE Industrial Electronics Society, 3612–3618*. IEEE.

Wijayasekara, D., Manic, M., Wright, J. L. et McQueen, M. (2012). Mining bug databases for unidentified software vulnerabilities. Dans *2012 5th International Conference on Human System Interactions, 89–96*. IEEE.

Wu, R., Zhang, H., Kim, S. et Cheung, S.-C. (2011). Relink : recovering links between bugs and changes. Dans *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 15–25. ACM.

Xiao, Y., Chen, B., Yu, C., Xu, Z., Yuan, Z., Li, F., Liu, B., Liu, Y., Huo, W., Zou, W. et Shi, W. (2020). Mvp : Detecting vulnerabilities using patch-enhanced vulnerability signatures. *Dans USENIX Security Symposium.*

Yamaguchi, F., Golde, N., Arp, D. et Rieck, K. (2014a). Modeling and discovering vulnerabilities with code property graphs. Dans *Security and Privacy (SP), 2014 IEEE Symposium on, 590–604*. IEEE.

Yamaguchi, F., Golde, N., Arp, D. et Rieck, K. (2014b). Modeling and discovering vulnerabilities with code property graphs. *2014 IEEE Symposium on Security and Privacy, 590–604.*

Yamaguchi, F., Wressnegger, C., Gascon, H. et Rieck, K. (2013). Chucky : Exposing missing checks in source code for vulnerability discovery. *Dans Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, 499–510*. ACM.

Yamamoto, K. (2018). Vulnerability detection in source code based on git history.

Yang, L., Li, X. et Yu, Y. (2017). Vuldigger : A just-in-time and cost-aware tool for digging vulnerability-contributing changes. Dans *GLOBECOM 2017- 2017 IEEE Global Communications Conference,* 1–7. http://dx.doi.org/10.1109/GLOCOM.2017.8254428.

Ying, A. T., Murphy, G. C., Ng, R. et Chu-Carroll, M. C. (2004). Predicting source code changes by mining change history. *IEEE transactions on Software Engineering*.

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

Zhang, T. et Oles, F. (2000). The value of unlabeled data for classification problems. Dans *Proceedings of the Seventeenth International Conference on Machine Learning,(Langley, P., ed.), volume 20, p. 0. Citeseer*.

Zhang, Y.-P., Zhang, L.-N. et Wang, Y.-C. (2010). Cluster-based majority under-sampling approaches for class imbalance learning. Dans *2010 2nd IEEE International Conference on Information and Financial Engineering, 400–404*. IEEE.

Zhou, Y., Liu, S., Siow, J., Du, X. et Liu, Y. (2019). Devign : Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Dans *NeurIPS*.

Zhou, Y. et Sharma, A. (2017). Automated identification of security issues from commit messages and bug reports. Dans *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 914–919. ACM.

Zhu, X., Feng, X., Jiao, T., Wen, S., Xiang, Y., Camtepe, S. et Xue, J. (2019). A feature-oriented corpus for understanding, evaluating and improving fuzz testing. Dans *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 658–663.

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

Zou, D., Deng, Z., Li, Z. et Jin, H. (2018). Information Security and Privacy - *23rd Australasian Conference, {ACISP} 2018, Wollongong, NSW, Australia, July 11-13, 2018, Proceedings. Springer International Publishing*.