THE UNIVERSITY
of ADELAIDE

Towards an Improved Understanding of Software Vulnerability Assessment Using Data-Driven Approaches

Author: **Triet Huynh Minh Le**
Centre for Research on Engineering Software Technologies (CREST)
School of Computer Science
Faculty of Engineering, Computer and Mathematical Sciences
The University of Adelaide

Principal Supervisor: Professor Muhammad Ali Babar
Co-Supervisor: Professor Cheng-Chew Lim

A thesis submitted for the degree of
DOCTOR OF PHILOSOPHY
The University of Adelaide

March 21, 2022

Contents

List of Figures vii

List of Tables ix

Abstract xi

Declaration of Authorship xiii

Acknowledgements xv

Dedication xvii

1 Introduction 1
- 1.1 Problem Statement and Research Objectives . . . 2
- 1.2 Thesis Overview and Contributions . . . 3
- 1.3 Related Publications . . . 5
- 1.4 Thesis Organization . . . 6

2 Literature Review on Data-Driven Software Vulnerability Assessment 7
- 2.1 Introduction . . . 8
- 2.2 Overview of the Literature Review . . . 9
  - 2.2.1 Scope . . . 9
  - 2.2.2 Methodology . . . 9
  - 2.2.3 Taxonomy of Data-Driven Software Vulnerability Assessment . . . 10
- 2.3 Exploitation Prediction . . . 11
  - 2.3.1 Summary of Primary Studies . . . 13
    - 2.3.1.1 Exploit Likelihood . . . 13
    - 2.3.1.2 Exploit Time . . . 14
    - 2.3.1.3 Exploit Characteristics . . . 15
  - 2.3.2 Theme Discussion . . . 17
- 2.4 Impact Prediction . . . 18
  - 2.4.1 Summary of Primary Studies . . . 18
    - 2.4.1.1 Confidentiality, Integrity, Availability, and Scope . . . 18
    - 2.4.1.2 Custom Vulnerability Consequences . . . 19
  - 2.4.2 Theme Discussion . . . 19
- 2.5 Severity Prediction . . . 20
  - 2.5.1 Summary of Primary Studies . . . 20
    - 2.5.1.1 Severe vs. Non-Severe . . . 20
    - 2.5.1.2 Severity Levels . . . 21
    - 2.5.1.3 Severity Score . . . 22
  - 2.5.2 Theme Discussion . . . 23
- 2.6 Type Prediction . . . 23
  - 2.6.1 Summary of Primary Studies . . . 23
    - 2.6.1.1 Common Weakness Enumeration (CWE) . . . 23

2.6.1.2 Custom Vulnerability Types . . . 25
2.6.2 Theme Discussion . . . 26
2.7 Miscellaneous Tasks . . . 26
2.7.1 Summary of Primary Studies . . . 26
2.7.1.1 Vulnerability Information Retrieval . . . 26
2.7.1.2 Cross-Source Vulnerability Patterns . . . 28
2.7.1.3 Vulnerability Fixing Effort . . . 28
2.7.2 Theme Discussion . . . 29
2.8 Analysis of Data-Driven Approaches for Software Vulnerability Assessment . 29
2.8.1 Data Sources . . . 29
2.8.2 Model Features . . . 31
2.8.3 Prediction Models . . . 32
2.8.4 Evaluation Techniques . . . 33
2.8.5 Evaluation Metrics . . . 33
2.9 Chapter Summary . . . 34
2.10 Appendix - Ever-Growing Literature on Data-Driven SV Assessment . . . 35

3 Automated Report-Level Software Vulnerability Assessment with Concept Drift 37
3.1 Introduction . . . 38
3.2 Background . . . 39
3.3 The Proposed Approach . . . 40
3.3.1 Approach Overview . . . 40
3.3.2 Text Preprocessing of SV Descriptions . . . 41
3.3.3 Model Selection with Time-based k-Fold Cross-Validation . . . 41
3.3.4 Feature Aggregation Algorithm . . . 43
3.4 Experimental Design and Setup . . . 44
3.4.1 Research Questions . . . 44
3.4.2 Dataset . . . 45
3.4.3 Machine Learning Models for Report-Level SV Assessment . . . 45
3.4.4 Evaluation Metrics . . . 46
3.5 Experimental Results and Discussion . . . 47
3.5.1 **RQ1**: Is Our Time-Based Cross-Validation More Effective Than a Non-Temporal Method to Handle Concept Drift in The Model Selection Step for Report-Level SV Assessment? . . . 47
3.5.2 **RQ2**: Which are the Optimal Models for Multi-Classification of Each SV Characteristic? . . . 49
3.5.3 **RQ3**: How Effective is Our Character-Word Model to Perform Automated Report-Level SV Assessment with Concept Drift? . . . 51
3.5.4 **RQ4**: To What Extent Can Low-Dimensional Model Retain the Original Performance? . . . 55
3.6 Threats to Validity . . . 56
3.7 Related Work . . . 57
3.7.1 Report-Level SV Assessment . . . 57
3.7.2 Temporal Modeling of SVs . . . 57
3.8 Chapter Summary . . . 58
3.9 Appendix - SVs with All Out-of-Vocabulary Words . . . 58

4 Automated Function-Level Software Vulnerability Assessment 59
4.1 Introduction . . . 60
4.2 Background and Motivation . . . 61
4.3 Research Questions . . . 64
4.4 Research Methodology . . . 64
4.4.1 Data Collection . . . 65
4.4.2 Vulnerable Code Context Extraction . . . 66
4.4.3 Code Feature Generation . . . 67
4.4.4 Data-Driven SV Assessment Models . . . 69
4.4.5 Model Evaluation . . . 69
4.5 Results . . . 70
4.5.1 **RQ1**: Are Vulnerable Code Statements More Useful Than Non-Vulnerable Counterparts for SV Assessment Models? . . . 70
4.5.2 **RQ2**: To What Extent do Different Types of Context of Vulnerable Statements Contribute to SV Assessment Performance? . . . 71
4.5.3 **RQ3**: Does Separating Vulnerable Statements and Context to Provide Explicit Location of SVs Improve Assessment Performance? . . . 74
4.6 Discussion . . . 75
4.6.1 Function-Level SV Assessment: Baseline Models and Beyond . . . 75
4.6.2 Threats to Validity . . . 77
4.7 Related Work . . . 77
4.7.1 Code Granularities of SV Detection . . . 77
4.7.2 Data-Driven SV Assessment . . . 78
4.8 Chapter Summary . . . 78

5 Automated Commit-Level Software Vulnerability Assessment 79
5.1 Introduction . . . 80
5.2 Background and Motivation . . . 81
5.2.1 Vulnerability in Code Commits . . . 81
5.2.2 Commit-Level SV Assessment with CVSS . . . 81
5.2.3 Feature Extraction from Commit Code Changes . . . 82
5.3 The DeepCVA Model . . . 84
5.3.1 Commit Preprocessing, Context Extraction & Tokenization . . . 84
5.3.2 Feature Extraction with Deep AC-GRU . . . 85
5.3.3 Commit-Level SV Assessment with Multi-task Learning . . . 87
5.4 Experimental Design and Setup . . . 88
5.4.1 Datasets . . . 88
5.4.2 Evaluation Metrics . . . 90
5.4.3 Hyperparameter and Training Settings of DeepCVA . . . 91
5.4.4 Baseline Models . . . 91
5.5 Research Questions and Experimental Results . . . 92
5.5.1 **RQ1**: How does DeepCVA Perform Compared to Baseline Models for Commit-level SV Assessment? . . . 92
5.5.2 **RQ2**: What are the Contributions of the Main Components in DeepCVA to Model Performance? . . . 94
5.5.3 **RQ3**: What are the Effects of Class Rebalancing Techniques on Model Performance? . . . 96
5.6 Discussion . . . 97
5.6.1 DeepCVA and Beyond . . . 97
5.6.2 Threats to Validity . . . 98
5.7 Related Work . . . 98

5.7.1 Data-Driven SV Prediction and Assessment . . . 98
5.7.2 SV Analytics in Code Changes . . . 99
5.8 Chapter Summary . . . 99

6 Collection and Analysis of Developers' Software Vulnerability Concerns on Question and Answer Websites 101
6.1 Introduction . . . 102
6.2 Related Work . . . 103
6.2.1 Topic Modeling on Q&A Websites . . . 103
6.2.2 SV Assessment Using Open Sources . . . 103
6.3 Research Method . . . 103
6.3.1 Research Questions . . . 103
6.3.2 Software Vulnerability Post Collection . . . 105
6.3.3 Topic Modeling with LDA . . . 109
6.4 Results . . . 110
6.4.1 **RQ1**: What are SV Discussion Topics on Q&A Sites? . . . 110
6.4.2 **RQ2**: What are the Popular and Difficult SV Topics on Q&A Sites? 113
6.4.3 **RQ3**: What is the Level of Expertise to Answer SV Questions on Q&A Sites? . . . 114
6.4.4 **RQ4**: What Types of Answers are Given to SV Questions on Q&A Sites? . . . 116
6.5 Discussion . . . 117
6.5.1 SV Discussion Topics on Q&A Sites vs. Existing Security Taxonomies 117
6.5.2 Implications for (Data-Driven) SV Assessment . . . 118
6.5.3 Threats to Validity . . . 120
6.6 Chapter Summary . . . 120
6.7 Appendix - PUMiner Overview . . . 120
6.7.1 PUMiner - A Context-aware Two-stage PU Learning Model for Retrieving Security Q&A Posts . . . 120

7 Conclusions and Future Work 123
7.1 Summary of Contributions and Findings . . . 124
7.1.1 A Systematization of Knowledge of Data-Driven SV Assessment . . . 124
7.1.2 Automated Report-Level Assessment for Ever-Increasing SVs . . . 124
7.1.3 Automated Early SV Assessment using Code Functions . . . 125
7.1.4 Automated Just-in-Time SV Assessment using Code Commits . . . 125
7.1.5 Insights of Developers' Real-World Concerns on Question and Answer Websites for Data-Driven SV Assessment . . . 126
7.2 Opportunities for Future Research . . . 126
7.2.1 Integration of SV Data on Issue Tracking Systems . . . 126
7.2.2 Improving Data Efficiency for Data-Driven SV Assessment . . . 127
7.2.3 Customized Data-Driven SV Assessment . . . 128
7.2.4 Enhancing Interpretability of SV Assessment Models . . . 128
7.2.5 Data-Driven SV Assessment in Data-Driven Systems . . . 129

References 130

List of Figures

1.1 Phases in a software vulnerability lifecycle. 2
1.2 Overview of the thesis. 3

2.1 Taxonomy of studies on data-driven software vulnerability assessment. . . . 11

3.1 Frequencies of each class of the seven vulnerability characteristics. 39
3.2 Workflow of our proposed model for report-level software vulnerability assessment with *concept drift*. 40
3.3 Our proposed time-based cross-validation method. 42
3.4 The number of new terms from 2000 to 2017 of SV descriptions in National Vulnerability Database. 47
3.5 Examples of new terms in National Vulnerability Database corresponding to new products, software, cyber-attacks from 2000 to 2018. 48
3.6 Performance differences between the validated and testing *Weighted F1-Scores* of our time-based validation and a normal cross-validation methods. 49
3.7 Average cross-validated *Weighted F1-Score*s comparison between ensemble and single models for each vulnerability characteristic. 52
3.8 The relationship between the size of vocabulary and the maximum number of character n-grams. 53

4.1 A vulnerable function extracted from the fixing commit *b38a1b3* of a software vulnerability (CVE-2017-1000487) in the *Plexus-utils* project. 61
4.2 Methodology used to answer the research questions. 63
4.3 Class distributions of the seven CVSS metrics. 66
4.4 Proportions of different types of lines in a function. 71
4.5 Differences in testing software vulnerability assessment performance (F1-Score and MCC) between models using different types of lines/context and those using only vulnerable statements. 73
4.6 Average performance (MCC) of six classifiers and five features for software vulnerability assessment in functions. 76

5.1 Exemplary software vulnerability fixing commit for the XML external entity injection (XXE) (CVE-2016-3674) and its respective software vulnerability contributing commit in the *xstream* project. 82
5.2 Workflow of DeepCVA for automated commit-level software vulnerability assessment. 83
5.3 Code changes outside of a method from the commit *4b9fb37* in the *Apache qpid-broker-j* project. 85
5.4 Class distributions of seven software vulnerability assessment tasks. 89
5.5 Time-based splits for training, validating & testing. 90
5.6 Differences of testing MCC of the model variants compared to the proposed DeepCVA. 95

6.1 Workflow of retrieving posts related to software vulnerability on Q&A websites using tag-based and content-based filtering heuristics. 107
6.2 Popularity and difficulty of 13 software vulnerability topics on Stack Overflow and Security StackExchange. 113
6.3 Topic correlations between software vulnerability questions & answerers' software vulnerability specific knowledge on Stack Overflow & Security StackExchange. 116

List of Tables

2.1 Comparison of contributions between our review and the existing related surveys/reviews. . . . 9
2.2 Inclusion and exclusion criteria for study selection. . . . 10
2.3 List of the reviewed papers in the *Exploit Likelihood* sub-theme of the *Exploitation* theme. . . . 12
2.4 List of the reviewed papers in the *Exploit Time* sub-theme of the *Exploitation* theme. . . . 14
2.5 List of the reviewed papers in the *Exploit Characteristics* sub-theme of the *Exploitation* theme. . . . 16
2.6 List of the reviewed papers in the *Impact* theme. . . . 18
2.7 List of the reviewed papers in the *Severe vs. Non-Severe* sub-theme of the *Severity* theme. . . . 20
2.8 List of the reviewed papers in the *Severity Levels* sub-theme of the *Severity* theme. . . . 21
2.9 List of the reviewed papers in the *Severity Score* sub-theme of the *Severity* theme. . . . 22
2.10 List of the reviewed papers in the *Type* theme. . . . 24
2.11 List of the reviewed papers in the *Miscellaneous Tasks* theme. . . . 27
2.12 The frequent data sources, features, models, evaluation techniques and evaluation metrics used for the five identified SV assessment themes. . . . 30
2.13 The mapping between the themes/tasks and the respective studies collected from May 2021 to February 2022. . . . 35

3.1 Word and character n-grams extracted from the sentence "Hello World". '_' represents a space. . . . 41
3.2 The eight configurations of Natural Language Processing representations used for model selection. . . . 42
3.3 Optimal hyperparameters found for each classifier. . . . 50
3.4 Optimal models and results after the validation step. . . . 50
3.5 Average cross-validated *Weighted F-scores* of term frequency vs. tf-idf grouped by six classifiers. . . . 51
3.6 Average cross-validated *Weighted F-scores* of uni-gram vs. n-grams ($2 \leq n \leq 4$) grouped by six classifiers. . . . 51
3.7 *P*-values of H_0: Ensemble models $\leq$ Single models for each vulnerability characteristic. . . . 52
3.8 Performance (*Accuracy*, *Macro F1-Score*, *Weighted F1-Score*) of our character-word vs. word-only and character-only models. . . . 54
3.9 *Weighted F1-Scores* of our original Character-Word Model, 300-dimension Latent Semantic Analysis (LSA-300), fastText trained on SV descriptions (fastText-300) and fastText trained on English Wikipedia pages (fastText-300W). . . . 56

4.1 Hyperparameter tuning for software vulnerability assessment models. . . . 69

4.2 Testing performance for software vulnerability assessment tasks of vulnerable vs. non-vulnerable statements. 72
4.3 Differences in testing performance for software vulnerability assessment tasks between double-input models (RQ3) and single-input models (RQ1/2). . . . 75

5.1 The number of commits and projects after each filtering step. 90
5.2 Testing performance of DeepCVA and baseline models. 93
5.3 Testing performance (MCC) of optimal baselines using oversampling techniques and multi-task DeepCVA. 97

6.1 Content-based thresholds ($a_{\text{SO/SSE}}$ & $b_{\text{SO/SSE}}$) for the two steps of the content-based filtering. 108
6.2 The obtained software vulnerability posts using our tag-based and content-based filtering heuristics. 108
6.3 Top-5 tags of software vulnerability, security and general posts on Stack Overflow and Security StackExchange. 109
6.4 Software vulnerability topics on Stack Overflow and Security StackExchange identified by Latent Dirichlet Allocation along with their proportions and trends over time. 110
6.5 General expertise in terms of average reputation of each topic on Stack Overflow and Security StackExchange. 115
6.6 Answer types of software vulnerability discussions identified on Q&A websites. 117
6.7 Top-1 answer types of 13 software vulnerability topics on Stack Overflow & Security StackExchange. 117
6.8 The mapping between 13 software vulnerability topics and their Common Weakness Enumeration (CWE) values. 119

Abstract

Software Vulnerabilities (SVs) can expose software systems to cyber-attacks, potentially causing enormous financial and reputational damage for organizations. There have been significant research efforts to detect these SVs so that developers can promptly fix them. However, fixing SVs is complex and time-consuming in practice, and thus developers usually do not have sufficient time and resources to fix all SVs at once. As a result, developers often need SV information, such as exploitability, impact, and overall severity, to prioritize fixing more critical SVs. Such information required for fixing planning and prioritization is typically provided in the *SV assessment* step of the SV lifecycle. Recently, data-driven methods have been increasingly proposed to automate SV assessment tasks. However, there are still numerous shortcomings with the existing studies on data-driven SV assessment that would hinder their application in practice.

This PhD thesis aims to contribute to the growing literature in data-driven SV assessment by investigating and addressing the constant changes in SV data as well as the lacking considerations of source code and developers' needs for SV assessment that impede the practical applicability of the field. Particularly, we have made the following five contributions in this thesis. (**1**) We systematize the knowledge of data-driven SV assessment to reveal the best practices of the field and the main challenges affecting its application in practice. Subsequently, we propose various solutions to tackle these challenges to better support the real-world applications of data-driven SV assessment. (**2**) We first demonstrate the existence of the concept drift (changing data) issue in descriptions of SV reports that current studies have mostly used for predicting the Common Vulnerability Scoring System (CVSS) metrics. We augment report-level SV assessment models with subwords of terms extracted from SV descriptions to help the models more effectively capture the semantics of ever-increasing SVs. (**3**) We also identify that SV reports are usually released after SV fixing. Thus, we propose using vulnerable code to enable earlier SV assessment without waiting for SV reports. We are the first to use Machine Learning techniques to predict CVSS metrics on the function level leveraging vulnerable statements directly causing SVs and their context in code functions. The performance of our function-level SV assessment models is promising, opening up research opportunities in this new direction. (**4**) To facilitate continuous integration of software code nowadays, we present a novel deep multi-task learning model, DeepCVA, to simultaneously and efficiently predict multiple CVSS assessment metrics on the commit level, specifically using vulnerability-contributing commits. DeepCVA is the first work that enables practitioners to perform SV assessment as soon as vulnerable changes are added to a codebase, supporting just-in-time prioritization of SV fixing. (**5**) Besides code artifacts produced from a software project of interest, SV assessment tasks can also benefit from SV crowdsourcing information on developer Question and Answer (Q&A) websites. We automatically retrieve large-scale security/SV-related posts from these Q&A websites. We then apply a topic modeling technique on these posts to distill developers' real-world SV concerns that can be used for data-driven SV assessment. Overall, we believe that this thesis has provided evidence-based knowledge and useful guidelines for researchers and practitioners to automate SV assessment using data-driven approaches.

Declaration of Authorship

I certify that this work contains no material which has been accepted for the award of any other degree or diploma in my name, in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. In addition, I certify that no part of this work will, in the future, be used in a submission in my name, for any other degree or diploma in any university or other tertiary institution without the prior approval of the University of Adelaide and where applicable, any partner institution responsible for the joint-award of this degree.

I acknowledge that copyright of published works contained within this thesis resides with the copyright holder(s) of those works.

I also give permission for the digital version of my thesis to be made available on the web, via the University's digital research repository, the Library Search and also through web search engines, unless permission has been granted by the University to restrict access for a period of time.

I acknowledge the support I have received for my research through the provision of University of Adelaide International Wildcard Scholarship.

Triet Huynh Minh Le

March 2022

Acknowledgements

This thesis would not have been possible without continuous support, guidance, and encouragement from many people and entities. I would like to hereby acknowledge them.

Firstly, I express my deepest gratitude to my principal supervisor, Professor M. Ali Babar, for giving me a valuable opportunity to conduct PhD research under his supervision. His constructive feedback has motivated me to continuously reflect and improve myself to become a better researcher and a more-rounded person in life. With his kind patience and persistent guidance, I have also managed to navigate myself through the challenging COVID-19 pandemic and complete my PhD research to the best of my ability. Besides research, he has also given me great opportunities to engage in numerous teaching and supervision activities that have tremendously helped me to enhance my communication and interpersonal skills. All in all, working under his mentorship has profoundly transformed me and enabled me to go beyond my limits and better prepare myself for my future career.

Secondly, I sincerely thank my co-supervisor, Professor Cheng-Chew Lim for providing insightful comments on the research carried out in this thesis.

Thirdly, I am extremely grateful to many current/former members in the Centre for Research on Engineering Software Technologies (CREST) at the University of Adelaide. Special thanks to Faheem Ullah, Chadni Islam, Bushra Sabir, Aufeef Chauhan, Huaming Chen, Bakheet Aljedaani, Mansooreh Zahedi, Hao Chen, Roland Croft, David Hin, and Mubin Ul Haque for not only academic contributions and feedback on the research papers related to this thesis, but also for being wonderful colleagues from whom I have learned a lot. Specifically, I cannot give enough appreciation to Roland Croft and David Hin for their great technical insights and contributions to improve the quality of many research endeavours I have pursued during my PhD. In addition, I am happy to be accompanied by Faheem Ullah and Aufeef Chauhan during weekly Friday dinners, which has helped me to relax and recharge each week. I also appreciate Nguyen Khoi Tran for introducing me to the CREST family. Thank you all for making my PhD journey memorable.

Fourthly, I have also had the chance to collaborate with and learn from many world-class researchers outside of CREST such as Xuanyu Duan, Mengmeng Ge, Shang Gao, and Xuequan Lu. I am also thankful for all the constructive feedback from the paper and thesis reviewers that helped significantly improve the research conducted in the thesis.

Fifthly, I fully acknowledge the University of Adelaide for providing me with the University of Adelaide International Wildcard Scholarship and world-class facilities that have supported me to pursue my doctoral research and activities.

Sixthly, I highly appreciate the Urbanest at the University of Adelaide for providing me with the best-conditioned accommodation that I can ever ask for so that I can enjoy my personal life and recharge after working hours during my PhD. I am also extremely fortunate that Urbanest has also given me sufficient facilities to work effectively from home during the pandemic. I also want to deeply thank my roommates, especially Zach Li, for cheering me up during my down days.

Seventhly, I am greatly appreciative of my ASUS laptop for always being my reliable companion and working restlessly to enable me to obtain experimental results as well as write research papers and this thesis in a timely manner.

Finally and most importantly, I am immensely and eternally indebted to my family, especially my grandmother, mother, and father, who always stand by my side during the ups and downs of my PhD. Without their constant support and unconditional caring, I would not have been able to pursue my dreams and be where I am now. I love all of you from the bottom of my heart.

I would like to dedicate this thesis to my parents.

Chapter 1

Introduction

Software has become an integral part of the modern world [1]. Software systems are rapidly increasing in size and complexity. For example, the whole software ecosystems at Google, which host many popular applications/services such as Google Search, YouTube, and Google Maps, contain more than two million lines of code [2]. Quality assurance of such large systems is a focal point for both researchers and practitioners to minimize disruptions to millions of people around the world.

Software Vulnerabilities (SVs)[1] have been long-standing issues that negatively affect software quality [3]. SVs are security bugs that are detrimental to the confidentiality, integrity and availability of software systems, potentially resulting in catastrophic cyber-security attacks [4]. The exploitation of these SVs such as the Heartbleed [5] or Log4j [6] attacks can damage the operations and reputation of millions of software systems and organizations globally. These cyber-attacks caused by SVs have led to huge financial losses as well. According to the Australian Cyber Security Centre, a loss of more than 30 billion dollars due to cyber-attacks have been reported worldwide from 2020 to 2021 [7]. Therefore, it is important to remediate critical SVs as promptly as possible.

In practice, different types of SVs have varying levels of security threats to software-intensive systems [8]. However, fixing all SVs at the same time is not always practical due to limited resources and time [9]. A common practice in this situation is to prioritize fixing SVs posing imminent and serious threats to a system of interest. Such fixing prioritization usually requires inputs from SV assessment [10, 11].

The *SV assessment* phase is between the SV *discovery/detection* and SV *remediation/mitigation/fixing/patching* phases in the SV management lifecycle [12], as shown in Fig. 1.1. The *assessment* phase first unveils the characteristics of the SVs found in the *discovery* phase to locate "hot spots" that contain many highly critical/severe SVs and require higher attention in a system. Practitioners then use the assessment outputs to devise an optimal remediation plan, i.e., the order/priority of fixing each SV, based on available human and technological resources. For example, an identified cross-site scripting (XSS) or SQL injection vulnerability in a web application will likely require an urgent remediation plan. These two types of SVs are well-known and can be easily exploited by attackers to gain unauthorized access and compromise sensitive data/information. On the other hand, an SV that requires admin access or happens only in a local network will probably have a lower priority since only a few people can initiate an attack. According to the plan devised in the assessment phase, SVs would be prioritized for fixing in the *remediation* phase. In practice, the tasks in the SV assessment phase for ever-increasing SVs are repetitive and time-consuming, and thus require automation to save time and effort for practitioners.

Given the increasing size and complexity of software systems nowadays, automation of SV assessment tasks has attracted significant attention in the Software Engineering community. Traditionally, static analysis tools have been the de-facto approach to SV assessment [13]. These tools rely on pre-defined rules to determine SV characteristics.

[1] In this thesis, "software vulnerability" and "security vulnerability" are used interchangeably.

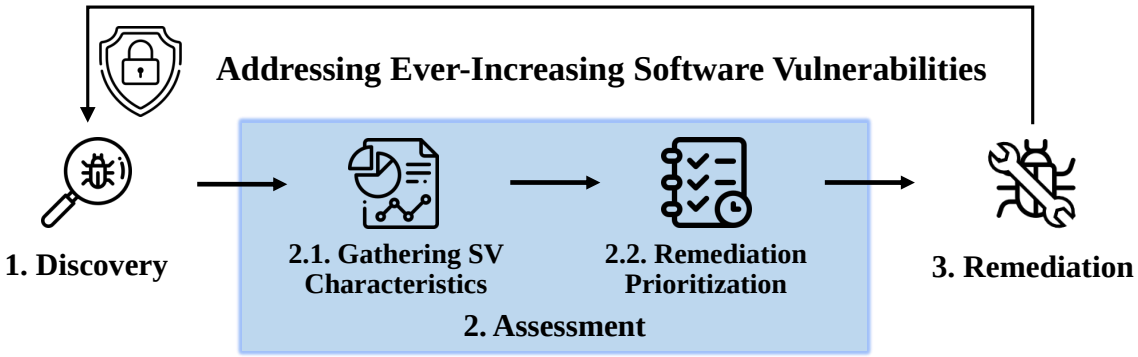


FIGURE 1.1: Phases in an SV lifecycle. **Note**: The main focus of this thesis is SV assessment.

However, these pre-defined rules require significant expertise and effort to define and may be easily error-prone [14]. In addition, these rules need manual modifications and extensions to adapt to ever-changing patterns of new SVs [15]. Such manual changes do not scale well with the fast-paced growth of SVs, potentially leading to delays in SV assessment and in turn untimely SV mitigation. Hence, there is an apparent need for automated techniques that can perform SV assessment without using manually-defined rules.

In the last decade, data-driven techniques have emerged as a promising alternative to static analysis counterparts for SV assessment, as indicated in our extensive review [11] (to be discussed in-depth in Chapter 2). The emergence of data-driven SV assessment is mainly because of the rapid increase in size of SV data in the wild (e.g., more than 170,000 SVs were reported on National Vulnerability Database (NVD) [16] from 2002 to 2021 [17]). These approaches are underpinned by Machine Learning (ML), Deep Learning (DL) and Natural Language Processing (NLP) models that are capable of automatically extracting complex patterns and rules from large-scale SV data, reducing the reliance on experts' knowledge. Overall, these data-driven models have opened up new opportunities for the field of automated SV assessment.

1.1 Problem Statement and Research Objectives

Data-driven models have many promising applications for SV assessment; however, according to our review on data-driven SV assessment [11], there are still several unaddressed yet important challenges/gaps that affect the practical application of this field. Specifically, this PhD thesis focuses on the three key practical challenges of data-driven SV assessment. The first challenge is the missing treatment for changing SV data over time that leads to degraded robustness and performance of assessment models. The second challenge is the lack of using rich and relevant knowledge of (vulnerable) source code that can result in untimely SV assessment. The third challenge is the negligence of incorporating developers' needs into data-driven models that limits customized SV assessment. These three challenges are captured in the problem statement of this thesis, which is stated as follows.

Problem statement: Practical applicability of SV assessment using data-driven approaches is negatively affected by the changing data of SVs along with the missing considerations/utilization of source code and developers' real-world needs. It is important to investigate, understand, and address these challenges to make data-driven SV assessment more timely and applicable in practice.

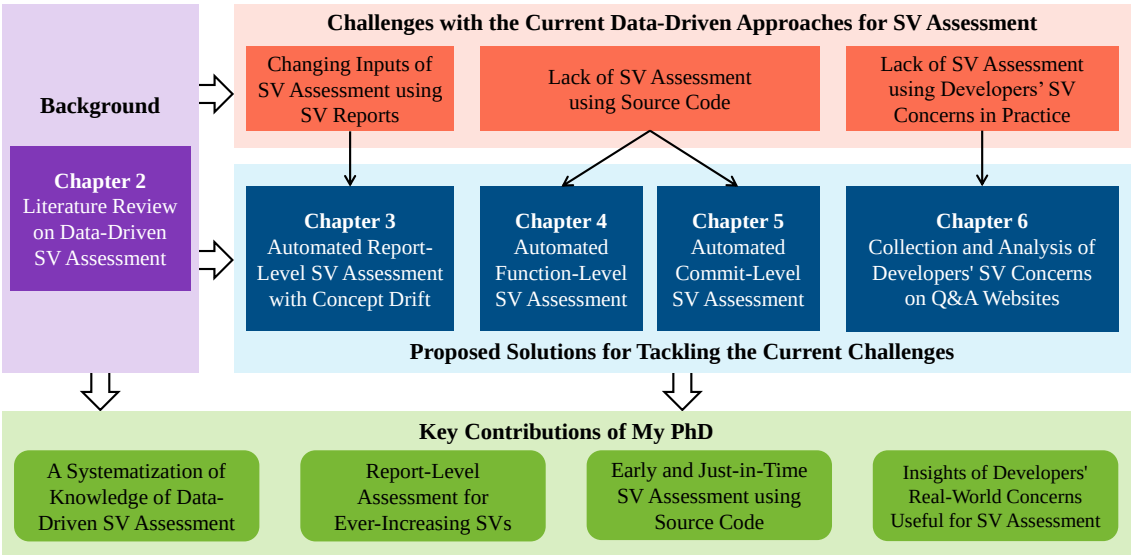


FIGURE 1.2: Overview of the thesis. **Note**: SV stands for Software Vulnerability.

The objectives of this thesis are to present and evaluate solutions to address the lack of (1) treatment for changing SV data, (2) usage of SV-related code, and (3) consideration of developers' real-world SV concerns to improve the practicality of data-driven SV assessment.

1.2 Thesis Overview and Contributions

A brief overview of the thesis is given in Fig. 1.2. The research objectives have been realized in six chapters, which are described hereafter. While I (the author of this thesis) am mainly responsible for research activities done in this thesis, most of the work was conducted in collaboration with other researchers. Thus, the pronoun "**we**" was used in this thesis to reflect the collaborative research efforts.

Chapter 2: Literature Review on Data-Driven SV Assessment

Chapter 2 (the purple box in Fig. 1.2) provides a background on the topic of data-driven SV assessment for this thesis. We conduct a literature review and use thematic analysis [18] to devise a taxonomy of the key SV assessment tasks that have been automated using data-driven approaches. For each theme of tasks, we identify the key practices that have been followed by the primary studies and highlighted the challenges with the current approaches to pave the way for future research. Particularly, we pinpoint three main challenges (the red boxes in Fig. 1.2) that potentially hinder the practical application of current data-driven approaches for SV assessment. These challenges lead to the four proposed solutions (the blue boxes in Fig. 1.2) in this thesis described in Chapters 3, 4, 5, and 6 below.

Chapter 3: Automated Report-Level SV Assessment with Concept Drift

Chapter 3 addresses the first challenge of changing data in the context of report-level SV assessment. Our review in Chapter 2 has pointed out that data-driven SV assessment tasks have been most commonly carried out using the information provided in SV reports. Each of these reports contains an expert-verified summary of an SV, e.g., a brief description of the type, the associated attack vectors and potential impacts if exploited. However, we observe that the content of these reports continuously changes in practice as experts usually need to use different/new terms to describe newly introduced SVs. Such issue is referred

to as *concept drift* [19] that can degrade the performance of SV assessment models over time. However, most of the existing report-level SV assessment models have not accounted for this concept drift issue, potentially affecting their performance when deployed in the wild. Using more than 100,000 SV reports from NVD, Chapter 3 performs a large-scale investigation of the prevalence and impacts of the concept drift issue on report-level SV assessment models. Moreover, we present a novel SV assessment model that combines characters and words extracted from SV descriptions to better capture the semantics of new terms, increasing the model robustness against concept drift.

Chapter 4: Automated Function-Level SV Assessment

Chapter 4 addresses the second challenge, i.e., the need for SV assessment using source code. Real-world SVs are usually rooted in source code, but Chapter 2 has found that the current data-driven SV assessment efforts have been mainly done on the report level. While report-level models have shown promising performance for various SV assessment tasks [20, 21, 22, 23], these models are still heavily dependent on SV reports that require expertise and manual effort to generate. Our analysis in Chapter 4 has also found that most (97%) of the reports used for SV assessment in the previous studies were not available at the fixing time of respective SVs. Such unavailability of SV reports affects the timeliness of report-level SV assessment in practice. A promising alternative is to directly utilize vulnerable code available for SV fixing to enable earlier SV assessment. Thus, in Chapter 4, we propose to use (vulnerable) code functions instead of SV reports for SV assessment. Using 1,782 vulnerable functions curated from 200 open-source projects, we particularly investigate the use of different code parts (i.e., (non-)vulnerable statements) in these functions for building effective function-level SV assessment models. To the best of our knowledge, we are the first to distill practices of performing data-driven SV assessment using source code. Such an approach can enable earlier fixing prioritization of SVs than the report-level SV assessment counterpart.

Chapter 5: Automated Commit-Level SV Assessment

Chapter 5 extends the work in Chapter 4 by addressing another practical scenario of code-based SV assessment. In real-world software development, developers have increasingly adopted DevOps for continuous integration, in which incremental changes are made to codebases via code commits to implement new features or fix bugs/SVs [24]. Meneely et al. [25] showed that such code commits/changes can introduce SVs. However, it is wastage of resources to use function-level SV assessment for these cases because functions are often not entirely changed/added in code commits [26]. Instead, performing SV assessment directly on these changes enables *just-in-time*, i.e., as soon as SVs are introduced, provision of SV characteristics for SV fixing. To the best of our knowledge, just-in-time SV assessment using code commits has never been explored. Therefore, in Chapter 5, we propose DeepCVA, the first model that automates commit-level SV assessment. This model leverages the multi-task DL paradigm [27] to automate various SV assessment tasks simultaneously in a unified model. We evaluate the effectiveness and efficiency of DeepCVA on 1,229 vulnerability-contributing commits in 246 open-source projects. DeepCVA is expected to increase the efficiency in model (re-)training and maintenance for continuous integration using DevOps in practice compared to conventional task-wise models.

Chapter 6: Collection and Analysis of Developers' SV Concerns on Question and Answer Websites

Chapter 6 tackles the third challenge of lacking considerations of developers' real-world SV concerns for SV assessment. Most of the current data-driven models, including the ones in Chapter 3, 4, and 5, have automated the SV assessment tasks that are based

on the expert-defined SV taxonomies/standards such as Common Weakness Enumeration (CWE) [28] or Common Vulnerability Scoring System (CVSS) [29]. While these taxonomies are designed to be as general as possible, they may not well represent the SV-related concerns that developers regularly have. For instance, developers often encounter only a small subset of SVs/SV types rather than all available ones; these SVs should be given a higher priority during SV assessment as they are of more use/interest to developers. Chapter 6 conducts the first empirical study on developers' real-world SV concerns using more than 70,000 SV-related posts curated from Question and Answer (Q&A) websites. We use the Latent Dirichlet Allocation topic modeling technique [30] to identify the commonly encountered issues. We then characterize these posts in terms of their popularity, difficulty, provided expertise, and available solutions. We also provide implications on leveraging such characteristics for making (data-driven) SV assessment more practical.

The key **contributions** of this thesis (the green boxes in Fig. 1.2) from the six aforementioned chapters are summarized as follows.

1. **A systematization of knowledge of data-driven SV assessment (Chapter 2)**: (*i*) A taxonomy of five SV assessment tasks. (*ii*) Detailed analysis of the pros and cons of frequent data sources, features, prediction models, evaluation techniques and evaluation metrics used for developing data-driven SV assessment models. (*iii*) Three key challenges limiting the practical application of data-driven SV assessment.

2. **Report-level assessment for ever-increasing SVs (Chapter 3)**. (*i*) Impact analysis of changing data (concept drift) of SV reports on the development and evaluation of SV assessment models. (*ii*) Concept-drift-aware models for automating report-level SV assessment.

3. **Early and just-in-time SV assessment using source code (Chapters 4 and 5)**. (*i*) Practices of developing effective data-driven models using vulnerable code statements and context in functions for early SV assessment without delays caused by missing SV reports. (*ii*) Just-in-time and efficient SV assessment using code commits (where SVs are first added) with deep multi-task learning.

4. **Insights of developers' real-world concerns that are useful for SV assessment (Chapter 6)**. (*i*) A taxonomy of 13 key SV concerns commonly encountered by developers in practice. (*ii*) Analysis of popularity, difficulty, expertise level, solutions provided for these real-world concerns. (*iii*) Implications of these concerns and their characteristics for practical data-driven SV assessment.

1.3 Related Publications

All of the core chapters and contributions of this thesis have been published during my PhD candidature. The list of directly related publications with respect to each chapter is given below.

❶ **Triet Huynh Minh Le**, Huaming Chen, and Muhammad Ali Babar, "A Survey on Data-Driven Software Vulnerability Assessment and Prioritization," ACM Computing Surveys (CSUR), 2021. [CORE ranking: **rank A***, Impact factor (2020): 10.282, SJR rating: Q1] (Chapter 2)

❷ **Triet Huynh Minh Le** and Muhammad Ali Babar, "On the Use of Fine-Grained Vulnerable Code Statements for Software Vulnerability Assessment Models", in Proceedings

of the 19th International Conference on Mining Software Repositories (MSR). ACM, 2022. [CORE ranking: **rank A**, Acceptance rate: 34%] (Chapter 3)

❸ **Triet Huynh Minh Le**, Bushra Sabir, and Muhammad Ali Babar, "Automated Software Vulnerability Assessment with Concept Drift," in Proceedings of the 16th International Conference on Mining Software Repositories (MSR). IEEE, 2019, pp. 371–382. [CORE ranking: **rank A**, Acceptance rate: 25%] (Chapter 4)

❹ **Triet Huynh Minh Le**, David Hin, Roland Croft, and Muhammad Ali Babar, "DeepCVA: Automated Commit-Level Vulnerability Assessment with Deep Multi-Task Learning," in Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2021, pp. 717–729. [CORE ranking: **rank A***, Acceptance rate: 19%] (Chapter 5)

❺ **Triet Huynh Minh Le**, David Hin, Roland Croft, and Muhammad Ali Babar, "PUMiner: Mining Security Posts from Developer Question and Answer Websites with PU Learning," in Proceedings of the 17th International Conference on Mining Software Repositories (MSR). ACM, 2020, pp. 350–361. [CORE ranking: **rank A**, Acceptance rate: 25.7%] (Chapter 6)

❻ **Triet Huynh Minh Le**, Roland Croft, David Hin, and Muhammad Ali Babar, "A Large-Scale Study of Security Vulnerability Support on Developer Q&A Websites," in Proceedings of the 25th Evaluation and Assessment in Software Engineering (EASE). ACM, 2021, pp. 109–118. [CORE ranking: **rank A**, Acceptance rate: 27%, **Nominated for the Best Paper Award**] (Chapter 6)

In addition to the six aforementioned publications, I contributed as the first author or coauthor of the following two publications during my PhD candidature, which are not directly related to the materials in this thesis.

❼ **Triet Huynh Minh Le**, Hao Chen, and Muhammad Ali Babar, "Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges," ACM Computing Surveys (CSUR), vol. 53, no. 3, pp. 1–38, 2020. [CORE ranking: **rank A***, Impact factor (2020): 10.282, SJR rating: Q1, **High-impact research work** selected by Faculty of Engineering, Computer & Mathematical Sciences at the University of Adelaide.]

❽ Xuanyu Duan, Mengmeng Ge, **Triet Huynh Minh Le**, Faheem Ullah, Shang Gao, Xuequan Lu, and Muhammad Ali Babar, "Automated Security Assessment for the Internet of Things," in 2021 IEEE 26th Pacific Rim International Symposium on Dependable Computing (PRDC). IEEE, 2021, pp. 47–56. [CORE ranking: **rank B**]

1.4 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 reports a literature review on the tasks, practices and challenges of data-driven SV assessment. Chapter 3 investigates the impacts of the concept drift issue on report-level SV assessment models and proposes an automated technique to address this issue. Chapter 4 explores the practices of building function-level SV assessment models. Chapter 5 describes DeepCVA, a novel multi-task DL model for automated commit-level SV assessment. Chapter 6 presents an empirical study of the common SV concerns encountered by developers on Q&A websites and distills respective implications for data-driven SV assessment. Finally, Chapter 7 summarizes the main contributions and findings of the thesis and suggests potential research avenues for future work in the area of data-driven SV assessment.

Chapter 2

Literature Review on Data-Driven Software Vulnerability Assessment

Related publication: This chapter is based on our paper titled "*A Survey on Data-Driven Software Vulnerability Assessment and Prioritization*", published in the ACM Computing Surveys journal (CORE A*) [11].

As mentioned in Chapter 1, Software Vulnerabilities (SVs) are increasing in complexity and scale, posing great security risks to many software systems. Given the limited resources in practice, SV assessment[1] help practitioners devise optimal SV mitigation plans based on various SV characteristics. The recent surges in SV data sources and data-driven techniques such as Machine Learning and Deep Learning have taken SV assessment to the next level. Chapter 2 provides a taxonomy of the key tasks performed by the past research efforts in the area. We also highlight the best practices, in terms of data sources, features, prediction models, evaluation techniques and evaluation metrics, for data-driven SV assessment. At the end of the review, we discuss some of the current and important challenges in the field that set the stage for the key contributions of this thesis in Chapters 3, 4, 5 and 6.

[1] In the original review paper [11], we used the term *SV assessment and prioritization* instead of *SV assessment*. However, in Chapter 2, the term *SV assessment* is used to ensure consistency with the other parts of the thesis, and it is important to note that *SV assessment* and *SV assessment and prioritization* can be used interchangeably as most SV assessment tasks can be used for prioritizing SV fixing.

2.1 Introduction

As discussed in Chapter 1, Software Vulnerability (SV) assessment is required to prioritize the remediation of critical SVs (i.e., the ones that can lead to devastating cyber-attacks) [10]. SV assessment includes tasks that determine various characteristics such as the types, exploitability, impact and severity levels of SVs [31]. Such characteristics help understand and select high-priority SVs to resolve early given the limited effort and resources. For example, SVs with simple exploitation and severe impacts likely require high fixing priority.

There has been an active research area to assess and prioritize SVs using increasingly large data from multiple sources. Many studies in this area have proposed different Natural Language Processing (NLP), Machine Learning (ML) and Deep Learning (DL) techniques to leverage such data to automate various tasks such as predicting the Common Vulnerability Scoring System [29] (CVSS) metrics (e.g., [23, 22, 21]) or public exploits (e.g., [32, 33, 34]). These prediction models can learn the patterns automatically from vast SV data, which would be otherwise impossible to do manually. Such patterns are utilized to speed up the assessment processes of ever-increasing and more complex SVs, significantly reducing practitioners' effort. Despite the rising research interest in data-driven SV assessment, to the best of our knowledge, there has been no comprehensive review on the state-of-the-art methods and existing challenges in this area. To bridge this gap, we are the first to review in-depth the research studies that automate *data-driven SV assessment* tasks leveraging SV data and NLP/ML/DL techniques.

The **contributions** of our review are summarized as follows:

1. We categorize and describe the key tasks of data-driven SV assessment performed in relevant primary studies.
2. We synthesize and discuss the pros and cons of data, features, models, evaluation methods and metrics commonly used in the reviewed studies.
3. We highlight some key challenges with the current practices.

We believe that our findings can provide useful guidelines for researchers and practitioners to effectively utilize data to perform SV assessment.

Related Work. There have been several existing surveys/reviews on SV analysis and prediction, but they are fundamentally different from ours (see Table 2.1). Ghaffarian et al. [3] conducted a seminal survey on ML-based SV analysis and discovery. Subsequently, several studies [35, 36, 37, 38] reviewed DL techniques for detecting vulnerable code. However, these prior reviews did not describe how ML/DL techniques can be used to assess and prioritize the detected SVs. There have been other relevant reviews on using Open Source Intelligence (OSINT) (e.g., phishing or malicious emails/URLs/IPs) to make informed security decisions [39, 40, 41]. However, these OSINT reviews did not explicitly discuss the use of SV data and how such data can be leveraged to automate the assessment processes. Moreover, most of the reviews on SV assessment have focused on either static analysis tools [13] or rule-based approaches (e.g., expert systems or ontologies) [9]. These methods rely on pre-defined patterns and struggle to work with new types and different data sources of SVs compared to contemporary ML or DL approaches presented in this chapter [14, 15, 42]. Recently, Dissanayake et al. [43] reviewed the socio-technical challenges and solutions for security patch management that involves SV assessment after SV patches are identified. Unlike [43], we focus on the challenges, solutions and practices of automating various SV assessment tasks with data-driven techniques. We also consider all types of SV assessment regardless of the patch availability.

TABLE 2.1: Comparison of contributions between our review and the existing related surveys/reviews.

Study \ Contribution	Focus on SV assessment	Analysis of SV data sources	Analysis of data-driven approaches
Ghaffarian et al. 2017 [3]	–	–	✓(Mostly ML)
Lin et al. 2020 [38] Semasaba et al. 2020 [37] Singh et al. 2020 [36] Zeng et al. 2020 [35]	–	–	✓(Mostly DL)
Pastor et al. 2020 [39]	–	✓(OSINT)	–
Sun et al. 2018 [41] Evangelista et al. 2020 [40]	–	✓(OSINT)	✓
Khan et al. 2018 [9]	✓(Rule-based methods)	–	–
Kritikos et al. 2019 [13]	✓(Static analysis)	✓	–
Dissanayake et al. 2020 [43]	✓(Socio-technical aspects)	–	–
Our review	✓	✓	✓

Chapter Organization. The remainder of the chapter is organized as follows. Section 2.2 presents the scope, methodology and taxonomy covered in this chapter. Sections 2.3, 2.4, 2.5, 2.6 and 2.7 review the studies in each theme of the taxonomy and discuss the limitations/gaps and open opportunities at the end of each theme. Section 2.8 identifies and discusses the common practices and respective implications for data-driven SV assessment. Finally, section 2.9 concludes the review and discusses the open challenges of this research area.

2.2 Overview of the Literature Review

2.2.1 Scope

This review's focus is on *data-driven SV assessment*. Unlike the existing surveys on rule-based or experience-based SV assessment [13, 9, 43] that hardly utilize the potential of SV data in the wild, this chapter aims to review research papers that have leveraged such data to automate tasks in this area using data-driven models. To keep our focus, we do not consider papers that only perform manual analyses or descriptive statistics (e.g., taking mean/median/variation of data) without using any data-driven models as these techniques cannot automatically assess or prioritize new SVs. We also do not directly compare the absolute performance of all the related studies as they did not use exactly the same experimental setup (e.g., data sources and model configurations). While it is theoretically possible to perform a comparative evaluation of the identified techniques by establishing and using a common setup, this type of evaluation is out of the scope of this chapter. However, we still cover the key directions/techniques of the studies in sections 2.3, 2.4, 2.5, 2.6 and 2.7. We also provide in-depth discussion on the common practices of these studies in section 2.8 and identify some current challenges with the field in section 2.9.

2.2.2 Methodology

Study selection. Our study selection was inspired by the Systematic Literature Review guidelines [44]. We first designed the search string: "*'software' AND vulner* AND (learn* OR data* OR predict*) AND (priority* OR assess* OR impact* OR exploit* OR severity*) AND NOT (fuzz* OR dynamic* OR intrusion OR adversari* OR malware* OR 'vulnerability detection' OR 'vulnerability discovery' OR 'vulnerability identification' OR 'vulnerability prediction')*". This search string covered the key papers (i.e., with more than

TABLE 2.2: Inclusion and exclusion criteria for study selection. **Notes**: We did not limit the selection to only peer-reviewed papers as this is an emerging field with many (high-quality) papers on Arxiv and most of them are under-submission. However, to ensure the quality of the papers on Arxiv, we only selected the ones with at least one citation.

Inclusion criteria
• *I1.* Studies that focused on SVs rather than hardware or human vulnerabilities
• *I2.* Studies that focused on assessment task(s) of SVs
• *I3.* Studies that used data-driven approaches (e.g., ML/DL/NLP techniques) for SV assessment
Exclusion criteria
• *E1.* Studies that were not written in English
• *E2.* Studies that we could not retrieve their full texts
• *E3.* Studies that were not related to Computer Science
• *E4.* Studies that were literature review or survey
• *E5.* Studies that only performed statistical analysis of SV assessment metrics
• *E6.* Studies that only focused on (automated) collection of SV data

50 citations) in the area and excluded many papers on general security and SV detection. We then adapted this search string[2] to retrieve an initial list of 1,765 papers up to April 2021[3] from various commonly used databases such as IEEE Xplore, ACM Digital Library, Scopus, SpringerLink and Wiley. We also defined the inclusion/exclusion criteria (see Table 2.2) to filter out irrelevant/low-quality studies with respect to our scope in section 2.2.1. Based on these criteria and the titles and abstracts and keywords of 1,765 initial papers, we removed 1,550 papers. After reading the full-text and applying the criteria on the remaining 215 papers, we obtained 70 papers directly related to data-driven SV assessment. To further increase the coverage of studies, we performed backward and forward snowballing [45] on these 70 papers (using the above sources and Google Scholar) and identified 14 more papers that satisfied the inclusion/exclusion criteria. In total, we included 84 studies in our review. We do not claim that we have collected all the papers in this area, but we believe that our selection covered most of the key studies to unveil the practices of data-driven SV assessment.

Data extraction and synthesis of the selected studies. We followed the steps of thematic analysis [18] to identify the taxonomy of data-driven SV assessment tasks in sections 2.3, 2.4, 2.5, 2.6 and 2.7 as well as the key practices of data-driven model building for automating these tasks in section 2.8. We first conducted a pilot study of 20 papers to familiarize ourselves with data to be extracted from the primary studies. After that, we generated initial codes and then merged them iteratively in several rounds to create themes. Two of the authors performed the analysis independently, in which each author analyzed half of the selected papers and then reviewed the analysis output of the other author. Any disagreements were resolved through discussions.

2.2.3 Taxonomy of Data-Driven Software Vulnerability Assessment

Based on the scope in section 2.2.1 and the methodology in section 2.2.2, we identified five main themes of the relevant studies in the area of data-driven SV assessment (see

[2]This search string was customized for each database and the database-wise search strings can be found at https://figshare.com/s/da4d238ecdf9123dc0b8.

[3]Given the ever-growing nature of this field, we maintain an up-to-date list of papers and resources on data-driven SV assessment at https://github.com/lhmtriet/awesome-vulnerability-assessment. More details are also given in Appendix 2.10.

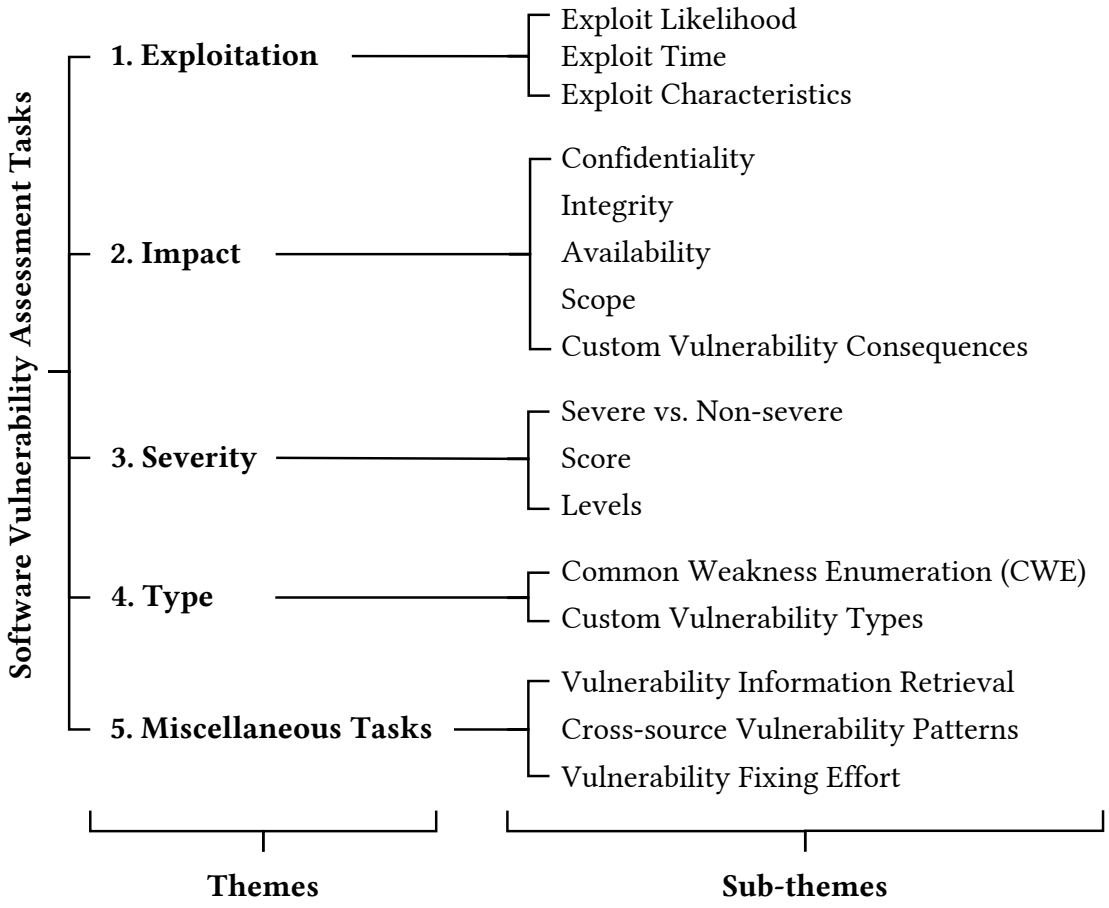


FIGURE 2.1: Taxonomy of studies on data-driven SV assessment.

Figure 2.1). Specifically, we extracted the themes by grouping related SV assessment tasks that the reviewed studies aim to automate/predict using data-driven models. Note that a paper is categorized into more than one theme if that paper develops models for multiple cross-theme tasks.

We acknowledge that there can be other ways to categorize the studies. However, we assert the reliability of our taxonomy as all of our themes (except theme 5) align with the security standards used in practice. For example, Common Vulnerability Scoring System (CVSS) [29] provides a framework to characterize exploitability, impact and severity of SVs (themes 1-3), while Common Weakness Enumeration (CWE) [28] includes many vulnerability types (theme 4). Hence, we believe our taxonomy can help identify and bridge the knowledge gap between the academic literature and industrial practices, making it relevant and potentially beneficial for both researchers and practitioners. Details of each theme in our taxonomy are covered in subsequent sections.

2.3 Exploitation Prediction

This section covers the *Exploitation* theme that automates the detection and understanding of both Proof-of-Concept (PoC) and real-world exploits[4] targeting identified SVs. This theme outputs the origin of SVs and how/when attackers would take advantage of such

[4] An *exploit* is a piece of code used to compromise vulnerable software [33]. Real-world exploits are harmful & used in real host/network-based attacks. PoC exploits are unharmful & used to show the potential threats of SVs in penetration tests.

TABLE 2.3: List of the reviewed papers in the *Exploit Likelihood* sub-theme of the *Exploitation* theme. **Note**: The nature of task of this sub-theme is binary classification of existence/possibility of proof-of-concept and/or real-world exploits.

Study	Data source	Data-driven technique
Bozorgi et al. 2010 [32]	CVE, Open Source Vulnerability Database (OSVDB)	Linear Support Vector Machine (SVM)
Sabottke et al. 2015 [33]	NVD, Twitter, OSVDB, ExploitDB, Symantec security advisories, private Microsoft security advisories	Linear SVM
Edkrantz et al. 2015 [46, 47]	NVD, Recorded Future security advisories, ExploitDB	Naïve Bayes, Linear SVM, Random forest
Bullough et al. 2017 [34]	NVD, Twitter, ExploitDB	Linear SVM
Almukaynizi et al. [48, 49]	NVD, ExploitDB, Zero Day Initiative security advisories & Darkweb forums/markets	SVM, Random forest, Naïve Bayes, Bayesian network, Decision tree, Logistic regression
Xiao et al. 2018 [50]	NVD, SecurityFocus security advisories, Symantec Spam/malicious activities based on daily blacklists from abuseat.org, spamhaus.org, spamcop.net, uceprotect.net, wpbl.info & list of unpatched SVs in hosts	Identification of malicious activity groups with community detection algorithms + Random forest for exploit prediction
Tavabi et al. 2018 [51]	NVD, 200 sites on Darkweb, ExploitDB, Symantec, Metasploit	Paragraph embedding + Radial basis function kernel SVM
de Sousa et al. 2020 [52]	NVD, Twitter, ExploitDB, Symantec Avast, ESET, Trend Micro security advisories	Linear SVM, Logistic regression, XGBoost, Light Gradient Boosting Machine (LGBM)
Fang et al. 2020 [53]	NVD, ExploitDB, SecurityFocus, Symantec	fastText + LGBM
Huang et al. 2020 [54]	NVD, CVE Details, Twitter, ExploitDB, Symantec security advisories	Random forest
Jacobs et al. 2020 [55]	NVD, Kenna Security Exploit sources: Exploit DB, Metasploit, FortiGuard Labs, SANS Internet Storm Center, Securewords CTU, Alienvault OSSIM, Canvas/D2 Security's Elliot Exploitation Frameworks, Contagio, Reversing Labs	XGBoost
Yin et al. 2020 [56]	NVD, ExploitDB, General text: Book Corpus & Wikipedia for pretraining BERT models	Fine-tuning BERT models pretrained on general text
Bhatt et al. 2021 [57]	NVD, ExploitDB	Features augmented by SV types + Decision tree, Random forest, Naïve Bayes, Logistic regression, SVM
Suciu et al. 2021 [58]	NVD, Vulners database, Twitter, Symantec, SecurityFocus, IBM X-Force Threat Intelligence Exploit sources: ExploitDB, Metasploit, Canvas, D2 Security's Elliot, Tenable, Skybox, AlienVault, Contagio	Multi-layer perceptron
Younis et al. 2014 [59]	Vulnerable functions from NVD (Apache HTTP Server project), ExploitDB, OSVDB	SVM
Yan et al. 2017 [60]	Executables (binary code) of 100 Linux applications	Combining ML (Decision tree) output & fuzzing with a Bayesian network
Tripathi et al. 2017 [61]	Program crashes from VDiscovery [62, 63] & LAVA [64] datasets	Static/Dynamic analysis features + Linear/Radial basis function kernel SVM
Zhang et al. 2018 [65]	Program crashes from VDiscovery [62, 63] dataset	n-grams of system calls from execution traces + Online passive-aggressive classifier

SVs to compromise a system of interest, assisting practitioners to quickly react to the more easily exploitable or already exploited SVs. The papers in this theme can be categorized into three groups/sub-themes: (*i*) Exploit likelihood, (*ii*) Exploit time, (*iii*) Exploit characteristics, as given in Tables 2.3, 2.4 and 2.5, respectively.

2.3.1 Summary of Primary Studies

2.3.1.1 Exploit Likelihood

The first sub-theme is *exploit likelihood* that predicts whether SVs would be exploited in the wild or PoC exploits would be released publicly (see Table 2.3). In 2010, Bozorgi et al. [32] were the first to use SV descriptions on Common Vulnerabilities and Exposures (CVE) [66] and Open Source Vulnerability Database (OSVDB)[5] to predict exploit existence based on the labels on OSVDB. In 2015, Sabottke et al. [33] conducted a seminal study that used Linear SVM and SV information on Twitter to predict PoC exploits on ExploitDB [67] as well as real-world exploits on OSVDB, Symantec's attack signatures [68] and private Microsoft's security advisories [69]. These authors urged to explicitly consider real-world exploits as *not* all PoC exploits would result in exploitation in practice. They also showed SV-related information on Twitter[6] can enable earlier detection of exploits than using expert-verified SV sources (e.g., NVD).

Built upon these two foundational studies [32, 33], the literature has mainly aimed to improve the performance and applicability of exploit prediction models by leveraging more exploit sources and/or better data-driven techniques/practices. Many researchers [47, 46, 48, 51, 49, 55] increased the amount of ground-truth exploits using extensive sources other than ExploitDB and Symantec in [32, 33]. The sources were security advisories such as Zero Day Initiative [70], Metasploit [71], SecurityFocus [72], Recorded Future [73], Kenna Security [74], Avast[7], ESET [75], Trend Micro [76], malicious activities in hosts based on traffic of spam/malicious IP addresses [50] and Darkweb sites/forums/markets [77]. In addition to enriching exploit sources, better data-driven models and practices for exploit prediction were also studied. Ensemble models (e.g., Random forest, eXtreme Gradient Boosting (XGBoost) [78], Light Gradient Boosting Machine (LGBM) [79]) were shown to outperform single-model baselines (e.g., Naïve Bayes, SVM, Logistic regression and Decision tree) for exploit prediction [53, 55, 52, 54]. Additionally, Bullough et al. [34] identified and addressed several issues with exploit prediction models, e.g., time sensitivity of SV data, already-exploited SVs before disclosure and training data imbalance, helping to improve the practical application of such models. Recently, Yin et al. [56] demonstrated that transfer learning is an alternative solution for improving the performance of exploit prediction with scarcely labeled exploits. Specifically, these authors pre-trained a DL model, BERT [80], on massive non-SV sources (e.g., text on Book Corpus [81] and Wikipedia [82]) and then fine-tuned this pre-trained model on SV data using additional pooling and dense layers. Bhatt et al. [57] also suggested that incorporating the types of SVs (e.g., SQL injection) into ML models can further enhance the predictive effectiveness. Suciu et al. [58] empirically showed that unifying SV-related sources used in prior work (e.g., SV databases [32], social media [33], SV-related discussions [51] and PoC code in ExploitDB [55]) supports more effective and timely prediction of *functional* exploits [83].

Besides using SV descriptions as input for exploit prediction, several studies in this sub-theme have also predicted exploits on the code level. Younis et al. [59] predicted the exploitability of vulnerable functions in the Apache HTTP Server project. Specifically,

[5] `http://osvdb.org`. Note that this database has been discontinued since 2016.

[6] `https://twitter.com`

[7] `https://avast.com/exploit-protection.php`. This link was provided by de Sousa et al. [52], but it is no longer available.

TABLE 2.4: List of the reviewed papers in the *Exploit Time* sub-theme of the *Exploitation* theme.

Study	Nature of task	Data source	Data-driven technique
Bozorgi et al. 2010 [32]	*Binary classification*: Likelihood that SVs would be exploited within 2 to 30 days after disclosure	CVE, OSVDB	Linear SVM
Edkrantz 2015 [46]	*Binary classification*: Likelihood of SV exploits within 12 months after disclosure	NVD, ExploitDB, Recorded Future security advisories	SVM, K-Nearest Neighbors (KNN), Naïve Bayes, Random forest
Jacobs et al. 2019 [90, 91]		NVD, Kenna Security Exploit sources: Exploit DB, Metasploit, D2 Security's Elliot & Canvas Exploitation Frameworks, Fortinet, Proofpoint, AlienVault & GreyNoise	Logistic regression
Chen et al. 2019 [92, 93]	*Binary classification*: Likelihood that SVs would be exploited within 1/3/6/9/12 months after disclosure *Regression*: number of days until SV exploits after disclosure	CVE, Twitter, ExploitDB, Symantec security advisories	Graph neural network embedding + Linear regression, Bayes, Random forest, XGBoost, Lasso/Ridge regression

these authors used an SVM model with features extracted from the dangerous system calls [84] in entry points/functions [85] and the reachability from any of these entry points to vulnerable functions [86]. Moving from high-level to binary code, Yan et al. [60] first used a Decision tree to obtain prior beliefs about SV types in 100 Linux applications using static features (e.g., *hexdump*) extracted from executables. Subsequently, they applied various fuzzing tools (i.e., Basic Fuzzing Framework [87] and OFuzz [88]) to detect SVs with the ML-predicted types. They finally updated the posterior beliefs about the exploitability based on the outputs of the ML model and fuzzers using a Bayesian network. The proposed method outperformed *!exploitable*,[8] a static crash analyzer provided by Microsoft. Tripathi et al. [61] also predicted SV exploitability from crashes (i.e., VDiscovery [62, 63] and LAVA [64] datasets) using an SVM model and static features from core dumps and dynamic features generated by the Last Branch Record hardware debugging utility. Zhang et al. [65] proposed two improvements to Tripathi et al. [61]'s approach. These authors first replaced the hardware utility in [61] that may not be available for resource-constrained devices (e.g., IoT) with sequence/n-grams of system calls extracted from execution traces. They also used an online passive-aggressive classifier [89] to enable online/incremental learning of exploitability for new crash batches on-the-fly.

2.3.1.2 Exploit Time

After predicting the likelihood of SV exploits in the previous sub-theme, this sub-theme provides more fine-grained information about *exploit time* (see Table 2.4). Besides performing binary classification of exploits, Bozorgi et al. [32] and Edkrantz [46] also predicted the time frame (2-30 days in [32] and 12 months in [46]) within which exploits would happen after the disclosure of SVs. Jacobs et al. [90, 91] then leveraged multiple sources containing both PoC and real-world exploits, as given in Table 2.4, to improve the number of labeled exploits, enhancing the prediction of exploit appearance within 12 months. Chen et al. [92] predicted whether SVs would be exploited within 1-12 months and the exploit time (number of days) after SV disclosure using Twitter data. The authors proposed a novel regression model whose feature embedding was a multi-layer graph neural network [94]

[8] https://microsoft.com/security/blog/2013/06/13/exploitable-crash-analyzer-version-1-6

capturing the content and relationships among tweets, respective tweets' authors and SVs. The proposed model outperformed many baselines and was integrated into the VEST system [93] to provide timely SV assessment information for practitioners. To the best of our knowledge, at the time of writing, Chen et al. [92, 93] have been the only ones pinpointing the exact exploit time of SVs rather than large/uncertain time-frames (e.g., months) in other studies, helping practitioners to devise much more fine-grained remediation plans.

2.3.1.3 Exploit Characteristics

Exploit characteristics is the final sub-theme that reveals various requirements/means of exploits (see Table 2.5), informing the potential scale of SVs; e.g., remote exploits likely affect more systems than local ones. The commonly used outputs are the Exploitability metrics provided by versions 2 [110] and 3 [111, 83] of Common Vulnerability Scoring System (CVSS).

Many studies have focused on predicting and analyzing version 2 of CVSS exploitability metrics (i.e., Access Vector, Access Complexity and Authentication). Yamamoto et al. [95] were the first one to leverage descriptions of SVs on NVD together with a supervised Latent Dirichlet Allocation topic model [112] to predict these CVSS metrics. Subsequently, Wen et al. [96] used Radial Basis Function (RBF)-kernel SVM and various SV databases/advisories other than NVD (e.g., SecurityFocus, OSVDB and IBM X-Force [113]) to predict the metrics. Le et al. [23][9] later showed that the prediction of CVSS metrics suffered from the *concept drift* issue; i.e., descriptions of new SVs may contain Out-of-Vocabulary terms for prediction models. They proposed to combine sub-word features with traditional Bag-of-Word (BoW) features to infer the semantics of novel terms/words from existing ones, helping assessment models be more robust against concept drift. Besides prediction, Toloudis et al. [97] used principal component analysis [114] and Spearman's ρ correlation coefficient to reveal the predictive contribution of each word in SV descriptions to each CVSS metric. However, this technique does not directly produce the value of each metric.

Recently, several studies have started to predict CVSS version 3 exploitability metrics including the new Privileges and User Interactions. Ognawala et al. [98] fed the features generated by a static analysis tool, Macke [99], to a Random forest model to predict these CVSS version 3 metrics for vulnerable software/components. Later, Chen et al. [93] found that many SVs were disclosed on Twitter before on NVD. Therefore, these authors developed a system built on top of a Graph Convolutional Network [115] capturing the content and relationships of related Twitter posts about SVs to enable more timely prediction of the CVSS version 3 metrics. Elbaz et al. [100] developed a linear regression model to predict the numerical output of each metric and then obtained the respective categorical value with the numerical value closest to the predicted value. For example, a predicted value of 0.8 for Attack Vector CVSS v3 is mapped to *Network* (0.85) [111]. To prepare a clean dataset to predict these CVSS metrics, Jiang et al. [101] replaced inconsistent CVSS values in various SV sources (i.e., NVD, ICS CERT and vendor websites) with the most frequent value.

Instead of building a separate model for each CVSS metric, there has been another family of approaches predicting these metrics using a single model to increase efficiency. Gawron et al. [102] and Spanos et al. [22] predicted multiple CVSS metrics as a unique string instead of individual values. The output of each metric is then extracted from the concatenated string. Later, Gong et al. [103] adopted the idea of a unified model from the DL perspective by using the multi-task learning paradigm [116] to predict CVSS metrics simultaneously. The model has a feature extraction module (based on a Bi-LSTM model with attention mechanism [117]) shared among all the CVSS metrics/tasks, yet

[9] This study is presented in Chapter 3.

TABLE 2.5: List of the reviewed papers in the *Exploit Characteristics* sub-theme of the *Exploitation* theme.

Study	Nature of task	Data source	Data-driven technique
Yamamoto et al. 2015 [95]	*Multi-class classification*: CVSS v2 (Access Vector & Access Complexity metrics) *Binary classification*: CVSS v2 (Authentication metric)	NVD	Supervised Latent Dirichlet Allocation (LDA)
Wen et al. 2015 [96]		NVD, OSVDB, SecurityFocus, IBM X-Force	Radial basis function kernel SVM
Le et al. 2019 [23]		NVD	Concept-drift-aware models with Naïve Bayes, KNN, Linear SVM, Random forest, XGBoost, LGBM
Toloudis et al. 2016 [97]	*Correlation analysis*: CVSS v2	NVD	Principal component analysis & Spearman correlation coefficient
Ognawala et al. 2018 [98]	*Multi-class classification*: CVSS v3 (Attack Vector, Attack Complexity & Privileges Required metrics) *Binary classification*: CVSS v3 (User Interaction metric)	NVD (buffer overflow SVs) & Source code of vulnerable software/-components	Combining static analysis tool (Macke [99]) & ML classifiers (Naïve Bayes & Random forest)
Chen et al. 2019 [93]		CVE, NVD, Twitter	Graph convolutional network
Elbaz et al. 2020 [100]	*Multi-class/Binary classification*: CVSS v2/v3	NVD	Mapping outputs of Linear regression to CVSS metrics with closest values
Jiang et al. 2020 [101]		NVD, ICS Cert, Vendor websites (Resolve inconsistencies with a majority vote)	Logistic regression
Gawron et al. 2017 [102]	*Multi-target classification*: CVSS v2	NVD	Naïve Bayes, Multilayer Perceptron (MLP)
Spanos et al. 2018 [22]		NVD	Random forest, boosting model, Decision tree
Gong et al. 2019 [103]	*Multi-task classification*: CVSS v2	NVD	Bi-LSTM with attention mechanism
Chen et al. 2010 [104]	*Multi-class classification*: Platform-specific vulnerability locations (Local, Remote, Local area network) & vulnerability causes (e.g., Access/Input/Origin validation error)	NVD, Secunia vulnerability database, SecurityFocus, IBM X-Force	Linear SVM
Ruohonen et al. 2017 [105]	*Binary classification*: Web-related exploits or not	ExploitDB	LDA + Random forest
Aksu et al. 2018 [106]	*Multi-class classification*: author-defined pre-/post-condition privileges (None, OS (Admin/User), App (Admin/User))	NVD	RBF network, Linear SVM, NEAT [107], MLP
Liu et al. 2019 [108]		NVD	Information gain + Convolutional neural network
Kanakogi et al. 2021 [109]	*Multi-class classification*: Common Attack Pattern Enumeration and Classification (CAPEC)	NVD, CAPEC	Doc2vec/tf-idf with cosine similarity

specific prediction head/layer for each metric/task. This model outperformed single-task counterparts while requiring much less time to (re-)train.

Although CVSS exploitability metrics were most commonly used, several studies used other schemes for characterizing exploitation. Chen et al. [104] used Linear SVM and SV descriptions to predict multiple SV characteristics, including three *SV locations* (i.e., Local, LAN and Remote) on SecurityFocus [72] and Secunia [118] databases as well as 11 *SV causes*[10] on SecurityFocus. Regarding the exploit types, Rouhonen et al. [105] used LDA [30] and Random forest to classify whether an exploit would affect a web application. This study can help find relevant exploits in components/sub-systems of a large system. For privileges, Aksu et al. [106] extended the Privileges Required metric of CVSS by incorporating the context (i.e., Operating system or Application) to which privileges are applied (see Table 2.5). They found MLP [119] to be the best-performing model for obtaining these privileges from SV descriptions. They also utilized the predicted privileges to generate attack graphs (sequence of attacks from source to sink nodes). Liu et al. [108] advanced this task by combining information gain for feature selection and Convolutional Neural Network (CNN) [120] for feature extraction. Regarding attack patterns, Kanakogi et al. [109] found Doc2vec [121] to be more effective than term-frequency inverse document frequency (tf-idf) when combined with cosine similarity to find the most relevant Common Attack Pattern Enumeration and Classification (CAPEC) [122] for a given SV on NVD. Such attack patterns can manifest how identified SVs can be exploited by adversaries, assisting the selection of suitable countermeasures.

2.3.2 Theme Discussion

In the *Exploitation* theme, the primary tasks are binary classification of whether Proof-of-Concept (PoC)/real-world exploits of SVs would appear and multi-classification of exploit characteristics based on CVSS. PoC exploits mostly come from ExploitDB [67]; whereas, real-world exploits, despite coming from multiple sources, are still much scarcer than PoC counterparts. Consequently, the models predicting real-world exploits have generally performed worse than those for PoC exploits. Similarly, the performance of the models determining CVSS v3 exploitability metrics has been mostly lower than that of the CVSS v2 based models. However, real exploits and CVSS v3 are usually of more interest to the community. The former can lead to real cyber-attacks and the latter is the current standard in practice. To improve the performance of these practical tasks, future work can consider adapting the patterns learned from PoC exploits and old CVSS versions to real exploits and newer CVSS versions, respectively, e.g., using transfer learning [123].

Besides the above tasks, there are other under-explored tasks targeting fine-grained prediction of exploits. In fact, mitigation of exploits in practice usually requires more information besides simply determining whether an SV would be exploited. Information gathered from predicting *when* and *how* the exploits would happen is also needed to devise better SV fixing prioritization and mitigation plans. VEST [93] is one of the first and few systems aiming to provide such all-in-one information about SV exploitation. However, this system currently only uses data from NVD/CVE and Twitter, which can be extended to incorporate more (exploit-related) sources and more sophisticated data-driven techniques in the future.

Most of the current studies have used SV descriptions on NVD and other security advisories to predict the exploitation-related metrics. This is surprising as SV descriptions do not contain root causes of SVs. Instead, SVs are rooted in source code, yet there is little work on code-based exploit prediction. So far, Younis et al. [59] have been the only

[10] Access/Input/Origin validation error, Atomicity/Configuration/Design/Environment/Serialization error, Boundary condition error, Failure on exceptions, Race condition error

TABLE 2.6: List of the reviewed papers in the *Impact* theme. **Note**: We grouped the first four sub-themes as they were mostly predicted together.

Study	Nature of task	Data source	Data-driven technique
Sub-themes: 1. Confidentiality, 2. Integrity, 3. Availability & 4. Scope (only in CVSS v3)			
Yamamoto et al. 2015 [95]	*Multi-class classification*: CVSS v2	NVD	Supervised Latent Dirichlet Allocation
Wen et al. 2015 [96]		NVD, OSVDB, SecurityFocus, IBM X-Force	Radial basis function kernel SVM
Le et al. 2019 [23]		NVD	Concept-drift-aware models with Naïve Bayes, KNN, Linear SVM, Random forest, XGBoost, LGBM
Toloudis et al. 2016 [97]	*Correlation analysis*: CVSS v2	NVD	Principal component analysis & Spearman correlation coefficient
Ognawala et al. 2018 [98]	*Multi-class classification*: CVSS v3 *Binary classification*: Scope in CVSS v3	NVD (buffer overflow SVs) & Source code of vulnerable software/-components	Combining static analysis tool (Macke [99]) & ML classifiers (Naïve Bayes & Random forest)
Chen et al. 2019 [93]		CVE, NVD, Twitter	Graph convolutional network
Elbaz et al. 2020 [100]	*Multi-class classification*: CVSS v2/v3 *Binary classification*: Scope in CVSS v3	NVD	Mapping outputs of Linear regression outputs to CVSS metrics with closest values
Jiang et al. 2020 [101]		NVD, ICS Cert, Vendor websites (Resolve inconsistencies with a majority vote)	Logistic regression
Gawron et al. 2017 [102]	*Multi-target classification*: CVSS v2	NVD	Naïve Bayes, MLP
Spanos et al. 2018 [22]		NVD	Random forest, boosting model, Decision tree
Gong et al. 2019 [103]	*Multi-task classification*: CVSS v2	NVD	Bi-LSTM with attention mechanism
Sub-theme: 5. Custom Vulnerability Consequences			
Chen et al. 2010 [104]	*Multi-label classification*: Platform-specific impacts (e.g., Gain system access)	NVD, Secunia vulnerability database, SecurityFocus, IBM X-Force	Linear SVM

ones using source code for exploit prediction, but their approach still requires manual identification of dangerous function calls in C/C++. More work is required to employ data-driven approaches to alleviate the need for manually defined rules to improve the effectiveness and generalizability of code-based exploit prediction.

2.4 Impact Prediction

This section describes the *Impact* theme that determines the (negative) effects that SVs have on a system of interest if such SVs are exploited. There are five key tasks that the papers in this theme have automated/predicted: (*i*) Confidentiality impact, (*ii*) Integrity impact, (*iii*) Availability impact, (*iv*) Scope and (*v*) Custom vulnerability consequences (see Table 2.6).

2.4.1 Summary of Primary Studies

2.4.1.1 Confidentiality, Integrity, Availability, and Scope

A majority of the papers have focused on the impact metrics provided by CVSS, including versions 2 [110] and 3 [111, 83]. Versions 2 and 3 share three impact metrics *Confidentiality*, *Integrity* and *Availability*. Version 3 also has a new metric, *Scope*, that specifies whether

an exploited SV would affect only the system that contains the SV. For example, *Scope* changes when an SV occurring in a virtual machine affects the whole host machine, in turn increasing individual impacts.

The studies that predicted the CVSS impact metrics are mostly the same as the ones predicting the CVSS exploitability metrics in section 2.3. Given the overlap, we hereby only describe the main directions and techniques of the *Impact*-related tasks rather than iterating the details of each study. Overall, a majority of the work has focused on classifying CVSS impact metrics (versions 2 and 3) using three main learning paradigms: single-task [95, 96, 23, 98, 93, 100, 101], multi-target [102, 22] and multi-task [103] learning. Instead of developing a separate prediction model for each metric like the single-task approach, multi-target and multi-task approaches only need a single model for all tasks. Multi-target learning predicts concatenated output; whereas, multi-task learning uses shared feature extraction for all tasks and task-specific softmax layers to determine the output of each task. These three learning paradigms were powered by applying and/or customizing a wide range of data-driven methods. The first method was to use single ML classifiers like supervised Latent Dirichlet Allocation [95], Principal component analysis [97], Naïve Bayes [23, 98, 102], Logistic regression [101], Kernel-based SVM [96], Linear SVM [23], KNN [23] and Decision tree [22]. Other studies employed ensemble models combining the strength of multiple single models such as Random forest [23, 98], boosting model [22] and XGBoost/LGBM [23]. Recently, more studies moved towards more sophisticated DL architectures such as MLP [102], attention-based (Bi-)LSTM [103] and graph neural network [93]. Ensemble and DL models usually beat the single ones, but there is a lack of direct comparisons between these two emerging model types.

2.4.1.2 Custom Vulnerability Consequences

To devise effective remediation strategies for a system of interest in practice, practitioners may want to know *custom vulnerability consequences* which are more interpretable than the levels of impact provided by CVSS. Chen et al. [104] curated a list of 11 vulnerability consequences[11] from X-Force [113] and Secunia [118] vulnerability databases. They then used a Linear SVM model to perform multi-label classification of these consequences for SVs, meaning that an SV can lead to more than one consequence. To the best of our knowledge, this is the only study that has pursued this research direction so far.

2.4.2 Theme Discussion

In the *Impact* theme, the common task is to predict the impact base metrics provided by CVSS versions 2 and 3. Similar to the Exploitation theme, the models for CVSS v3 still require more attention and effort from the community to reach the same performance level as the models for CVSS v2. These impact metrics are also usually predicted together with the exploitability metrics given their similar nature (multi-class classification) using either task-wise models or a unified (multi-target or multi-task) model. Multi-target and multi-task learning are promising as they can reduce the time for continuous (re)training and maintenance when deployed in production.

Besides CVSS impact metrics, other fine-grained SV consequences have also been explored [104], but there is still no widely accepted taxonomy for such consequences. Thus, these consequences have seen less adoption in practice than CVSS metrics, despite being potentially useful by providing more concrete information about what assets/components

[11] Gain system access, Bypass security, Configuration manipulation, Data/file manipulation, Denial of Service, Privilege escalation, Information leakage, Session hijacking, Cross-site scripting (XSS), Source spoofing, Brute-force proneness.

TABLE 2.7: List of the reviewed papers in the *Severe vs. Non-Severe* sub-theme of the *Severity* theme. **Note**: The nature of task here is binary classification of severe SVs with High/Critical CVSS v2/v3 severity levels.

Study	Data source (software project)	Data-driven technique
Kudjo et al. 2019 [124]	NVD (Mozilla Firefox, Google Chrome, Internet Explorer, Microsoft Edge, Sea Monkey, Linux Kernel, Windows 7, Windows 10, Mac OS, Chrome OS)	Term frequency & inverse gravity moment weighting + KNN, Decision tree, Random forest
Chen et al. 2020 [125]	NVD (Adobe Flash Player, Enterprise Linux, Linux Kernel, Foxit Reader, Safari, Windows 10, Microsoft Office, Oracle Business Suites, Chrome, QuickTime)	Term frequency & inverse gravity moment weighting + KNN, Decision tree, Naïve Bayes, SVM, Random forest
Kudjo et al. 2020 [126]	NVD (Google Chrome, Mozilla Firefox, Internet Explorer and Linux Kernel)	Find the best smallest training dataset using KNN, Logistic regression, MLP, Random forest
Malhotra et al. 2021 [127]	NVD (Apache Tomcat)	Chi-square/Information gain + bagging technique, Random forest, Naïve Bayes, SVM

in a system that an SV can compromise. We recommend that future work investigate SV-related issues that practitioners commonly encounter in practice to potentially create a systematic taxonomy of custom SV consequences.

2.5 Severity Prediction

This section discusses the work in the *Severity* theme. Severity is often a function/combination of Exploitation (section 2.3) and Impact (section 2.4). SVs with higher severity usually require more urgent remediation. There are three main prediction tasks in this theme: (*i*) Severe vs. Non-severe, (*ii*) Severity levels and (*iii*) Severity score, shown in Tables 2.7, 2.8 and 2.9, respectively.

Similar to the *Exploitation* and *Impact* themes, many studies in the *Severity* theme have used CVSS versions 2 and 3. According to both CVSS versions, the severity score shares the same range from 0 to 10, with an increment of 0.1. Based on the score, the existing studies have either defined a threshold to decide whether an SV is severe (requiring high attention), or predicted levels/groups of severity score that require a similar amount of attention or determined the raw score value.

2.5.1 Summary of Primary Studies

2.5.1.1 Severe vs. Non-Severe

The first group of studies have classified whether an SV is *severe or non-severe*, making it a binary classification problem (see Table 2.7). These studies have typically selected severe SVs as the ones with at least High severity level (i.e., CVSS severity score ≥ 7.0). Kudjo et al. [124] showed that using term frequency (BoW) with inverse gravity moment weighting [128] to extract features from SV descriptions can enhance the performance of ML models (i.e., KNN, Decision tree and Random forest) in predicting the severity of SVs. Later, Chen et al. [125] confirmed that this feature extraction method was also effective for more projects and classifiers (e.g., Naïve Bayes and SVM). Besides investigating feature extraction, Kudjo et al. [126] also highlighted the possibility of finding Bellwether, i.e., the smallest set of data that can be used to train an optimal prediction model, for classifying severity. Recently, Malhotra et al. [127] revisited this task by showing that Chi-square and information gain can be effective dimensionality reduction techniques for multiple classifiers, i.e., bagging technique, Random forest, Naïve Bayes and SVM.

TABLE 2.8: List of the reviewed papers in the *Severity Levels* sub-theme of the *Severity* theme.

Study	Nature of task	Data source	Data-driven technique
Spanos et al. 2017 [20]	*Multi-class classification*: NVD severity levels based on CVSS v2 & WIVSS (High, Medium, Low)	NVD	Decision tree, SVM, MLP
Wang et al. 2019 [129]	*Multi-class classification*: NVD severity levels based on CVSS v2(High, Medium, Low)	NVD (XSS attacks)	XGBoost, Logistic regression, SVM, Random forest
Le et al. 2019 [23]		NVD	Concept-drift-aware models with Naïve Bayes, KNN, Linear SVM, Random forest, XGBoost, LGBM
Liu et al. 2019 [130]		NVD, China National Vulnerability Database (XSS attacks)	Recurrent Convolutional Neural Network (RCNN), Convolutional Neural Network (CNN), Long-Short Term Memory (LSTM)
Sharma et al. 2020 [131]		CVE Details	CNN
Han et al. 2017 [21]	*Multi-class classification*: Atlassian categories of CVSS severity score (Critical, High, Medium, Low)	CVE Details	1-layer CNN, 2-layer CNN, CNN-LSTM, Linear SVM
Sahin et al. 2019 [132]		NVD	1-layer CNN, LSTM, XGBoost, Linear SVM
Nakagawa et al. 2019 [133]		CVE Details	Character-level CNN vs. Word-based CNN + Linear SVM
Gong et al. 2019 [103]	*Multi-task classification*: Atlassian categories of CVSS severity score (Critical, High, Medium, Low)	CVE Details	Bi-LSTM with attention mechanism
Chen et al. 2010 [104]	*Multi-class classification*: severity levels of Secunia (Extremely/highly/moderately/less/non- critical)	CVE, Secunia vulnerability database, SecurityFocus, IBM X-Force	Linear SVM
Zhang et al. 2020 [134]	*Multi-class classification*: Platform-specific levels (High-/Medium/Low)	China National Vulnerability Database	Logistic regression, Linear discriminant analysis, KNN, CART, SVM, bagging/boosting models
Khazaei et al. 2016 [135]	*Multi-class classification*: 10 severity score bins (one unit/bin)	CVE & OSVDB	Linear SVM, Random forest, Fuzzy system

2.5.1.2 Severity Levels

Rather than just performing binary classification of whether an SV is severe, several studies have identified one among multiple *severity levels* that an SV belongs to (see Table 2.8). This setting can be considered as multi-class classification. Spanos et al. [20] were to first one to show the applicability of ML to classify SVs into one of the three severity levels using SV descriptions. These three levels are provided by NVD and based on the severity score of CVSS version 2 [110] and WIVSS [136], i.e., Low (0.0 – 3.9), Medium (4.0 – 6.9), High (7.0 – 10.0). Note that WIVSS assigns different weights for the Confidentiality, Integrity and Availability impact metrics of CVSS, enhancing the ability to capture varied contributions of these impacts to the final severity score. Later, Wang et al. [129] showed that XGBoost [78] performed the best among the investigated ML classifiers for predicting these three NVD-based severity levels. Le et al. [23] also confirmed that ensemble methods (e.g., XGBoost [78], LGBM [79] and Random forest) outperformed single models (e.g., Naïve Bayes, KNN and SVM) for this task. Predicting severity levels has also been tackled with DL techniques [130, 131] such as Recurrent Convolutional Neural Network (RCNN) [137], Convolutional Neural Network (CNN) [120], Long-Short Term Memory (LSTM) [138]. These studies showed potential performance gain of DL models compared to traditional ML counterparts. Han et al. [21] showed that DL techniques (i.e.,

TABLE 2.9: List of the reviewed papers in the *Severity Score* sub-theme of the *Severity* theme. **Notes**: †denotes that the severity score is computed from ML-predicted base metrics using the formula provided by an assessment framework (CVSS and/or WIVSS).

Study	Nature of task	Data source	Data-driven technique
Sahin et al. 2019 [132]	*Regression*: CVSS v2 (0-10)	NVD	1-layer CNN, LSTM, XGBoost regressor, Linear regression
Wen et al. 2015 [96]		OSVDB, SecurityFocus, IBM X-Force	Radial basis function kernel SVM†
Ognawala et al. 2018 [98]	*Regression*: CVSS v3 (0-10)	NVD (buffer overflow SVs)	Combining a static analysis tool (Macke [99]) & ML classifiers (Naïve Bayes & Random forest)†
Chen et al. 2019 [93, 140]		CVE, NVD, Twitter	Graph convolutional network
Anwar et al. 2020 [141, 142]		NVD	Linear regression, Support vector regression, CNN, MLP
Elbaz et al. 2020 [100]	*Regression*: CVSS v2/v3 (0-10)	NVD	Mapping outputs of Linear regression to CVSS metrics with closest values†
Jiang et al. 2020 [101]		NVD, ICS Cert, Vendor websites (Resolve inconsistencies with a majority vote)	Logistic regression†
Spanos et al. 2018 [22]	*Regression*: CVSS v2 & WIVSS (0-10)	NVD	Random forest, boosting model, Decision tree†
Toloudis et al. 2016 [97]	*Correlation analysis*: CVSS v2 & WIVSS (0-10)	NVD	Principal component analysis & Spearman correlation coefficient

1-layer CNN) also achieved promising results for predicting a different severity categorization, namely Atlassian's levels.[12] Such findings were successfully replicated by Sahin et al. [132]. Nakagawa et al. [133] further enhanced the DL model performance for the same task by incorporating the character-level features into a CNN model [139]. Complementary to performance enhancement, Gong et al. [103] proposed to predict these severity levels concurrently with other CVSS metrics in a single model using multi-task learning [116] powered by an attention-based Bi-LSTM shared feature extraction model. The unified model was demonstrated to increase both the prediction effectiveness and efficiency. Besides Atlassian's categories, several studies applied ML models to predict severity levels on other platforms such as Secunia [104] and China National Vulnerability Database[13] [134]. Instead of using textual categories, Khazaei et al. [135] divided the CVSS severity score into 10 bins with 10 increments each (e.g., values of 0 – 0.9 are in one bin) and obtained decent results (86-88% Accuracy) using Linear SVM, Random forest and Fuzzy system.

2.5.1.3 Severity Score

To provide even more fine-grained severity value than the categories, the last sub-theme has predicted the *severity score* (see Table 2.9). Using SV descriptions on NVD, Sahin et al. [132] compared the performance of ML-based regressors (e.g., XGBoost [78] and Linear regression) and DL-based ones (e.g., CNN [120] and LSTM [138]) for predicting the severity score of CVSS version 2 [110]. These authors showed that DL-based approaches generally outperformed the ML-based counterparts. For CVSS version 3 [111, 83], Chen et al. [93, 140] and Anwar et al. [141, 142] also reported the strong performance of DL-based models (e.g., CNN and graph convolutional neural network [115]). Some other studies

[12] https://www.atlassian.com/trust/security/security-severity-levels
[13] https://www.cnvd.org.cn

did not directly predict severity score from SV descriptions, instead they aggregated the predicted values of the CVSS Exploitability (see section 2.3) and Impact metrics (see section 2.4) using the formulas of CVSS version 2 [96, 22, 100, 101], version 3 [98, 100, 101] and WIVSS [22]. We noticed the papers predicting both versions (e.g., CVSS versions 2 vs. 3 or CVSS version 2 vs. WIVSS) usually obtained better performance for version 3 and WIVSS than version 2 [100, 101]. These findings may suggest that the improvements made by experts in version 3 and WIVSS compared to version 2 help make the patterns in severity score clearer and easier for ML models to capture. In addition to predicting severity score, Toloudis et al. [97] examined the correlation between words in descriptions of SVs and the severity values of such SVs, aiming to shed light on words that increase or decrease the severity score of SVs.

2.5.2 Theme Discussion

In the *Severity* theme, predicting the severity levels is the most prevalent task, followed by severity score prediction and then binary classification of the severity. In practice, severity score gives more fine-grained information (fewer SVs per value) for practitioners to rank/prioritize SVs than categorical/binary levels. However, predicting continuous score values is usually challenging and requires more robust models as this task involves higher uncertainty to learn inherent patterns from data than classifying fixed/discrete levels. We observed that DL models such as graph neural networks [93, 140], LSTM [132] and CNN [141, 142] have been shown to be better than traditional ML models for predicting severity score. However, most of these studies did not evaluate their models in a continuous deployment setting to investigate how the models will cope with changing patterns of new SVs over time. This issue particularly manifests and requires remediation in the context of report-level SV assessment (see section 2.9) where SV descriptions, the main input for SV assessment models, contain changing/new terms to describe ever-increasing SVs.

2.6 Type Prediction

This section reports the work done in the *Type* theme. Type groups SVs with similar characteristics, e.g., causes, attack patterns and impacts, and thus facilitating the reuse of known prioritization and remediation strategies employed for prior SVs of the same types. Two key prediction outputs are: (*i*) Common Weakness Enumeration (CWE) and (*ii*) Custom vulnerability types (see Table 2.10).

2.6.1 Summary of Primary Studies

2.6.1.1 Common Weakness Enumeration (CWE)

The first sub-theme determines and analyzes the patterns of the SV types provided by *CWE* [28]. CWE is currently the standard for SV types with more than 900 entries. The first group of studies has focused on multi-class classification of these CWEs. Wang et al. [143] were the first to tackle this problem with a Naïve Bayes model using the CVSS metrics (version 2) [110] and product names. Later, Shuai et al. [144] used LDA [30] with a location-aware weighting to extract important features from SV descriptions for building an effective SVM-based CWE classifier. Na et al. [145] also showed that features extracted from SV descriptions can improve the Naïve Bayes model in [143]. Ruohonen et al. [146] studied an information retrieval method, i.e., term-frequency inverse document frequency (tf-idf) and cosine similarity, to detect the CWE-ID with a description most similar to that of a given SV collected from NVD and Snyk.[14] This method performed well for CWEs

[14] https://snyk.io/vuln

TABLE 2.10: List of the reviewed papers in the *Type* theme.

Study	Nature of task	Data source	Data-driven technique
Sub-theme: 1. Common Weakness Enumeration (CWE)			
Wang et al. 2010 [143]	*Multi-class classification*: CWE classes	NVD, CVSS	Naïve Bayes
Shuai et al. 2013 [144]		NVD	SVM
Na et al. 2016 [145]		NVD	Naïve Bayes
Ruohonen et al. 2018 [146]		NVD, CWE, Snyk	tf-idf with 1/2/3-grams and cosine similarity
Huang et al. 2019 [147]		NVD, CWE	MLP, Linear SVM, Naïve Bayes, KNN
Aota et al. 2020 [148]		NVD	Random forest, Linear SVM, Logistic regression, Decision tree, Extremely randomized trees, LGBM
Aghaei et al. 2020 [149]		NVD, CVE	Adaptive fully-connected neural network with one hidden layer
Das et al. 2021 [150]		NVD, CWE	BERT, Deep Siamese network
Zou et al. 2019 [151]		NVD & Software Assurance Reference Dataset (SARD)	Three Bi-LSTM models for extracting and combining global and local features from code functions
Murtaza et al. 2016 [152]	*Unsupervised learning*: sequence mining of SV types (over time)	NVD (CWE & CPE)	2/3/4/5-grams of CWEs
Lin et al. 2017 [153]	*Unsupervised learning*: association rule mining of CWE-related aspects (prog. language, time of introduction & consequence scope)	CWE	FP-growth association rule mining algorithm
Han et al. 2018 [154]	*Binary/Multi-class classification*: CWE relationships (CWE links, link types & CWE consequences)	CWE	Deep knowledge graph embedding of CWE entities
Sub-theme: 2. Custom Vulnerability Types			
Venter et al. 2008 [155]	*Unsupervised learning*: clustering	CVE	Self-organizing map
Neuhaus et al. 2010 [156]	*Unsupervised learning*: topic modeling	CVE	Latent Dirichlet Allocation (LDA)
Mounika et al. [157, 158]		CVE, Open Web Application Security Project (OWASP)	LDA
Aljedaani et al. 2020 [159]		SV reports (Chromium project)	LDA
Williams et al. [160, 161]	*Multi-class classification*: manually coded SV types	NVD	Supervised Topical Evolution Model & Diffusion-based storytelling technique
Russo et al. 2019 [162]		NVD	Bayesian network, J48 tree, Logistic regression, Naïve Bayes, Random forest
Yan et al. 2017 [60]		Executables of 100 Linux applications	Decision tree
Zhang et al. 2020 [134]	*Multi-class classification*: platform-specific vulnerability types	China National Vulnerability Database	Logistic regression, Linear discriminant analysis, KNN, CART, SVM, bagging/boosting models

without clear patterns/keywords in SV descriptions. Aota et al. [148] utilized the Boruta feature selection algorithm [163] and Random forest to improve the performance of *base* CWE classification. Base CWEs give more fine-grained information for SV remediation than categorical CWEs used in [145].

There has been a recent rise in using neural network/DL based models for CWE classification. Huang et al. [147] implemented a deep neural network with tf-idf and information gain for the task and obtained better performance than SVM, Naïve Bayes and KNN. Aghaei et al. [149] improved upon [148] for both categorical (coarse-grained) and base (fine-grained) CWE classification with an adaptive hierarchical neural network to determine sequences of less to more fine-grained CWEs. To capture the hierarchical structure and rare classes of CWEs, Das et al. [150] matched SV and CWE descriptions instead of predicting CWEs directly. They presented a deep Siamese network with a BERT-based [80] shared feature extractor that outperformed many baselines even for rare/unseen CWE classes. Recently, Zou et al. [151] pioneered the multi-class classification of CWE in vulnerable functions curated from Software Assurance Reference Dataset (SARD) [164] and NVD. They achieved high performance (∼95% F1-Score) with DL (Bi-LSTM) models. The strength of their model came from combining global (semantically related statements) and local (variables/statements affecting function calls) features. Note that this model currently only works for functions in C/C++ and 40 selected classes of CWE.

Another group of studies has considered unsupervised learning methods to extract CWE sequences, patterns and relationships. Sequences of SV types over time were identified by Murtaza et al. [152] using an n-gram model. This model sheds light on both co-occurring and upcoming CWEs (grams), raising awareness of potential cascading attacks. Lin et al. [153] applied an association rule mining algorithm, FP-growth [165], to extract the rules/patterns of various CWEs aspects including types, programming language, time of introduction and consequence scope. For example, buffer overflow (CWE type) usually appears during the implementation phase (time of introduction) in C/C++ (programming language) and affects the availability (consequence scope). Lately, Han et al. [154] developed a deep knowledge graph embedding technique to mine the relationships among CWE types, assisting in finding relevant SV types with similar properties.

2.6.1.2 Custom Vulnerability Types

The second sub-theme is about *custom vulnerability types* other than CWE. Venter et al. [155] used Self-organizing map [166], an unsupervised clustering algorithm, to group SVs with similar descriptions on CVE. This was one of the earliest studies that automated SV type classification. Topic modeling is another popular unsupervised learning model [156, 157, 158, 159] to categorize SVs without an existing taxonomy. Neuhaus et al. [156] applied LDA [30] on SV descriptions to identify 28 prevalent SV types and then analyzed the trends of such types over time. The identified SV topics/types had considerable overlaps (up to 98% precision and 95% recall) with CWEs. Mounika et al. [157, 158] extended [156] to map the LDA topics with the top-10 OWASP [167]. However, the LDA topics/keywords did not agree well (< 40%) with the OWASP descriptions, probably because 10 topics did not cover all the underlying patterns of SV descriptions. Aljedaani et al. [159] again used LDA to identify 10 types of SVs reported in the bug tracking system of Chromium[15] and found memory-related issues were the most prevalent topics.

Another group of studies has classified manually defined/selected SV types rather than CWE as some SV types are encountered more often in practice and require more attention. Williams et al. [160, 161] applied a supervised topical evolution model [168] to identify the

[15] https://bugs.chromium.org/p/chromium/issues/list

features that best described the 10 pre-defined SV types[16] prevalent in the wild. These authors then used a diffusion-based storytelling technique [169] to show the evolution of a particular topic of SVs over time; e.g., increasing API-related SVs requires hardening the APIs used in a product. To support user-friendly SV assessment using ever-increasing unstructured SV data, Russo et al. [162] used Bayesian network to predict 10 pre-defined SV types.[17] Besides predicting manually defined SV types using SV natural language descriptions, Yan et al. [60] used a decision tree to predict 22 SV types prevalent in the executables of Linux applications. The predicted type was then combined with fuzzers' outputs to predict SV exploitability (see section 2.3.1.1). Besides author-defined types, custom SV types also come from specific SV platforms. Zhang et al. [134] designed an ML-based framework to predict the SV types collected from China National Vulnerability Database. Ensemble models (bagging and boosting models) achieved, on average, the highest performance for this task.

2.6.2 Theme Discussion

In the *Type* theme, detecting and characterizing coarse-grained and fine-grained CWE-based SV types are the frequent tasks. The large number and hierarchical structure of classes are the main challenges with CWE classification/analysis. In terms of solutions, deep Siamese networks [150] are more robust to the class imbalance issue (due to many CWE classes), while graph-based neural networks [154] can effectively capture the hierarchical structure of CWEs. Future work can investigate the combination of these two types of DL architectures to solve both issues simultaneously. Besides model-level solutions, author-selected or platform-specific SV types have been considered to reduce the complexity of CWE. However, similar to custom SV consequences in section 2.4.1.2, there is not yet a universally accepted taxonomy for these custom SV types. To reduce the subjectivity in selecting SV types for prediction, we suggest that future work should focus on the types that are commonly encountered and discussed by developers in the wild.

2.7 Miscellaneous Tasks

The last theme is *Miscellaneous Tasks* covering the studies that are representative yet do not fit into the four previous themes. This theme has three main sub-themes/tasks: (*i*) Vulnerability information retrieval, (*ii*) Cross-source vulnerability patterns and (*iii*) Vulnerability fixing effort (see Table 2.11).

2.7.1 Summary of Primary Studies

2.7.1.1 Vulnerability Information Retrieval

The first and major sub-theme is *vulnerability information retrieval* that studies data-driven methods to extract different SV-related entities (e.g., affected products/versions) and their relationships from SV data. The current sub-theme extracts assessment information appearing explicitly in SV data (e.g., SV descriptions on NVD) rather than predicting implicit properties as done in prior sub-themes. For instance, CWE-119, i.e., "Improper Restriction of Write Operations within the Bounds of a Memory Buffer", can be retrieved

[16] 1. Buffer errors, 2. Cross-site scripting, 3. Path traversal, 4. Permissions and Privileges, 5. Input validation, 6. SQL injection, 7. Information disclosure, 8. Resources Error, 9. Cryptographic issues, 10. Code injection.

[17] 1. Authentication bypass or Improper Authorization, 2. Cross-site scripting or HTML injection, 3. Denial of service, 4. Directory Traversal, 5. Local/Remote file include and Arbitrary file upload, 6. Information disclosure and/or Arbitrary file read, 7. Buffer/stack/heap/integer overflow, 8. Remote code execution, 9. SQL injection, 10. Unspecified vulnerability

TABLE 2.11: List of the reviewed papers in the *Miscellaneous Tasks* theme.

Study	Nature of task	Data source	Data-driven technique
Sub-theme: 1. Vulnerability Information Retrieval			
Weerawardhana et al. 2014 [170]	*Multi-class classification*: Extraction of entities (software name/version, impact, attacker/user actions) from SV descriptions	NVD (210 randomly selected and manually labeled SVs)	Stanford Named Entity Recognizer implementing a CRF classifier
Dong et al. 2019 [171]	*Multi-class classification*: Vulnerable software names/versions	CVE Details, NVD, ExploitDB, SecurityFocus, SecurityFocus Forum, SecurityTracker, Openwall	Word-level and character-level Bi-LSTM with attention mechanism
Gonzalez et al. 2019 [172]	*Multi-class classification*: Extraction of 19 Vulnerability Description Ontology [173] classes from SV descriptions	NVD	Naïve Bayes, Decision tree, SVM, Random forest, Majority voting model
Binyamini et al. 2020 [174, 175]	Multi-class classification: Extraction of entities (attack vector/means/technique, privilege, impact, vulnerable platform/version/OS, network protocol/port) from SV descriptions to generate MulVal [176] interaction rules	NVD	Bi-LSTM with various feature extractors: word2vec, ELMo, BERT (pre-trained or trained from scratch)
Guo et al. 2020 [177, 178]	*Multi-class classification*: Extraction of entities (SV type, root cause, attack type, attack vector) from SV descriptions	NVD, SecurityFocus	CNN, Bi-LSTM (with or without attention mechanism)
Waareus et al. 2020 [179]	*Multi-class classification*: Common Product Enumeration (CPE)	NVD	Word-level and character-level Bi-LSTM
Yitagesu et al. 2021 [180]	*Multi-class classification*: Part-of-speech tagging of SV descriptions	NVD, CVE, CWE, CAPEC, CPE, Twitter, PTB corpus [181]	Bi-LSTM
Sun et al. 2021 [182]	*Multi-class classification*: Extraction of entities (vulnerable product/version/component, type, attack type, root cause, attack vector, impact) from ExploitDB to generate SV descriptions	NVD, ExploitDB	BERT models
Sub-theme: 2. Cross-source Vulnerability Patterns			
Horawalavithana et al. 2019 [183]	*Regression*: Number of software development activities on GitHub after disclosure of SVs	Twitter, Reddit, GitHub	MLP, LSTM
Xiao et al. 2019 [184]	*Knowledge-graph reasoning*: modeling the relationships among SVs, its types and attack patterns	CVE, CWE, CAPEC (Linux project)	Translation-based knowledge-graph embedding
Sub-theme: 3. Vulnerability Fixing Effort			
Othmane et al. 2017 [185]	*Regression*: time (days) to fix SVs	Proprietary SV data collected at the SAP company	Linear/Tree-based/Neural network regression

directly from CVE-2020-28022,[18] but not from CVE-2021-2122.[19] The latter case requires techniques from section 2.6.1.1.

Most of the retrieval methods in this sub-theme have been formulated under the multi-class classification setting. One of the earliest works was conducted by Weerawardhana et al. [170]. This study extracted software names/versions, impacts and attacker's/user's action from SV descriptions on NVD using Stanford Named Entity Recognition (NER) technique, a.k.a. CRF classifier [186]. Later, Dong et al. [171] proposed to use a word/character-level Bi-LSTM to improve the performance of extracting vulnerable software names and versions from SV descriptions available on NVD and other SV databases/advisories (e.g., CVE Details [187], ExploitDB [67], SecurityFocus [72], SecurityTracker [188] and Openwall [189]). Based on the extracted entities, these authors also highlighted the inconsistencies in vulnerable software names and versions across different SV sources. Besides version products/names of SVs, Gonzalez et al. [172] used a majority vote of different ML models (e.g., SVM and Random forest) to extract the 19 entities of Vulnerability Description Ontology (VDO) [173] from SV descriptions to check the consistency of these descriptions based on the guidelines of VDO. Since 2020, there has been a trend in using DL models (e.g., Bi-LSTM, CNNs or BERT [80]/ELMo [190]) to extract different information from SV descriptions including required elements for generating MulVal [176] attack rules [174, 175] or SV types/root cause, attack type/vector [177, 178], Common Product Enumeration (CPE) [191] for standardizing names of vulnerable vendors/products/versions [179], part-of-speech [180] and relevant entities (e.g., vulnerable products, attack type, root cause) from ExploitDB to generate SV descriptions [182]. BERT models [80], pre-trained on general text (e.g., Wikipedia pages [82] or PTB corpus [181]) and fine-tuned on SV text, have also been increasingly used to address the data scarcity/imbalance for the retrieval tasks.

2.7.1.2 Cross-Source Vulnerability Patterns

The second sub-theme, *cross-source vulnerability patterns*, finds commonality and/or discovers latent relationships among SV sources to enrich information for SV assessment. Horawalavithana et al. [183] found a positive correlation between development activities (e.g., push/pull requests and issues) on GitHub and SV mentions on Reddit[20] and Twitter. These authors then used DL models (i.e., MLP [119] and LSTM [138]) to predict the appearance and sequence of development activities when SVs were mentioned on the two social media platforms. Xiao et al. [184] applied a translation-based graph embedding method to encode and predict the relationships among different SVs and the respective attack patterns and types. This work [184] was based on the DeepWeak model of Han et al. [154], but it still belongs to this sub-theme as they provided a multi-dimensional view of SVs using three different sources (NVD [16], CWE [28] and CAPEC [122]). Xiao et al. [184] envisioned that their knowledge graph can be extended to incorporate the source code introducing/fixing SVs.

2.7.1.3 Vulnerability Fixing Effort

The last sub-theme is *vulnerability fixing effort* that focuses on estimating SV fixing effort through proxies such as the SV fixing time, usually in days. Othmane and the co-authors were among the first to approach this problem. These authors first conducted a large-scale qualitative study at the SAP company and identified 65 important code-based, process-based and developer-based factors contributing to the SV fixing effort [192]. Later, the same

[18] https://nvd.nist.gov/vuln/detail/CVE-2020-28022
[19] https://nvd.nist.gov/vuln/detail/CVE-2021-21220
[20] https://reddit.com

group of authors [185] leveraged the identified factors in their prior qualitative study to develop various regression models such as linear regression, tree-based regression and neural network regression models, to predict time-to-fix SVs using the data collected at SAP. These authors found that code components containing detected SVs are more important for the prediction than SV types.

2.7.2 Theme Discussion

In the *Miscellaneous Tasks* theme, the key focus is on retrieving SV-related entities and characteristics from SV descriptions. The retrieval tasks are usually formulated as Named Entity Recognition from SV descriptions. However, we observed that NVD descriptions do not follow a consistent template [141, 142], posing significant challenges in labeling the entities for retrieval. The affected versions and vendor/product names of SVs also contain inconsistencies [171, 141, 142], making the retrieval tasks difficult. We recommend that data normalization and cleaning should be performed before labeling entities and building respective retrieval models to ensure the reliability of results.

Besides information retrieval, other tasks such as linking multi-sources, extracting cross-source patterns or estimating fixing effort are also useful to obtain richer SV information for assessment, yet these tasks are still in early stages. Linking multiple sources and their patterns is the first step towards building an SV knowledge graph to answer different queries regarding a particular SV (e.g., what systems are affected, exploitation status, how to fix, or what SVs are similar). In the future, such a knowledge graph can be extended to capture artifacts of SVs in emerging software types like AI-based systems [193]. Moreover, to advance SV fixing effort prediction, future work can consider adapting/customizing the existing practices/techniques used to predict fixing effort for general bugs [194, 195].

2.8 Analysis of Data-Driven Approaches for Software Vulnerability Assessment

We extract and analyze the five key elements for data-driven SV assessment: (*i*) Data sources, (*ii*) Model features, (*iii*) Prediction models, (*iv*) Evaluation techniques and (*v*) Evaluation metrics. These elements correspond to the four main steps in building data-driven models: data collection (data sources), feature engineering (model features), model training (prediction models) and model evaluation (evaluation techniques/metrics) [15, 197]. We present the most common practices for each element in Table 2.12.

2.8.1 Data Sources

Identifying and collecting rich and reliable SV-related data are the first tasks to build data-driven models for automating SV assessment tasks. As shown in Table 2.12, a wide variety of data sources have been considered to accomplish the five identified themes.

Across the five themes, NVD [16] and CVE [66] have been the most prevalently used data sources. The popularity of NVD/CVE is mainly because they publish expert-verified SV information that can be used to develop prediction models. Firstly, many studies have considered SV descriptions on NVD/CVE as model inputs. Secondly, the SV characteristics on NVD have been heavily used as assessment outputs in all the themes, e.g., CVSS Exploitability metrics for *Exploitation*, CVSS Impact/Scope metrics for *Impact*, CVSS severity score/levels for *Severity*, CWE for *Type*, CWE/CPE for *Miscellaneous tasks*. Thirdly, external sources on NVD/CVE have enabled many studies to obtain richer SV information (e.g., exploitation availability/time [93] or vulnerable code/crashes [60, 61]) and extract relationships among multiple SV sources to develop a knowledge graph of

TABLE 2.12: The frequent data sources, features, models, evaluation techniques and evaluation metrics used for the five identified SV assessment themes. **Notes**: The values are organized based on their overall frequency across the five themes. For the Prediction Model and Evaluation Metric elements, the values are first organized by categories (ML then DL for Prediction Model and classification then regression for Evaluation Metric) and then by frequencies. k-CV stands for k-fold cross-validation. The full list of values and their appearance frequencies for the five elements in the five themes can be found at https://figshare.com/s/da4d238ecdf9123dc0b8.

Source/Technique/Metric	Strengths	Weaknesses
Element: Data Source		
NVD/CVE/CVE Details (deprecated OSVDB)	• Report expert-verified information (with CVE-ID) • Contain CWE and CVSS entries for each SV • Link to external sources (e.g., official fixes)	• Missing/incomplete links to vulnerable code/fixes • Inconsistencies due to human errors • Delayed SV reporting and assignment of CVSS metrics
ExploitDB	• Report PoC exploits of SVs (with links to CVE-ID)	• May not lead to real exploits in the wild
Other security advisories (e.g., SecurityFocus, Symantec or X-Force)	• Report real-world exploits of SVs • Cover diverse SVs (including ones w/o CVE-ID)	• Some exploits may not have links to CVE entries for mapping with other assessment metrics
Informal sources (e.g., Twitter and darkweb)	• Early reporting of SVs (maybe earlier than NVD) • Contain non-technical SV information (e.g., financial damage or socio-technical challenges in addressing SVs)	• Contain non-verified and even misleading information • May cause adversarial attacks to assessment models
Element: Model Feature		
BoW/tf-idf/n-grams	• Simple to implement • Strong baseline for text-based inputs (e.g., SV descriptions in security databases/advisories)	• May suffer from vocabulary explosion (e.g., many new description words for new SVs) • No consideration of word context/order (maybe needed for code-based SV analysis) • Cannot handle Out-of-Vocabulary (OoV) words (can resolve with subwords [23])
Word2vec	• Capture nearby context of each word • Can reuse existing pre-trained model(s)	• Cannot handle OoV words (can resolve with fastText [196]) • No consideration of word order
DL model end-to-end trainable features	• Produce SV task-specific features	• May not produce high-quality representation for tasks with limited data (e.g., real-world exploit prediction)
Bidirectional Encoder Representations from Transformers (BERT)	• Capture contextual representation of text (i.e., the feature vector of a word is specific to each input) • Capture word order in an input • Can handle OoV words	• May require GPU to speed up feature inference • May be too computationally expensive and require too much data to train a strong model from scratch • May require fine-tuning to work well for a source task
Source/expert-defined metadata features	• Lightweight • Human interpretable for a task of interest	• Require SV expertise to define relevant features • Hard to generalize to new tasks
Element: Prediction Model		
Single ML models (e.g., Linear SVM, Logistic regression, Naïve Bayes)	• Simple to implement • Efficient to (re-)train on large data (e.g., using the entire NVD database)	• May be prone to overfitting • Usually do not perform as well as ensemble/DL models
Ensemble ML models (e.g., Random forest, XGBoost, LGBM)	• Strong baseline (usually stronger than single models) • Less prone to overfitting	• Take longer to train than single models
Latent Dirichlet Allocation (LDA – topic modeling)	• Require no labeled data for training • Can provide features for supervised learning models	• Require SV expertise to manually label generated topics • May generate human non-interpretable topics
Deep Multi-Layer Perceptron (MLP)	• Work readily with tabular data (e.g., manually defined features or BoW/tf-idf/n-grams)	• Perform comparably yet are more costly compared to ensemble ML models • Less effective for unstructured data (e.g., SV descriptions)
Deep Convolutional Neural Networks (CNN)	• Capture local and hierarchical patterns of inputs • Usually perform better than MLP for text-based data	• Cannot effectively capture sequential order of inputs (maybe needed for code-based SV analysis)
Deep recurrent neural networks (e.g., LSTM or Bi-LSTM)	• Capture short-/long-term dependencies from inputs • Usually perform better than MLP for text-based data	• May suffer from the information bottleneck issue (can resolve with attention mechanism [117]) • Usually take longer to train than CNNs
Deep graph neural networks (e.g., Graph convolutional network)	• Capture directed relationships among multiple SV entities and sources	• Require graph-structured inputs to work • More computationally expensive than other DL models
Deep transfer learning with fine-tuning (e.g., BERT with task-specific classification layer(s))	• Can improve the performance for tasks with small data (e.g., real-world exploit prediction)	• Require target task to have similar nature as source task
Deep constrastive learning (e.g., Siamese neural networks)	• Can improve performance for tasks with small data • Robust to class imbalance (e.g., CWE classes)	• Computationally expensive (two inputs instead of one) • Do not directly produce class-wise probabilities
Deep multi-task learning	• Can share features for predicting multiple tasks (e.g., CVSS metrics) simultaneously • Reduce training/maintenance cost	• Require predicted tasks to be related • Hard to tune the performance of individual tasks
Element: Evaluation Technique		
Single k-CV without test	• Easy to implement • Reduce the randomness of results with multiple folds	• No separate test set for validating optimized models (can resolve with separate test set(s)) • Maybe infeasible for expensive DL models • Use future data/SVs for training, may bias results
Single/multiple random train/test with/without val (using val to tune hyperparameters)	• Easy to implement • Reduce the randomness of results (the multiple version)	• May produce unstable results (the single version) • Maybe infeasible for expensive DL models (the multiple version) • Use future data/SVs for training, may bias results
Single/multiple time-based train/test with/without val (using val to tune hyper-parameters)	• Consider the temporal properties of SVs, simulating the realistic evaluation of ever-increasing SVs in practice • Reduce the randomness of results (the multiple version)	• Similar drawbacks for the single & multiple versions as the random counterparts • May result in uneven/small splits (e.g., many SVs in a year)
Element: Evaluation Metric		
F1-Score/Precision/Recall (classification)	• Suitable for imbalanced data (common in SV assessment tasks)	• Do not consider True Negatives in a confusion matrix (can resolve with Matthews Correlation Coefficient (MCC))
Accuracy (classification)	• Consider all the cells in a confusion matrix	• Unsuitable for imbalanced data (can resolve with MCC)
Area Under the Curve (AUC) (classification)	• Independent of prediction thresholds	• May not represent real-world settings (i.e., as models in practice mostly use fixed classification thresholds) • ROC-AUC may not be suitable for imbalanced data (can resolve with Precision-Recall-AUC)
Mean absolute (percentage) error/Root mean squared error (regression)	• Show absolute performance of models	• Maybe hard to interpret a value on its own without domain knowledge (i.e., whether an error of x is sufficiently effective)
Correlation coefficient (r)/ Coef. of determination (R^2) (regression)	• Show relative performance of models (0 – 1), where 0 is worst & 1 is best	• R^2 always increases when adding any new feature (can resolve with adjusted R^2)

SVs (e.g., [154, 184]). However, NVD/CVE still suffer from information inconsistencies [171, 141, 142] and missing relevant external sources (e.g., SV fixing code) [198]. Such issues motivate future work to validate/clean NVD data and utilize more sources for code-based SV assessment (see section 2.9).

To enrich the SV information on NVD/CVE, many other security advisories and SV databases have been commonly leveraged by the reviewed studies, notably ExploitDB [67], Symantec [68, 199], SecurityFocus [72], CVE Details [187] and OSVDB. Most of these sources disclose PoC (ExploitDB and OSVDB) and/or real-world (Symantec and Security Focus) exploits. However, real-world exploits are much rarer and different compared to PoC ones [33, 55]. It is recommended that future work should explore more data sources (other than the ones in Table 2.3) and better methods to retrieve real-world exploits, e.g., using semi-supervised learning [200] to maximize the data efficiency for exploit retrieval and/or few-shot learning for tackling the extreme exploit data imbalance issue [201]. Additionally, CVE Details and OSVDB are SV databases like NVD yet with a few key differences. CVE Details explicitly monitors Exploit-DB entries that may be missed on NVD and provides a more user-friendly interface to view/search SVs. OSVDB also reports SVs that do not appear on NVD (without CVE-ID), but this site was discontinued in 2016.

Besides official/expert-verified data sources, we have seen an increasing interest in mining SV information from informal sources that also contain non-expert generated content such as social media (e.g., Twitter) and darkweb. Especially, Twitter has been widely used for predicting exploits as this platform has been shown to contain many SV disclosures even before official databases like NVD [33, 140]. Recently, darkweb forums/sites/markets have also gained traction as SV mentions on these sites have a strong correlation with their exploits in the wild [48, 49]. However, SV-related data on these informal sources are much noisier because they neither follow any pre-defined structure nor have any verification and they are even prone to fake news [33]. Thus, the data integrity of these sources should be checked, potentially by checking the reputation of posters, to avoid inputting unreliable data to prediction models and potentially producing misleading findings.

2.8.2 Model Features

Collected raw data need to be represented by suitable features for training prediction models. There are three key types of feature representation methods in this area: term frequency (e.g., BoW, tf-idf and n-grams), DL learned features (e.g., BERT and word2vec) and source/expert-defined metadata (e.g., CVSS metrics and CPE on NVD or tweet properties on Twitter), as summarized in Table 2.12.

Regarding the term-frequency based methods, BoW has been the most popular one. Its popularity is probably because it is one of the simplest ways to extract features from natural language descriptions of SVs and directly compatible with popular ML models (e.g., Linear SVM, Logistic regression and Random forest) in section 2.8.3. Besides plain term count/frequency, other studies have also considered different weighting mechanisms such as inverse document frequency weighting (tf-idf) or tf-igm [128] inverse gravity moment weighting (tf-igm). Tf-igm has been shown to work better than BoW and tf-idf at classifying severity [124, 125]. Future work is still needed to evaluate the applicability and generalizability of tf-igm for other SV assessment tasks.

Recently, Neural Network (NN) or DL based features such as word2vec [202] and BERT [80] have been increasingly used to improve the performance of predicting CVSS exploitation/impact/severity metrics [21, 103], CWE types [150] and SV information retrieval [177, 178, 179]. Compared to BoW and its variants, NN and DL can extract more efficient and context-aware features from vast SV data [203]. NN/DL techniques rely on distributed representation to encode SV-related words using fixed-length vectors much smaller

than a vocabulary size. Moreover, these techniques capture the sequential order and context (nearby words) to enable better SV-related text comprehension (e.g., SV vs. general *exploit*). Importantly, these NN/DL learned features can be first trained in a non-SV domain with abundant data (e.g., Wikipedia pages [82]) and then transferred/fine-tuned in the SV domain to address limited/imbalanced SV data [56]. The main concern with these sophisticated NN/DL features is their limited interpretability, which is an exciting future research area [204].

The metadata about SVs can also complement the missing information in descriptions or code for SV assessment. For example, prediction of exploits and their characteristics have been enhanced using CVSS metrics [49], CPE [106] and SV types [57] on NVD. Additionally, Twitter-related statistics (e.g., number of followers, likes and retweets) have been shown to increase the performance of predicting SV exploitation, impact and severity [33, 93]. Recently, alongside features extracted from vulnerable code, the information about a software development process and involved developers have also been extracted to predict SV fixing effort [185]. Currently, metadata-based and text-based features have been mainly integrated by concatenating their respective feature vectors (e.g., [140, 92, 48, 49]). An alternative yet unexplored way is to build separate models for each feature type and then combine these models using meta-learning (e.g., model stacking [205]).

2.8.3 Prediction Models

The extracted features enter a wide variety of ML/DL-based prediction models shown in Table 2.12 to automate various SV assessment tasks. Classification techniques have the largest proportion, while regression and unsupervised techniques are less common.

Linear SVM [206] has been the most frequently used classifier, especially in the Exploitation, Impact and Severity themes. This popularity is reasonable as Linear SVM works well with the commonly used features, i.e., BoW and tf-idf, as mentioned in section 2.8.2. Besides Linear SVM, Random forest, Naïve Bayes and Logistic regression have also been common classification models. In recent years, advanced boosting models (e.g., XGBoost [78] and LGBM [79]), and more lately, DL techniques (e.g., CNN [120] and (Bi-)LSTM with attention [117]) have been increasingly utilized and shown better results than simple ML models like Linear SVM or Logistic regression. In this area, some DL models are essential for certain tasks, e.g., building SV knowledge graph from multiple sources with graph neural networks [115]. DL models also offer solutions to data-related issues such as addressing class imbalance (e.g., deep Siamese network [207]) or improving data efficiency (e.g., deep multi-task learning [116]). Whenever applicable, it is recommended that future work should still consider simple baselines alongside sophisticated ones as simple methods can perform on par with advanced ones [208].

Besides classification, various prediction models have also been investigated for regression (e.g., predicting exploit time, severity score and fixing time). Linear SVM has again been the most commonly used regressor as SV descriptions have usually been the regression input. Notably, many studies in the Severity theme did not build regression models to directly obtain the severity score (e.g., [96, 98, 100, 101, 22]). Instead, they used the formulas defined by assessment frameworks (e.g., CVSS versions 2/3 [110, 111] or WIVSS [136]) to compute the severity score from the base metrics predicted by respective classification models. We argue that more effort should be invested in determining the severity score directly from SV data as these severity formulas can be subjective [209]. We also observe that there is still limited use of DL models for regression compared to classification.

In addition to supervised (classification/regression) techniques, unsupervised learning has also been considered for extracting underlying patterns of SV data, especially in the Type theme. Latent Dirichlet Allocation (LDA) [30] has been the most commonly used

topic model to identify latent topics/types of SVs without relying on a labeled taxonomy. The identified topics were mapped to the existing SV taxonomies such as CWE [156] and OWASP [157, 158]. The topics generated by topic models like LDA can also be used as features for classification/regression models [105] or building topic-wise models to capture local SV patterns [210]. However, definite interpretations for unsupervised outputs are challenging to obtain as they usually rely on human judgement [211].

2.8.4 Evaluation Techniques

It is important to evaluate a trained model to ensure the model meets certain requirements (e.g., advancing the state-of-the-art). The evaluation generally needs to be conducted on a different set of data other than the training set to avoid overfitting and objectively estimate model generalizability [119]. The commonly used evaluation techniques are summarized in Table 2.12.

The reviewed studies have mostly used one or multiple validation and/or test sets[21] to evaluate their models, in which each validation/test set has been either randomly or time-based selected. Specifically, k-fold cross-validation has been one of the most commonly used techniques. The number of folds has usually been 5 or 10, but less standard values like 4 [60] have also been used. However, k-fold cross-validation uses all parts of data at least once for training; thus, there is no hidden test set to evaluate the optimal model with the highest (cross-)validation performance.

To address the lack of hidden test set(s), a common practice in the studied papers has been to split a dataset into single training and test sets, sometimes with an additional validation set for tuning hyperparameters to obtain an optimal model. Recently, data has been increasingly split based on the published time of SVs to better reflect the changing nature of ever-increasing SVs [23].[22] However, the results reported using single validation/test sets may be unstable (i.e., unreproducible results using different set(s)) [212].

To ensure both the time order and reduce the result randomness, we recommend using multiple splits of training and test sets in combination with time-based validation in each training set. Statistical analyses (e.g., hypothesis testing and effect size) should also be conducted to confirm the reliability of findings with respect to the randomization of models/data in multiple runs [213].

2.8.5 Evaluation Metrics

Evaluating different aspects of a model requires respective proper metrics. The popular metrics for evaluating the tasks in each theme are given in Table 2.12.

Across the five themes, Accuracy, Precision, Recall and F1-Score [214] have been the most commonly used metrics because of a large number of classification tasks in the five themes. However, Accuracy is not a suitable measure for SV assessment tasks with imbalanced data (e.g., SVs with real-world exploits vs. non-exploited SVs). The sample size of one class is much smaller than the others, and thus the overall Accuracy would be dominated by the majority classes. Besides these four commonly used metrics, AUC based on the ROC curve (ROC-AUC) [214] has also been considered as it is threshold-independent. However, we suggest that ROC-AUC should be used with caution in practice as most deployed models would have a fixed decision threshold (e.g., 0.5). Instead of ROC-AUC, we suggest Matthews Correlation Coefficient [214] (MCC) as a more meaningful evaluation

[21] Validation set(s) helps optimize/tune a model (finding the best task/data-specific hyperparameters), and test set(s) evaluates the optimized/tuned model. Using only validation set(s) means evaluating a model with default/pre-defined hyperparameters.

[22] This study is presented in Chapter 3.

metric to be considered as it explicitly captures all values in a confusion matrix, and thus has less bias in results.

For regression tasks, various metrics have been used such as Mean absolute error, Mean absolute percentage error, Root mean squared error [22] as well as Correlation coefficient (r) and Coefficient of determination (R^2) [185]. Note that *adjusted* R^2 should be preferred over R^2 as R^2 would always increase when adding a new (even irrelevant) feature.

A model can have a higher value of one metric yet lower values of others.[23] Therefore, we suggest using a combination of suitable metrics for a task of interest to avoid result bias towards a specific metric. Currently, most studies have focused on evaluating model effectiveness, i.e., how well the predicted outputs match the ground-truth values. Besides effectiveness, other aspects (e.g., efficiency in training/deployment and robustness to input changes) of models should also be evaluated to provide a complete picture of model applicability in practice.

2.9 Chapter Summary

SV assessment is crucial to optimize resource utilization in addressing SVs at scale. This phase has witnessed radical transformations following the increasing availability of SV data from multiple sources and advances in data-driven techniques. We presented a taxonomy to summarize the five main directions of the research work so far in this area. We also identified and analyzed the key practices to develop data-driven models for SV assessment in the reviewed studies.

Despite the great potential of data-driven approaches for SV assessment, we highlighted the following three open challenges limiting the practical application of the field:[24]

1. **Unrealistic evaluation settings for report-level SV assessment models**. Most of the reviewed studies have evaluated their prediction models without capturing many factors encountered during the deployment of such models to production. Specifically, the models used in practice would require to handle new SV data over time. In the context of report-level SV assessment, Out-of-Vocabulary (OoV) words in SV descriptions of new SVs need to be properly accommodated to avoid performance degradation of prediction models. The impact of using time-based splits rather than random splits (e.g., k-fold cross-validation) for these models to avoid leaking unseen (future) patterns to the model training also requires further investigation. We address the first challenge in Chapter 3.

2. **Untimely SV assessment models**. Although SV descriptions have been commonly used as model inputs (see section 2.8.1), these descriptions are usually published long after SVs introduced/discovered [25] and even after SV are fixed [215, 216] in codebases. One potential solution to this issue is to directly perform SV assessment using (vulnerable) source code. The key benefit of using vulnerable code for SV assessment is that such code is always available/required before SV fixing. Thus, performing code-based SV assessment can be done even when SV reports are not (yet) available. Version control systems like GitHub[25] can provide such vulnerable code for SV assessment. Leveraging data from these version control systems, we address the second challenge in Chapters 4 and 5, in which data-driven SV assessment is investigated on the code function and code commit levels, respectively.

[23] https://stackoverflow.com/questions/34698161

[24] The presented challenges are the ones that have been addressed in this thesis. More challenges related to data-driven SV assessment can be found in Chapter 7.

[25] https://github.com

TABLE 2.13: The mapping between the themes/tasks and the respective studies collected from May 2021 to February 2022.

Theme/Task	Studies
Exploitation Prediction	[217], [218], [219], [220], [221], [222], [223], [224], [225]
Impact Prediction	[218], [219], [220], [222], [225]
Severity Prediction	[218], [219], [220], [222], [225]
Type Prediction	[226], [227], [228], [229], [230]
Miscellaneous Tasks	[218], [231], [232], [233], [234]

3. **Lack of utilization of developers' real-world SV concerns**. The current SV data hardly contain specific developers' concerns and practices when addressing real-world SVs. *Developer Question & Answer (Q&A) platforms* like Stack Overflow[26] and Security StackExchange[27] contain millions of posts about different challenges and solutions shared by millions of developers when tackling different software-related issues in real-world scenarios. The rich data on Q&A sites can be collected and analyzed to determine the key concerns that practitioners are facing while addressing SV in practice. Such SV concerns can be incorporated into other technical metrics like CVSS metrics for more thoroughly assessing and prioritizing SVs. For example, the fixing effort of SVs may depend on the technical difficulty of implementing the respective mitigation strategies in a language or system of interest. We address the third challenge in Chapter 6.

2.10 Appendix - Ever-Growing Literature on Data-Driven SV Assessment

Given that data-driven SV assessment is an emerging field, there have been many new contributions in this area since this literature review was conducted in April 2021. In this appendix, we would like to briefly outline the recent trends in data-driven SV assessment from May 1, 2021 to February 28, 2022 (at the time of writing this thesis). We have also maintained a website (`https://github.com/lhmtriet/awesome-vulnerability-assessment`) to keep track of the latest contributions of data-driven SV assessment for researchers and practitioners in the field.

To obtain new papers from May 2021 to February 2022, we followed the same methodology of study selection, as described in section 2.2.2. We obtained 228 new studies returned from the search on the online databases. We then applied the inclusion/exclusion criteria, as given in Table 2.2, on the titles, abstracts and full-texts of the curated papers to obtain the list of relevant papers. After the filtering steps, we selected 18 papers relevant to data-driven SV assessment from May 2021 to February 2022.[28] The categorization of these selected papers into the five themes, as described in section 2.2.3, is given in Table 2.13.

The key patterns and practices of these 18 papers are summarized as follows:

- All the tasks tackled by the studies have aligned with the ones presented in this literature review, reinforcing the robustness of our taxonomy presented in Fig. 2.1.

[26] `https://stackoverflow.com`

[27] `https://security.stackexchange.com`

[28] We do not claim that this list of updated papers is complete, yet we believe that we have covered the representative ones. It should also be noted that we did not include our own papers as these papers would be presented in subsequent chapters of this thesis.

- Among the *themes*, Exploitation has attracted the most contributions from these new studies, similar to that of the reviewed papers prior to May 2021. In addition, prediction of the CVSS exploitability, impact, and severity metrics is still the most common task. However, it is encouraging to see that more studies have worked on CVSS version 3.1, which is closer to the current industry standard.
- Regarding *data sources*, NVD/CVE is still most prevalently used. Besides CAPEC [122], MITRE ATT&CK Framework[29] is a new source of attack patterns being utilized [217].
- Regarding *model features*, BERT [80] has been commonly used to extract contextual feature embeddings from SV descriptions for various tasks.
- Regarding *prediction models*, DL techniques, which are mainly based on CNN and/or (Bi-)LSTM with attention, have been increasingly used to improve the performance of the tasks.
- Regarding *evaluation techniques* and *evaluation metrics*, since most of the tasks are the same as before, evaluation practices have largely stayed the same as the ones presented in sections 2.8.4 and 2.8.5.

Overall, the three key practical challenges presented in section 2.9 are still not addressed by the new studies. Thus, we believe that our contributions/solutions to address these challenges in this thesis would set the foundation for future research to further improve the practical applicability of SV assessment using data-driven approaches.

[29] https://attack.mitre.org/

Chapter 3

Automated Report-Level Software Vulnerability Assessment with Concept Drift

Related publication: This chapter is based on our paper titled "*Automated Software Vulnerability Assessment with Concept Drift*", published in the 16$^{\text{th}}$ International Conference on Mining Software Repositories (MSR), 2019 (CORE A) [23].

In our literature review in Chapter 2, Software Engineering researchers are increasingly using Natural Language Processing (NLP) techniques to automate Software Vulnerability (SV) assessment using SV descriptions in public repositories. However, the existing NLP-based approaches suffer from *concept drift*. This problem is caused by a lack of proper treatment of new (out-of-vocabulary) terms for evaluating unseen SVs over time. To perform automated SV assessment with *concept drift* using SV descriptions present in SV reports, in Chapter 3, we propose a systematic approach that combines both character and word features. The proposed approach is used to predict seven Vulnerability Characteristics (VCs). The optimal model of each VC is selected using our customized time-based cross-validation method from a list of eight NLP representations and six well-known Machine Learning models. We use the proposed approach to conduct large-scale experiments on more than 100,000 SVs in National Vulnerability Database (NVD). The results show that our approach can effectively tackle the *concept drift* issue of the SVs' descriptions reported from 2000 to 2018 in NVD even without retraining the model. In addition, our approach performs competitively compared to the existing word-only method. We also investigate how to build compact *concept-drift*-aware models with much fewer features and give some recommendations on the choice of classifiers and NLP representations for report-level SV assessment.

3.1 Introduction

Software Vulnerability (SV) reports in public repositories, such as National Vulnerability Database (NVD) [16], have been widely leveraged to automatically predict SV characteristics using data-driven approaches (see Chapter 2). However, data in these SV reports have the temporal property since many new terms appear in the descriptions of SVs. Such terms are a result of the release of new technologies/products or the discovery of a zero-day attack or SV; for example, NVD received more than 13,000 new SVs in 2017 [16]. The appearance of new concepts makes SV data and patterns change over time [160, 152, 156], which is known as *concept drift* [34]. For example, the keyword Android has only started appearing in NVD since 2008, the year when Google released Android. We assert that such new SV terms can cause problems for building report-level SV assessment models.

Previous studies [22, 49, 32] have suffered from *concept drift* as they have usually mixed new and old SVs in the model validation step. Such approach accidentally merges new SV information with existing one, which can lead to biased results. Moreover, the previous work of SV analysis [22, 20, 49, 32, 97, 95, 47] used predictive models with only word features without reporting how to handle novel or extended concepts (e.g., new versions of the same software) in new SVs' descriptions. Research on machine translation [235, 236, 237, 238] has shown that unseen (Out-of-Vocabulary (OoV)) terms can make existing word-only models less robust to future prediction due to their missing information. For SV prediction, Han et al. [21] did use random embedding vectors to represent the OoV words, which still discards the relationship between new and old concepts. Such observations motivated us to tackle the research problem "**How to handle the *concept drift* issue of the SV descriptions in public repositories to improve the robustness of automated report-level SV assessment?**" It appears to us that it is important to address the issue of *concept drift* to enable the practical applicability of automated SV assessment tools. To the best of our knowledge, there has been no existing work to systematically address the *concept drift* issue in report-level SV assessment.

To perform report-level SV assessment with *concept drift* using SV descriptions in public repositories, we present a Machine Learning (ML) model that utilizes both character-level and word-level features. We also propose a customized time-based version of cross-validation method for model selection and validation. Our cross-validation method splits the data by year to embrace the temporal relationship of SVs. We evaluate the proposed model on the prediction of seven Vulnerability Characteristics (VCs), i.e., Confidentiality, Integrity, Availability, Access Vector, Access Complexity, Authentication, and Severity. Our key **contributions** are:

1. We demonstrate the *concept drift* issue of SVs using concrete examples from NVD.
2. We investigate a customized time-based cross-validation method to select the optimal ML models for SV assessment. Our method can help prevent future SV information from being leaked into the past in model selection and validation steps.
3. We propose and extensively evaluate an effective Character-Word Model (CWM) to assess SVs using the descriptions with *concept drift*. We also investigate the performance of low-dimensional CWM models. We provide our models and associated source code for future research at `https://github.com/lhmtriet/MSR2019`.

Chapter organization. Section 3.2 introduces SV descriptions and VCs. Section 3.3 describes our proposed approach. Section 3.4 presents the experimental design of this work. Section 3.5 analyzes the experimental results and discusses the findings. Section 3.6 identifies the threats to validity. Section 3.7 covers the related works. Section 3.8 concludes and suggests some future directions.

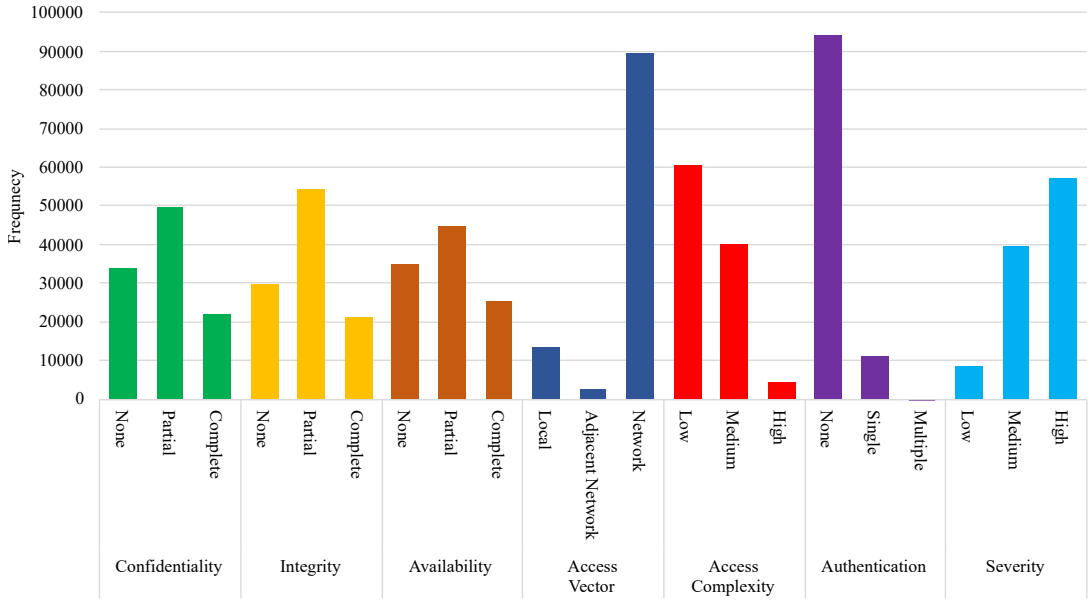


FIGURE 3.1: Frequencies of each class of the seven VCs.

3.2 Background

Software Vulnerability (SV) assessment is an important step in the SV lifecycle, which determines various characteristics of detected SVs [10]. Such characteristics support developers to understand the nature of SVs, which can inform prioritization and remediation strategies. For example, if an SV can severely damage the confidentiality of a system, e.g., allowing attackers to access/steal sensitive information, this SV should have a high fixing priority. A fixing protocol to ensure confidentiality can then be followed, e.g., checking/enforcing privileges to access the affected component/data.

National Vulnerability Database [16] (NVD) is one of the most popular and trustworthy sources for SV assessment. NVD is maintained by governmental bodies (National Cyber Security and Division of the United States Department of Homeland Security). This site inherits unique SV identifiers and descriptions from Common Vulnerabilities and Exposures (CVE) [66]. For SV assessment, NVD provides expert-verified assessment metrics, namely Common Vulnerability Scoring System (CVSS) [29], for each reported SV.

CVSS is one of the most commonly used frameworks by both researchers and practitioners to perform SV assessment. There are two main versions of CVSS, namely versions 2 and 3, in which version 3 only came into effect in 2015. CVSS version 2 is still widely used as many SVs prior to 2015 can yet pose threats to contemporary systems. For instance, the SV with CVE-2004-0113 first found in 2004 was exploited in 2018 [239]. Hence, we adopt the assessment metrics of CVSS version 2 as the outputs for the SV assessment models in this study.

CVSS version 2 provides metrics to quantify the three main aspects of SVs, namely exploitability, impact, and severity. We focus on the *base* metrics because the temporal metrics (e.g., exploit availability in the wild) and environmental metrics (e.g., potential impact outside of a system) are unlikely obtainable from project artifacts (e.g., SV code/reports) alone. Specifically, the base *Exploitability* metrics examine the technique (Access Vector) and complexity to initiate an exploit (Access Complexity) as well as the authentication requirement (Authentication). The base *Impact* metrics of CVSS focus on the system Confidentiality, Integrity, and Availability. The Exploitation and Impact metrics are used to compute the *Severity* of SVs. Severity approximates the criticality of an SV.

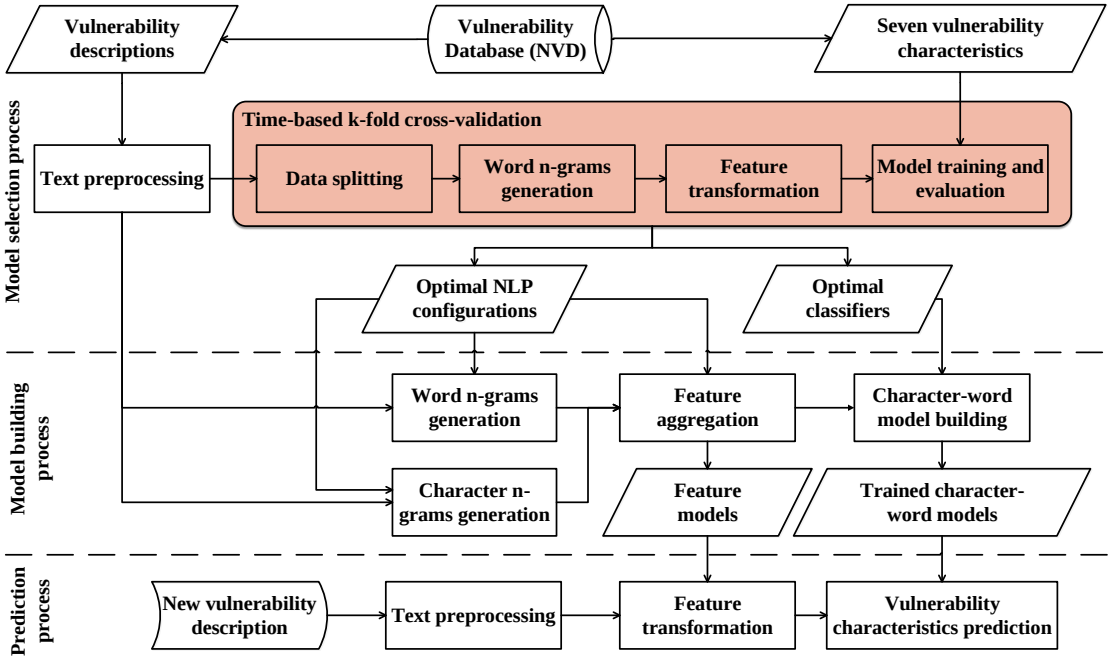


FIGURE 3.2: Workflow of our proposed model for report-level software vulnerability assessment with *concept drift*.

Nevertheless, relying solely on Severity may be insufficient because an SV with medium severity may still have high impacts as it is considerably complex to be exploited. It is important to assign a high fixing priority to such an SV as an affected system would face tremendous risks in case of a successful cyber-attack. Therefore, in this study, we consider all the base metrics of CVSS version 2 (i.e., Confidentiality, Integrity, Availability, Access Vector, Access Complexity, Authentication and Severity), as shown in Fig. 3.1, for developing SV assessment models. From the perspective of ML, predicting CVSS metrics is a classification problem, which can be solved readily using ML algorithms. It is noted that Access Vector, Access Complexity and Authentication characteristics suffer the most from the issue of imbalanced data, in which the number of elements in the minority class is much smaller compared to those of the other classes.

3.3 The Proposed Approach

3.3.1 Approach Overview

The overall workflow of our proposed approach is given in Fig. 3.2. Our approach consists of three processes: model selection, model building and prediction. The first two processes work on the training set, while the prediction process performs on either a separate testing set or new SV descriptions. The first model selection has two steps: Text preprocessing and Time-based k-fold cross-validation. The text preprocessing step (see section 3.3.2) is necessary to reduce noise in text to build a better assessment model. Next, the preprocessed text enters the time-based k-fold cross-validation step to select the optimal classifier and Natural Language Processing (NLP) representations for each VC. It should be noted that this step only tunes the word-level models instead of the combined models of both word and character features. One reason is that the search space of the combined model is much larger than that of the word-only model since we at least have to consider different NLP representations for character-level features. The computational resource to extract

TABLE 3.1: Word and character n-grams extracted from the sentence "Hello World". '_' represents a space.

n-grams	Words	Characters
1	Hello, World	H, e, l, l, o, W, o, r, l, d
2	Hello World	He, el, ll, lo, o_, _W, Wo, or, rl, ld

character-level n-grams is also more than that of word-level counterparts. Section 3.3.3 provides more details about the time-based k-fold cross-validation method.

Next comes the model building process with four main steps: (*i*) word n-grams generation, (*ii*) character n-grams generation, (*iii*) feature aggregation and (*iv*) character-word model building. Steps (*i*) and (*ii*) use the preprocessed text in the previous process to generate word and character n-grams based on the identified optimal NLP representations of each VC. The word n-grams generation step (*i*) here is the same as the one in the time-based k-fold cross-validation of the previous process. An example of the word and character n-grams in our approach is given in Table 3.1. Such character n-grams increase the probability of capturing parts of OoV terms due to *concept drift* in SV descriptions. Subsequently, both levels of the n-grams and the optimal NLP representations are input into the feature aggregation step (*iii*) to extract features from the preprocessed text using our proposed algorithm in section 3.3.4. This step also combines the aggregated character and word vocabularies with the optimal NLP representations of each VC to create the feature models. We save such models to transform data of future prediction. In the last step (*iv*), the extracted features are trained with the optimal classifiers found in the model selection process to build the complete character-word models for each VC to perform automated report-level SV assessment with *concept drift*.

In the prediction process, a new SV description is first preprocessed using the same text preprocessing step. Then, the preprocessed text is transformed to create features by the feature models saved in the model building process. Finally, the trained character-word models use such features to determine each VC.

3.3.2 Text Preprocessing of SV Descriptions

The text preprocessing is an important step for any NLP task [240]. We use the following text preprocessing techniques: (*i*) removal of stop words and punctuations, (*ii*) conversion to lowercase and (*iii*) stemming. The stop words are combined from the default lists of the *scikit-learn* [241] and *nltk* [242] libraries. We only remove the punctuations followed by at least one space or the ones at the end of a sentence. This punctuation removal method keeps important words in the software and security contexts such as "*input.c*", "*man-in-the-middle*", "*cross-site*".

Subsequently, the stemming step is done using the Porter Stemmer algorithm [243] in the *nltk* library. Stemming is needed to avoid two or more words with the same meaning but in different forms (e.g., "*allow*" vs. "*allows*"). The main goal of stemming is to retrieve consistent features (words), thus any algorithm that can return each word's root should work. Researchers may use lemmatization, which is relatively slower as it also considers the surrounding context.

3.3.3 Model Selection with Time-based k-Fold Cross-Validation

We propose a time-based cross-validation method (see Fig. 3.3) to select the best classifier and NLP representation for each VC. The idea has been inspired by the time-series domain [244]. As shown in Fig. 3.2, our method has four steps: (*i*) data splitting, (*ii*) word

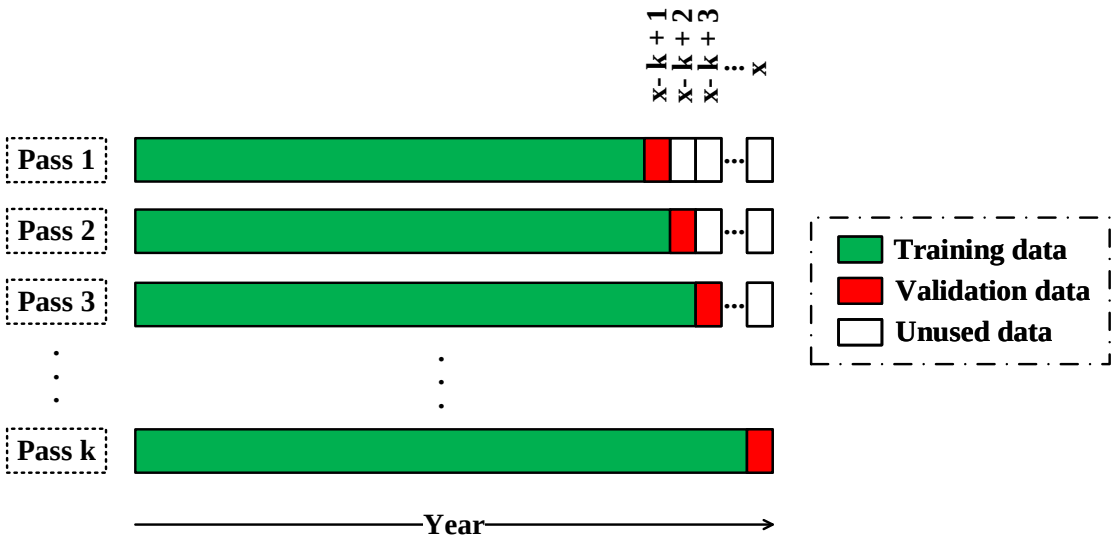


FIGURE 3.3: Our proposed time-based cross-validation method. **Note**: x is the final year in the original training set and k is the number of cross-validation folds.

TABLE 3.2: The eight configurations of NLP representations used for model selection. **Note**: '✓' is selected, '-' is non-selected.

Configuration	Word n-grams	tf-idf
1	1	-
2	1	✓
3	1-2	-
4	1-3	-
5	1-4	-
6	1-2	✓
7	1-3	✓
8	1-4	✓

n-grams generation, (*iii*) feature transformation, and (*iv*) model training and evaluation. Data splitting explicitly considers the time order of SVs to ensure that in each pass/fold, the new information of the validation set does not exist in the training set, which maintains the temporal property of SVs. New terms can appear at different time during a year; thus, the preprocessed text in each fold is split by year explicitly, not by equal sample size as in traditional time-series splits;[1] e.g., SVs from 1999 to 2010 are for training and those in 2011 are for validation in a pass/fold.

After data splitting in each fold, we use the training set to generate the word n-grams. Subsequently, with each of the eight NLP configurations in Table 3.2, the feature transformation step uses the word n-grams as the vocabulary to transform the preprocessed text of both training and validation sets into the features for building a model. We create the NLP configurations from various values of n-grams combined with either term frequency or tf-idf measure. Uni-gram with term frequency is also called Bag-of-Words (BoW). These NLP representations have been selected since they are popular and have performed well for SV analysis [22, 49, 95]. For each NLP configuration, the model training and evaluation

[1] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.TimeSeriesSplit.html

Algorithm 1: Feature aggregation algorithm to transform the documents with the aggregated word and character-level features.

Input: List of SV descriptions: D_{in}
Set of word-level n-grams: $F_w = \{f_{1w}, f_{2w}, ..., f_{nw}\}$
Set of character-level n-grams: $F_c = \{f_{1c}, f_{2c}, ..., f_{mc}\}$
The minimum and maximum character n-grams: min_{n-gram} and max_{n-gram}
The optimal NLP configuration of the current VC: *config*
Output: The aggregated data matrix: $\mathbf{X}_{agg}$
The word and character feature models: $\text{model}_w, \text{model}_c$

1 $slt\_chars \leftarrow \emptyset$
2 **foreach** $f_i \in F_c$ **do**
3 $tokens \leftarrow f_i$ trimmed and split by space
4 **if** *(size of tokens = 1)* **and** *((length of the first element in tokens) > 1)* **then**
5 $slt\_chars \leftarrow slt\_chars + \{tokens\}$
6 $diff\_words \leftarrow F_w - slt\_chars$
7 $\text{model}_w \leftarrow \text{Feature\_transformation}(diff\_words, config)$
8 $\text{model}_c \leftarrow \text{Feature\_transformation}(slt\_chars, min_{n-gram} - 1, max_{n-gram}, config)$
9 $\mathbf{X}_{word} \leftarrow D_{in}$ transformed with model_w
10 $\mathbf{X}_{char} \leftarrow D_{in}$ transformed with model_c
11 $\mathbf{X}_{agg} \leftarrow \text{horizontal\_append}(\mathbf{X}_{word}, \mathbf{X}_{char})$
12 **return** $\mathbf{X}_{agg}$, model_w, model_c

step trains six classifiers (see section 3.4.3) on the training set and then evaluates the models on the validation set using different evaluation metrics (see section 3.4.4). The model with the highest average cross-validated performance is selected for a VC. The process is repeated for every VC, then the optimal classifiers and NLP representations are returned for all seven VCs.

3.3.4 Feature Aggregation Algorithm

We propose Algorithm 1 to combine word and character n-grams in the model building process to create features for our character-word model. Six inputs of the algorithm are (*i*) input descriptions, (*ii*) word n-grams, (*iii*) character n-grams, (*iv*) the minimum, (*v*) the maximum number of character n-grams, and (*vi*) the optimal NLP configuration of a VC. The main output is a feature matrix containing the term weights of the documents transformed by the aggregated character and word vocabularies to build the character-word models. We also output the character and word feature models for future prediction of VCs.

Steps 2-7 of the algorithm filter the character features. More specifically, step 3 removes (trims) spaces from both ends of each feature. Then, we split such feature by space(s) to determine how many words to which the character(s) belongs. Subsequently, steps 4-6 retain only the character features that are parts of single words (size of $tokens = 1$), except the single characters such as *x*, *y*, *z* ((length of the first element in *tokens*) > 1). The n-gram characters with space(s) in between represent more than one word, which can make a classifier more prone to overfitting. Similarly, single characters are too short and they can belong to too many words, which is likely to make a model hardly generalizable. In fact, '*a*' is a meaningful single character, but it has been already removed as a stop word. The characters can even represent a whole word (e.g., "*attack*" token with $max_{n-gram} \geq 6$). In such cases, step 8 removes duplicated word-level features ($F_w - slt\_chars$). Based on the assumption that unseen or misspelled terms can share common characters with existing

words, such choice can enhance the probability of a model capturing the OoV words in new descriptions. Retaining only the character features also helps reduce the number of features and model overfitting. After that, steps 9-10 define the feature models model_w and model_c using the word ($diff\_words$) and character (slt_chars) vocabularies, respectively, along with the NLP configurations to transform the input documents into feature matrices for building an assessment model. Steps 11-12 then use the two defined word and character models to actually transform the input documents into the feature matrices $\mathbf{X}_{word}$ and $\mathbf{X}_{word}$, respectively. Step 13 concatenates the two feature matrices by columns. Finally, step 14 returns the final aggregated feature matrix $\mathbf{X}_{agg}$ along with the word and character feature models model_w and model_c.

3.4 Experimental Design and Setup

All the classifiers and NLP representations (n-grams, term frequency and tf-idf) in this work were implemented in the *scikit-learn* [241] and *nltk* [242] libraries in Python. Our code ran on a fourth-generation Intel Core i7-4200HQ CPU (four cores) running at 2.6 GHz with 16 GB of RAM.

3.4.1 Research Questions

Our research aims at addressing the *concept drift* issue in SVs' descriptions to improve the robustness of both model selection and prediction steps of report-level SV assessment. Specifically, we evaluate our two-phase character-word models. The first phase selects the optimal word-only models for each VC. The second phase incorporates character features to build character-word models. We raise and answer four Research Questions (RQs):

- **RQ1**: *Is our time-based cross-validation more effective than a non-temporal method to handle concept drift in the model selection step for report-level SV assessment?* To answer RQ1, we first identify the new terms in SV descriptions. We associate such terms with their release or discovery years. We then use qualitative examples to demonstrate information leakage in the non-temporal model selection step. We also quantitatively compare the effectiveness of the proposed time-based cross-validation method with a traditional non-temporal one for addressing the temporal relationship in the context of report-level SV assessment.

- **RQ2**: *Which are the optimal models for multi-classification of each SV characteristic?* To answer RQ2, we present the optimal models (i.e., classifiers and NLP representations) using word features for each VC selected by a five-fold time-based cross-validation method (see section 3.3.3). We also compare the performance of different classes of models (single vs. ensemble) and NLP representations to give recommendations for future use.

- **RQ3**: *How effective is our character-word model to perform automated report-level SV assessment with concept drift?* For RQ3, we first demonstrate how the OoV phrases identified in RQ1 can affect the performance of the existing word-only models. We then highlight the ability of the character features to handle the *concept drift* issue of SVs. We also compare the performance of our character-word model with those of the word-only (without handling *concept drift*) and character-only models.

- **RQ4**: *To what extent can low-dimensional model retain the original performance?* The features of our proposed model in RQ3 are high-dimensional and sparse. Hence, we evaluate a dimensionality reduction technique (i.e., Latent Semantic Analysis [245])

and the sub-word embeddings (i.e., fastText [196, 246]) to show how much information of the original model is approximated in lower dimensions. RQ4 findings can facilitate the building of more efficient *concept-drift*-aware predictive models.

3.4.2 Dataset

We retrieved 113,292 SVs from NVD in JSON format. The dataset contains the SVs from 1988 to 2018. We discarded 5,926 SVs that contain "** REJECT **" in their descriptions since they had been confirmed duplicated or incorrect by experts. Seven VCs of CVSS 2 (see section 3.2) were used as our SV assessment metrics. It turned out that there are 2,242 SVs without any value of CVSS 2. Therefore, we also removed such SVs from our dataset. Finally, we obtained a dataset containing **105,124 SVs** along with their descriptions and the values of seven VCs indicated previously. For evaluation purposes, we followed the work in [22] to use the year of 2016 to divide our dataset into training and testing sets with the sizes of 76,241 and 28,883, respectively. The primary reason for splitting the dataset based on the time order is to consider the temporal relationship of SVs.

3.4.3 Machine Learning Models for Report-Level SV Assessment

To solve our multi-class classification (report-level SV assessment) problem, we used six well-known ML models. These classifiers have achieved great results in recent data science competitions such as Kaggle [247]. We provide brief descriptions and the hyperparameters of each classifier below.

- Naïve Bayes (NB) [248] is a simple probabilistic model that is based on Bayes' theorem. This model assumes that all the features are conditionally independent with respect to each other. In this study, NB had no tuning hyperparameter during the validation step.

- Logistic Regression (LR) [249] is a linear classifier in which the logistic function is used to convert a linear output into a respective probability. The one-vs-rest scheme was applied to split the multi-class problem into multiple binary classification problems. In this work, we selected the optimal value of the regularization parameter for LR from the list of values: 0.01, 0.1, 1, 10, 100.

- Support Vector Machine (SVM) [206] is a classification model in which a maximum margin is determined to separate the classes. For NLP, the linear kernel is preferred because of its more scalable computation and sparsity handling [250]. The tuning regularization values of SVM are the same as LR.

- Random Forest (RF) [251] is a bagging model in which multiple decision trees are combined to reduce the variance and sensitivity to noise. The complexity of RF is mainly controlled by (*i*) the number of trees, (*ii*) maximum depth, and (*iii*) maximum number of leaves. (*i*) tuning values were: 100, 300, 500. We set (*ii*) to *unlimited*, which makes the model the highest degree of flexibility and easier to adapt to new data. For (*iii*), the tuning values were 100, 200, 300 and *unlimited*.

- XGBoost - Extreme Gradient Boosting (XGB) [78] is a variant Gradient Boosting Tree Model (GBTM) in which multiple weak tree-based classifiers are combined and regularized to enhance the robustness of the overall model. Three hyperparameters of XGB that require tuning were the same as RF. It should be noted that the *unlimited* value of the maximum number of leaves is not applicable to XGB.

- Light Gradient Boosting Machine (LGBM) [79] is a light-weight version of GBTM. Its main advantage is the scalability since the sub-trees are grown in a leaf-wise manner rather than depth-wise of other GBT algorithms. Three hyperparameters of LGBM that require tuning were the same as XGB.

In this work, we considered NB, LR and SVM as single models, while RF, XGB and LGBM as ensemble models.

3.4.4 Evaluation Metrics

Our multi-class classification problem can be decomposed into multiple binary classification problems. To define the standard evaluation metrics for a binary problem [22, 20, 21], we first describe four possibilities as follows.

- True positive (TP): The classifier correctly predicts that an SV has a particular characteristic.
- False positive (FP): The classifier incorrectly predicts that an SV has a particular characteristic.
- True negative (TN): The classifier correctly predicts that an SV does not have a particular characteristic.
- False negative (FN): The classifier incorrectly predicts that an SV does not have a particular characteristic.

Based on TP, FP, TN, FN, *Accuracy*, *Precision*, *Recall* and *F1-Score* can be defined accordingly in (3.1), (3.2), (3.3), (3.4).

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{3.1}$$

$$Precision = \frac{TP}{TP + FP} \tag{3.2}$$

$$Recall = \frac{TP}{TP + FN} \tag{3.3}$$

$$F1 - Score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{3.4}$$

Whilst *Accuracy* measures the global performance of all classes, *F1-Score* (a harmonic mean of *Precision* and *Recall*) evaluates each class separately. Such local estimate like F1-Score is more favorable to the imbalanced VCs such as Access Vector, Access Complexity, Authentication, Severity (see Fig. 3.1). In fact, there are several variants of *F1-Score* for a multi-class classification problem, namely *Micro*, *Macro* and Weighted *F1-Scores*. In the case of multi-class classification, *Micro F1-Score* is actually the same as *Accuracy*. For *Macro* and *Weighted F1-Scores*, the former does not consider class distribution (the number of elements in each class) for computing *F1-Score* of each class; whereas, the latter does. To account for the balanced and imbalanced VCs globally and locally, we used *Accuracy*, *Macro*, and *Weighted F1-Scores* to evaluate our models. For model selection, if there was a performance tie among models regarding *Accuracy* and/or *Macro F1-Score*, *Weighted F1-Score* would be chosen as the discriminant criterion. The reason is that *Weighted F1-Score* can be considered a compromise between *Macro F1-Score* and *Accuracy*. If the tie still existed, the less complex model with the smaller number of hyperparameters would be selected as per the Occam's razor principle [252]. In the last tie scenario, the model with shorter training time would be chosen.

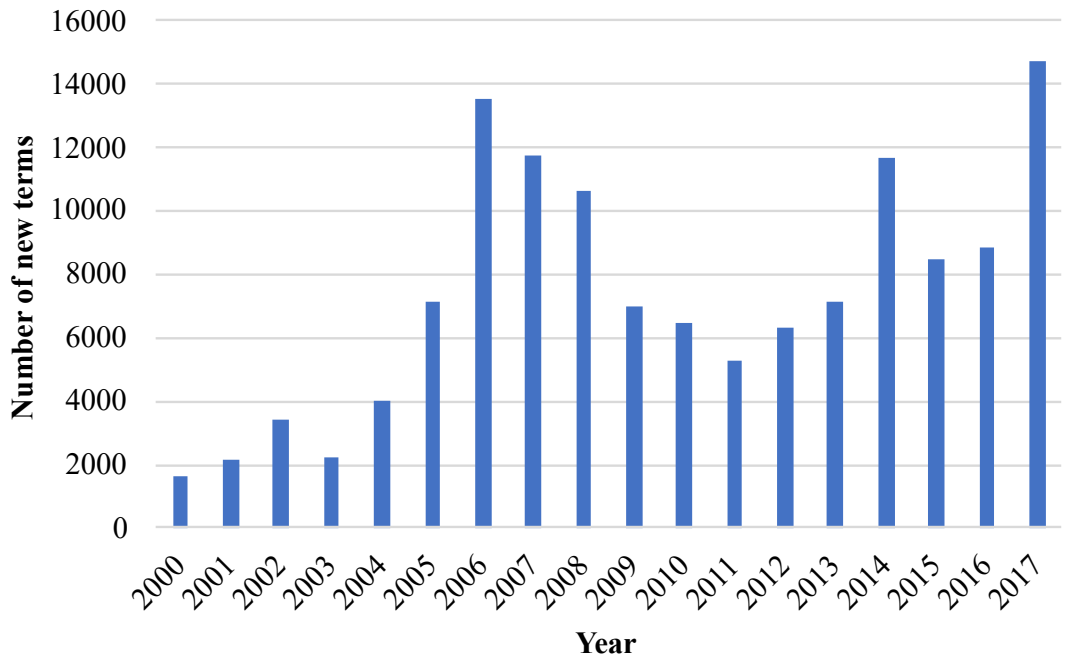


FIGURE 3.4: The number of new terms from 2000 to 2017 of SV descriptions in NVD.

3.5 Experimental Results and Discussion

3.5.1 RQ1: Is Our Time-Based Cross-Validation More Effective Than a Non-Temporal Method to Handle Concept Drift in The Model Selection Step for Report-Level SV Assessment?

We performed both qualitative and quantitative analyses to demonstrate the relationship between *concept drift* and the model selection step of report-level SV assessment. Firstly, it is intuitive that data of SVs intrinsically change over time because of new products, software and attack vectors. The number of new terms appearing in the NVD descriptions each year during the period from 2000 to 2017 is given in Fig. 3.4. On average each year, there were 7345 new terms added to the vocabulary. Moreover, from 2015 to 2017, the number of new terms had been consistently increasing and achieved an all-time high value of 14684 in 2017. We also highlight some concrete examples about the terms appearing in the database after a particular technology, product or attack was released in Fig. 3.5. There seems to be a strong correlation between the time of appearance of some new terms in the descriptions and their years of release or discovery. Such unseen terms contained many concepts about new products (e.g., *Firefox*, *Skype*, and *iPhone*), operating systems (e.g., *Android*, *Windows Vista/7/8/10*), technologies (e.g., *Ajax*, *jQuery*, and *Node.js*), attacks (e.g., *Code Red*, *Slammer*, and *Stuxnet* worms). There were also the extended forms of existing ones such as the updated versions of Java Standard Edition (Java SE) each year. These qualitative results depict that if the time property of SVs is not considered in the model selection step, then future terms can be mixed with past ones. Such information leakage can result in a discrepancy in the real-world model performance.

In fact, the main goal of the validation step is to select the optimal models that can exhibit similar behavior on unseen data. Next, our approach quantitatively compared the degree of model overfitting between our time-based cross-validation method and a stratified

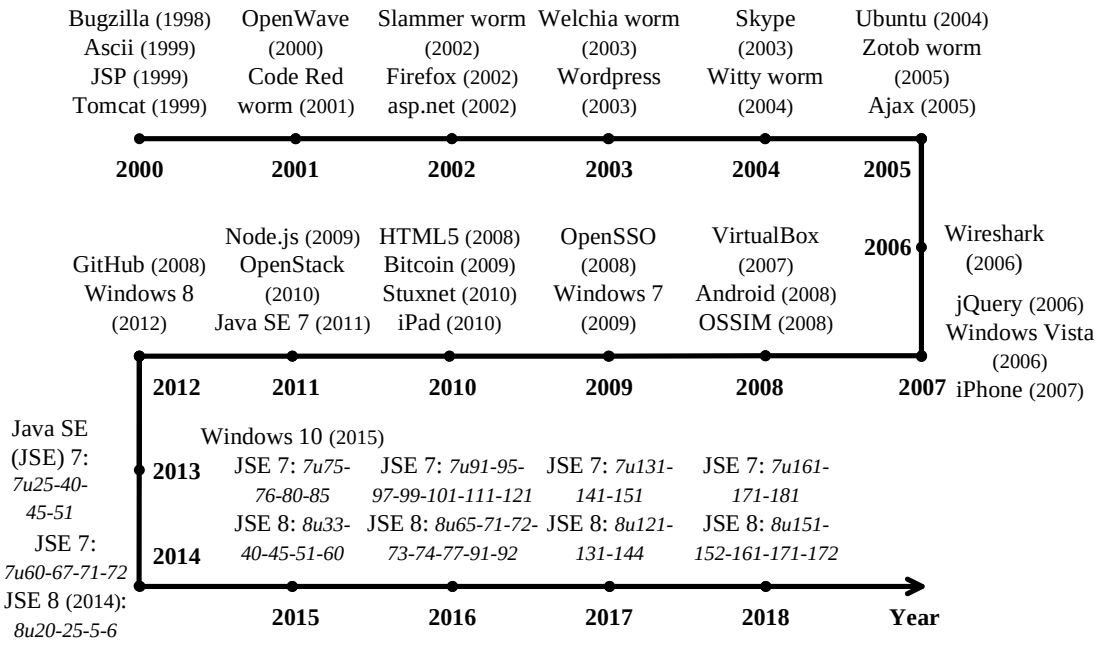


FIGURE 3.5: Examples of new terms in NVD corresponding to new products, software, cyber-attacks from 2000 to 2018. **Note**: The year of release/discovery is put in parentheses.

non-temporal one used in [22, 20]. For each method, we computed the *Weighted F1-Scores* difference between the cross-validated and testing results of the optimal models found in the validation step (see Fig. 3.6). The model selection and selection criteria procedures of the normal cross-validation method were the same as our temporal one. Fig. 3.6 shows that traditional non-temporal cross-validation was overfitted in four out of seven cases (i.e., Availability, Access Vector, Access Complexity, and Authentication). Especially, the degrees of overfitting of non-temporal validation method were 1.8, 4.7 and 1.8 times higher than those of the time-based version for Availability, Access Vector, and Access Complexity, respectively. For the other three VCs, both methods were similar, in which the differences were within 0.02. Moreover, on average, the *Weighted F1-Scores* on the testing set of the non-temporal cross-validation method were only 0.002 higher than our approach. This value is negligible compared to the difference of 0.02 (ten times more) in the validation step. It is worth noting that a similar comparison also held for non-stratified non-temporal cross-validation. Overall, both qualitative and quantitative findings suggest that the time-based cross-validation method should be preferred to lower the performance overestimation and mis-selection of report-level SV assessment models due to the effect of *concept drift* in the model selection step.

The summary answer to RQ1: The qualitative results show that many new terms are regularly added to NVD, after the release or discovery of the corresponding software products or cyber-attacks. Normal random-based evaluation methods mixing these new terms can inflate the cross-validated model performance. Quantitatively, the optimal models found by our time-based cross-validation are also less overfitted, especially two to five times for Availability, Access Vector and Access Complexity. It is recommended that the time-based cross-validation should be adopted in the model selection step for report-level SV assessment.

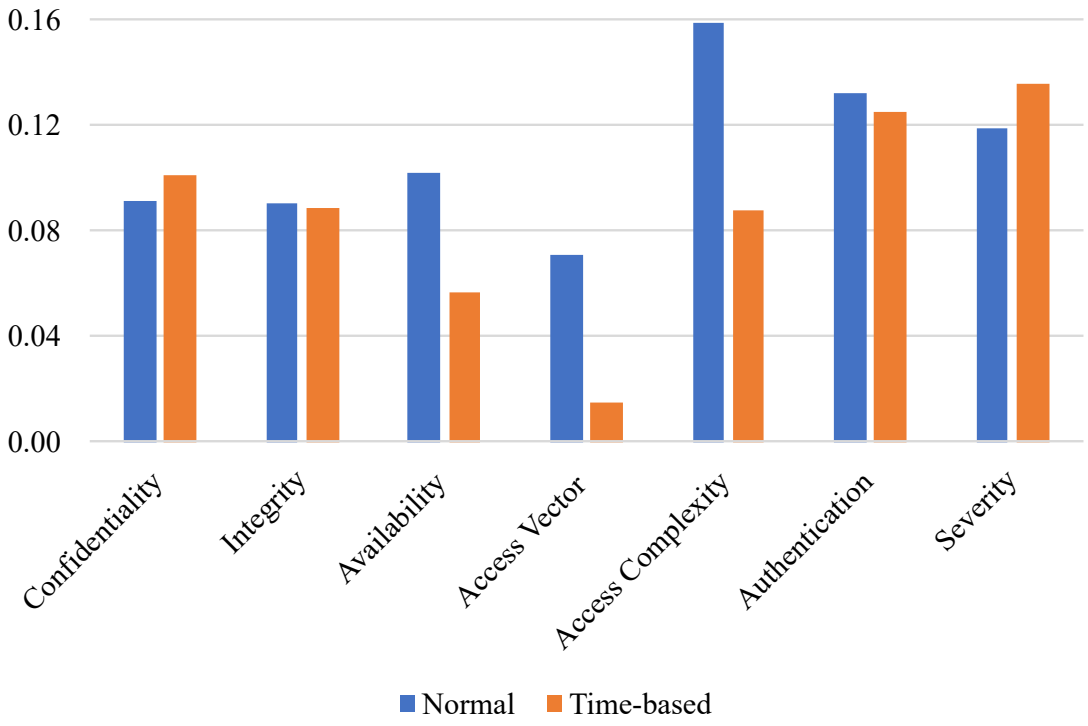


FIGURE 3.6: Performance differences between the validated and testing *Weighted F1-Scores* of our time-based validation and a normal cross-validation methods.

3.5.2 RQ2: Which are the Optimal Models for Multi-Classification of Each SV Characteristic?

The answer to RQ1 has shown that the temporal cross-validation should be used for selecting the optimal models in the context of report-level SV assessment. RQ2 presents the detailed results of the first phase of our model. Specifically, we used our five-fold time-based cross-validation to select the optimal word-only model for each of the seven VCs from six classifiers (see section 3.4.3) and eight NLP representations (see Table 3.2). We followed the guidelines of the previous work [22] to extract only the words appearing in more than 0.1% of all descriptions as features for RQ2.

Firstly, each classifier was tuned using random VCs to select its optimal set of hyperparameters. Such selected hyperparameters are reported in Table 3.3. It is worth noting that we utilized local optimization as a filter to reduce the search space. We found that 0.1 was a consistently good value of regularization coefficient for SVM. Unlike SVM, for LR, 0.1 was suitable for term frequency representation; whereas, 10 performed better for the case of tf-idf. One possible explanation is that LR provides a decision boundary that is more sensitive to hyperparameter. Additionally, although tf-idf with l2-normalization helps a model converge faster, it usually requires more regularization to avoid overfitting [253]. For ensemble models, more hyperparameters need tuning, as mentioned in section 3.4.3. Regarding the maximum number of leaves, the optimal value for RF was *unlimited*, which is expected since it would give more flexibility to the model.

However, for XGB and LGBM, the *unlimited* value was not available. In fact, the higher value did not improve the performance, but significantly increased the computational time. As a result, we chose 100 to be the number of leaves for XGB and LGBM. Similarly, we obtained 100 as a good value for the number of trees of each ensemble model. We noticed

Table 3.3: Optimal hyperparameters found for each classifier.

Classifier	Hyperparameters
NB	None
LR	Regularization value: + 0.1 for term frequency + 10 for tf-idf
SVM	Kernel: linear Regularization value: 0.1
RF	Number of trees: 100 Max. depth: unlimited Max. number of leaf nodes: unlimited
XGB	Number of trees: 100 Max. depth: unlimited Max. number of leaf nodes: 100
LGBM	Number of trees: 100 Max. depth: unlimited Max. number of leaf nodes: 100

Table 3.4: Optimal models and results after the validation step. **Note**: The NLP configuration number is put in parentheses.

SV characteristic	Classifier (config)	*Accuracy*	*Macro F1-Score*	*Weighted F1-Score*
Confidentiality	LGBM (1)	0.839	0.831	0.840
Integrity	XGB (4)	0.861	0.853	0.861
Availability	LGBM (1)	0.785	0.783	0.782
Access Vector	XGB (7)	0.936	0.643	0.914
Access Complexity	LGBM (1)	0.771	0.553	0.758
Authentication	LR (3)	0.973	0.626	0.972
Severity	LGBM (5)	0.814	0.763	0.811

that the maximum depth of ensemble methods was the hyperparameter that affected the validation result the most; the others did not change the performance dramatically. Finally, we got a search space of size of 336 in the cross-validation step ((six classifiers) $\times$ (eight NLP configurations) $\times$ (seven characteristics)). The optimal validation results after using our five-fold time-based cross-validation method in section 3.3.3 are given in Table 3.4.

Besides each output, we also examined the validated results across different types of classifiers (single vs. ensemble models) and NLP representations (n-grams and tf-idf vs. term frequency). Since the NLP representations mostly affect the classifiers, their validated results are grouped by six classifiers in Tables 3.5 and 3.6. The result shows that tf-idf did not outperform term frequency for five out of six classifiers. This result agrees with the existing work [22, 20]. It seemed that n-grams with $n > 1$ improved the result. We used a one-sided non-parametric Wilcoxon signed rank test [254] to check the significance of such improvement of n-grams ($n > 1$). The p-value was 0.169, which was larger than 0.01 (the significance level). Thus, we were unable to accept the improvement of n-grams over unigram. Furthermore, there was no performance improvement after increasing the number of n-grams. The above-reported three observations implied that the more complex NLP representations did not provide a statistically significant improvement over the simplest BoW (configuration 1 in Table 3.2). This argument helped explain why three out of seven optimal models in Table 3.4 were BoW.

Table 3.5: Average cross-validated *Weighted F-scores* of term frequency vs. tf-idf grouped by six classifiers.

	Classifier					
	NB	**LR**	**SVM**	**RF**	**XGB**	**LGBM**
Term frequency	0.781	**0.833**	**0.835**	**0.843**	**0.846**	**0.846**
tf-idf	**0.786**	0.832	0.831	0.836	0.843	0.844

Table 3.6: Average cross-validated *Weighted F-scores* of uni-gram vs. n-grams ($2 \leq n \leq 4$) grouped by six classifiers.

Classifier	**1-gram**	**2-grams**	**3-grams**	**4-grams**
NB	0.756	0.778	0.784	**0.785**
LR	0.821	0.835	**0.836**	**0.836**
SVM	0.823	0.835	0.836	**0.837**
RF	0.838	**0.840**	0.838	0.838
XGB	0.844	0.845	**0.846**	**0.846**
LGBM	**0.845**	**0.845**	**0.845**	**0.845**

Along with the NLP representations, we also investigated the performance difference between single (NB, LR, and SVM) and ensemble (RF, XGB, and LGBM) models. The average *Weighted F1-Scores* grouped by VCs for single and ensemble models are illustrated in Fig. 3.7. The ensemble models seemed to consistently demonstrate the superior performance compared to the single counterparts. We also observed that the ensemble methods produced mostly consistent results (i.e., small variances) for Access Vector and Authentication characteristics. We performed the one-sided non-parametric Wilcoxon signed rank tests [254] to check the significance of the better performance of the ensemble over the single models. Table 3.7 reports the p-values of the results from the hypothesis testing. The tests confirmed that the superiority of the ensemble methods was significant since all p-values were smaller than the significance level of 0.01. The validated results in Table 3.4 also affirmed that six out of seven optimal classifiers were ensemble (i.e., LGBM and XGB). It is noted that the XGB model usually took more time to train than the LGBM model, especially for tf-idf representation. Our findings suggest that LGBM, XGB and BoW should be considered as baseline classifiers and NLP representations for report-level SV assessment.

The summary answer to RQ2: LGBM and BoW are the most frequent optimal classifiers and NLP representations. Overall, the more complex NLP representations such as n-grams, tf-idf do not provide a statistically significant performance improvement than BoW. The ensemble models perform statistically better than single ones. It is recommended that the ensemble classifiers (e.g., XGB and LGBM) and BoW should be used as baseline models for report-level SV assessment.

3.5.3 RQ3: How Effective is Our Character-Word Model to Perform Automated Report-Level SV Assessment with Concept Drift?

The OoV terms presented in RQ1 actually directly have an impact on the word-only models. Such missing features can make a model unable to produce reliable results. Especially when no existing term is found (i.e., all features are zero), a model would have the same output regardless of the context. To answer RQ3, we first tried to identify such all-zero

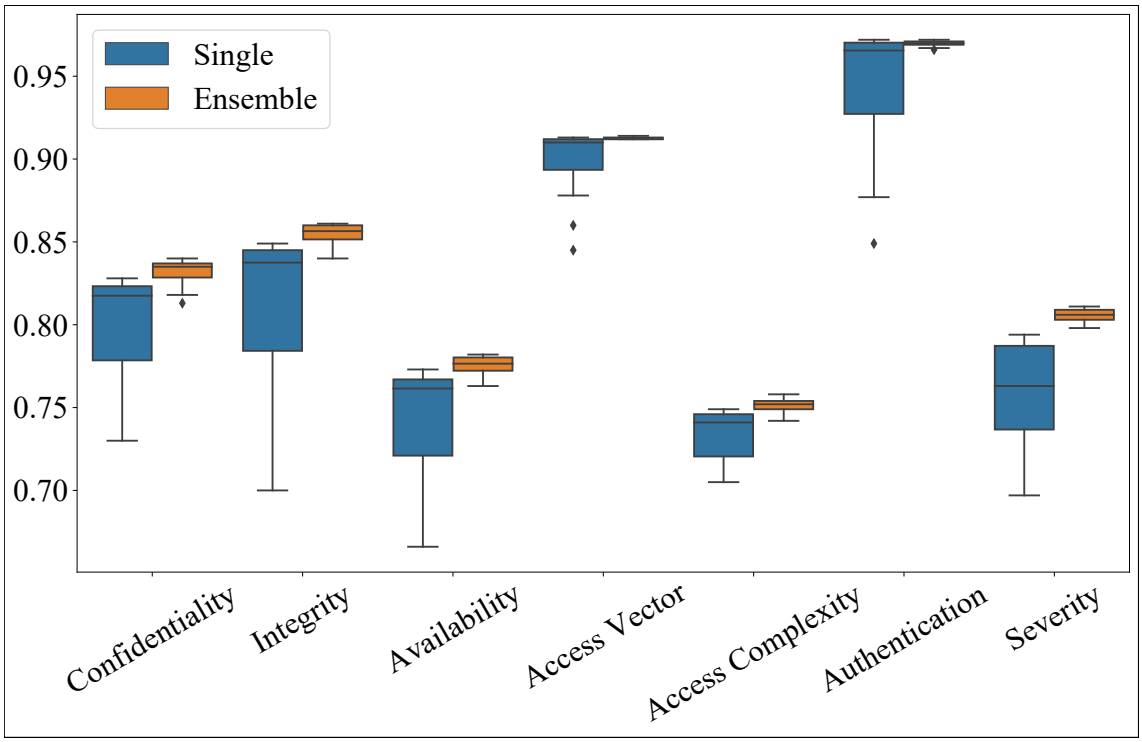


FIGURE 3.7: Average cross-validated *Weighted F1-Scores* comparison between ensemble and single models for each VC.

TABLE 3.7: P-values of H_0: Ensemble models $\leq$ Single models for each VC.

SV characteristic	p**-value**
Confidentiality	3.261×10^{-5}
Integrity	9.719×10^{-5}
Availability	3.855×10^{-5}
Access Vector	2.320×10^{-3}
Access Complexity	1.430×10^{-5}
Authentication	1.670×10^{-3}
Severity	1.060×10^{-7}

cases in the SV descriptions from 2000 to 2018. For each year from 2000 to 2018, we split the dataset into (i) training set (data from the previous year backward) for building the vocabulary, and (ii) testing set (data from the current year to 2018) for checking the vocabulary existence. We found 64 cases from 2000 to 2018 in the testing data, in which all the features were missing (see Appendix 3.9). We used the terms appearing at least 0.1% in all descriptions. It should be noted that the number of all-zero cases may be reduced using a larger vocabulary with the trade-off for larger computational time. We also investigated the descriptions of these vulnerabilities and found several interesting patterns. The average length of these abnormal descriptions was only 7.98 words compared to 39.17 of all descriptions. It turned out that the information about the threats and sources of such SVs was limited. Most of them just included the assets and attack/SV types. For example, the vulnerabilities with ID of CVE-2016-10001xx had nearly the same format "Reflected XSS in WordPress plugin" with the only differences were the name and version of the plugin. This format made it hard for a model to evaluate the impact of each SV separately. Another issue was due to specialized or abbreviated terms such

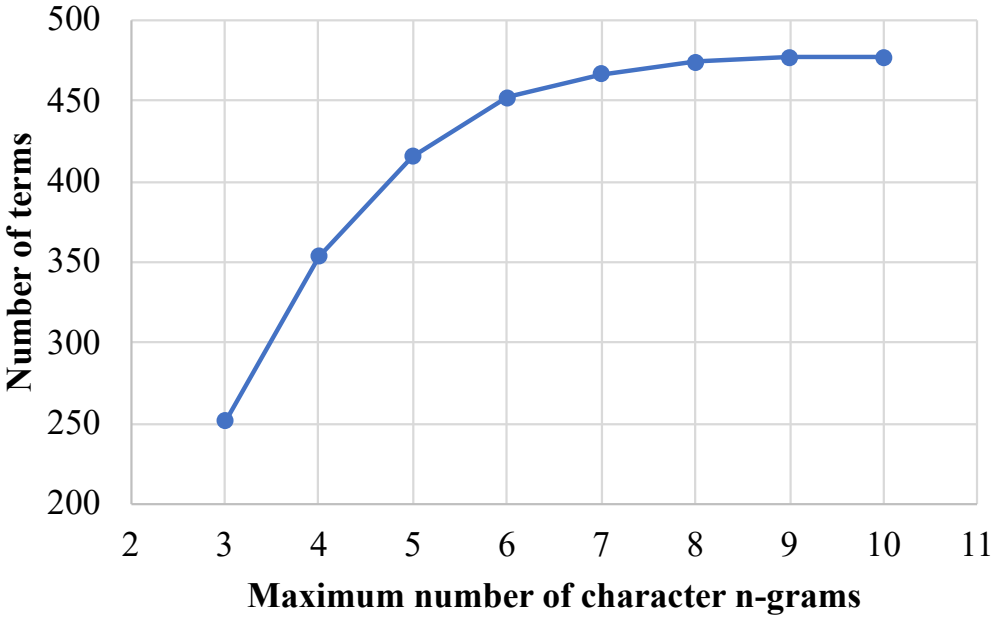


FIGURE 3.8: The relationship between the size of vocabulary and the maximum number of character n-grams.

as $/redirect?url = XSS, SEGV, CSRF$ without proper explanation. The above issues suggest that SV descriptions should be written with sufficient information to enhance the comprehensibility of SVs.

For RQ3, the solution to the issue of the word-only models using character-level features is evaluated. We considered the non-stop-words with high frequency (i.e., appearing in more than 10% of all descriptions) to generate the character features. Using the same 0.1% value as RQ2 increased the dimensions more than 30 times, but the performance only changed within 0.02. According to Algorithm 1, the output minimum number of character n-grams was chosen to be two. We first tested the robustness of the character-only models by setting the maximum number of characters to only three. For each year y from 1999 to 2017, we used such character model to generate the characters from the data of the considering year y backward. We then verified the existence of such features using the descriptions of the other part of the data (i.e., from year $y + 1$ towards 2018). Surprisingly, the model using only two-to-three-character n-grams could produce at least one non-zero feature for all the descriptions even using only training data in 1999 (i.e., the first year in our dataset based on CVE-ID). Such finding shows that our approach is stable to SV data changes (*concept drift*) in testing data from 2000 to 2018 even with the limited amount of data and without retraining.

Next, to increase the generalizability of our approach, values of 3-10 were considered for selecting the maximum number of character n-grams based on their corresponding vocabulary sizes (see Fig. 3.8). Using the elbow method in cluster analysis [255], six was selected since the vocabulary size did not increase dramatically after this point. The selected minimum and maximum values of character n-grams matched the minimum and average word lengths of all NVD descriptions in our dataset, respectively.

We then used the feature aggregation algorithm (see section 3.3.4) to create the aggregated features from the character n-grams ($2 \leq n \leq 6$) and word n-grams to build the final model set and compared it with two baselines: Word-only Model (WoM) and Character-only Model (CoM).

TABLE 3.8: Performance (*Accuracy*, *Macro F1-Score*, *Weighted F1-Score*) of our character-word vs. word-only and character-only models.

SV characteristic	Our optimal model (CWM)			Word-only model (WoM)			Character-only model (CoM)		
	Accuracy	***Macro F1-Score***	***Weighted F1-Score***	***Accuracy***	***Macro F1-Score***	***Weighted F1-Score***	***Accuracy***	***Macro F1-Score***	***Weighted F1-Score***
Confidentiality	**0.727**	**0.717**	**0.728**	0.722	0.708	0.723	0.694	0.683	0.698
Integrity	0.763	**0.749**	0.764	0.763	0.744	0.764	0.731	0.718	0.734
Availability	**0.712**	**0.711**	**0.711**	0.700	0.696	0.702	0.660	0.657	0.660
Access Vector	**0.914**	**0.540**	**0.901**	0.904	0.533	0.894	0.910	0.538	0.899
Access Complexity	0.703	0.468	0.673	**0.718**	**0.476**	**0.691**	0.700	0.457	0.668
Authentication	**0.875**	**0.442**	**0.844**	0.864	0.425	0.832	0.866	0.441	0.840
Severity	0.668	**0.575**	0.663	**0.686**	0.569	**0.675**	0.661	0.549	0.652

It should be noted that WoM is the model in which *concept drift* is not handled. Unfortunately, a direct comparison with the existing WoM [22] was not possible since they used an older NVD dataset and more importantly, they did not release their source code for reproduction. However, we tried to set up the experiments based on the guidelines and results presented in the previous paper [22].

To be more specific, we used BoW predictors and random forest (the best of their three models used) with the following hyperparameters: the number of trees was 100 and the number of features for splitting was 40. For CoM, we used the same optimal classifier of each VC. The comparison results are given in Table 3.8. CWM performed slightly better than the WoM for four out of seven VCs regarding all evaluation metrics. Also, 4.98% features of CWM were non-zero, which was nearly five-time denser than 1.03% of WoM. Also, CoM was the worst model among the three, which had been expected since it contained the least information (smallest number of features). Although CWM did not significantly outperform WoM, its main advantage is to effectively handle the OoV terms (*concept drift*), except new terms without any matching parts. We hope that our solution to *concept drift* will be integrated into practitioner's existing framework/workflow and future research work to perform more robust report-level SV assessment.

The summary answer to RQ3: The WoM does not handle the new cases well, especially those with all zero-value features. Without retraining, the tri-gram character features can still handle the OoV words effectively with no all-zero features for all testing data from 2000 to 2018. Our CWM performs comparably well with the existing WoM and provides nearly five-time richer SV information. Hence, our CWM is better for automated report-level SV assessment with *concept drift*.

3.5.4 RQ4: To What Extent Can Low-Dimensional Model Retain the Original Performance?

The n-gram NLP models usually have an issue with the high-dimensional and sparse feature vectors [240]. The large feature sizes of our CWMs in Table 3.8 were 1,649 for Confidentiality, Availability and Access Complexity; 4,154 for Integrity and Access Vector; 3,062 for Authentication; and 5,104 for Severity. To address such challenge in RQ4, we investigated a dimensionality reduction method (i.e., Latent Semantic Analysis (LSA) [245]) and recent sub-word embeddings (e.g., fastText [196, 246]) for SV assessment. fastText is an extension of Word2Vec [202] word embeddings, in which the character-level features are also considered. fastText is different to traditional n-grams in the sense that it determines the meaning of a word/subword based on the surrounding context. Here, we computed the sentence representation as an average fastText embedding of its constituent words and characters. We implemented fastText using the *Gensim* [256] library in Python.

For LSA, using the elbow method and total explained variances of the principal components, we selected 300 for the dimensions and called it LSA-300. Table 3.9 shows that LSA-300 retained from 90% to 99% performance of the original model, but used only 300 dimensions (6-18% of original model sizes). More remarkably, with the same 300 dimensions, the fastText model trained on SV descriptions was on average better than LSA-300 (97% vs. 94.5%). fastText model even slightly outperformed our original CWM for Access Complexity. Moreover, for all seven cases, the fastText model using SV knowledge (fastText-300) had higher *Weighted F1-Scores* than that trained on English Wikipedia pages (fastText-300W) [257]. The result implied that SV descriptions contain specific terms that do not frequently appear in general domains. The domain relevance turns out to be not only applicable to word embeddings [21], but also to character/sub-word

TABLE 3.9: *Weighted F1-Scores* of our original CWM (green-colored baseline), 300-dimension Latent Semantic Analysis (LSA-300), fastText trained on SV descriptions (fastText-300) and fastText trained on English Wikipedia pages (fastText-300W).

SV characteristic	Our CWM	LSA-300	fastText-300	fastText-300W
Confidentiality	**0.728**	0.656	**0.679**	0.648
Integrity	**0.764**	0.695	**0.719**	0.672
Availability	**0.711**	0.656	**0.687**	0.669
Access Vector	**0.901**	0.892	**0.893**	0.866
Access Complexity	**0.673**	0.611	**0.679**	0.678
Authentication	**0.844**	**0.842**	0.815	0.765
Severity	**0.663**	**0.656**	0.654	0.635

embeddings for SV analysis and assessment. Overall, our findings show that LSA and fastText are capable of building efficient report-level SV assessment models without too much performance trade-off.

The summary answer to RQ4: The LSA model with 300 dimensions (6-18% of the original size) retains from 90% up to 99% performance of the original model. With the same feature dimensions, the model with fastText sub-word embeddings provide even more promising results. The fastText model with the SV knowledge outperforms that trained on a general context (e.g., Wikipedia). LSA and fastText can help build efficient models for report-level SV assessment.

3.6 Threats to Validity

Internal validity. We used well-known tools such as the *scikit-learn* [241] and *nltk* [242] libraries for ML and NLP. Our optimal models may not guarantee the highest performance for every SV characteristic since there are infinite values of hyperparameters to tune. However, even when the optimal values change, a time-based cross-validation method should still be preferred since we considered the general trend of all SVs. Our models may not provide the state-of-the-art results, but at least they give the baseline performance for handling the *concept drift* of SVs.

External validity. Our work used NVD – one of the most comprehensive public repositories of SVs. The size of our dataset is more than 100,000 with the latest SVs in 2018. Our character-word model was demonstrated to consistently handle the OoV words well even with very limited data for all years in the dataset. It is recognized that the model may not work for extreme rare terms in which no existing parts can be found. However, our model is totally re-trainable to deal with such cases or incorporate more sources of SVs' descriptions.

Conclusion validity. We mitigated the randomness of the results by taking the average value of five-fold cross-validation. The performance comparison of different types of classifiers and NLP representations was also confirmed using statistical hypothesis tests with p-values that were much lower than the significance level of 1%.

3.7 Related Work

3.7.1 Report-Level SV Assessment

It is important to patch critical-first SVs [258]. Besides CVSS, there have been many other assessment schemes for SVs [259, 136, 260]. Recently, there has been a detailed Bayesian analysis of various SV scoring systems [261], which highlights the good overall performance of CVSS. Therefore, we used the well-known CVSS as the ground truth for our approach. We assert that our approach can be generalizable to other SV assessment systems following the same scheme of multi-class classification.

As mentioned in Chapter 2, Bozorgi et al. [32] pioneered the use of ML models for report-level SV assessment. Their paper used an SVM model and various features (e.g., NVD description, CVSS, published and modified dates) to estimate the likelihood of exploitation and *time-to-exploit* of SVs. Another piece of work analyzed the VCs and trends of SVs by incorporating different SV information from multiple repositories [152, 49], security advisories [47, 262], darkweb/deepnet [49, 77] and social network (Twitter) [33]. These efforts assumed that all VCs have been available at the time of analysis. However, our work relaxes this assumption by using only SV descriptions – one of the first pieces of information appearing in new SV reports. As a result, our model can be used for both new and old SVs.

Many studies have built upon the work of Bozorgi et al. [32] and used NVD descriptions for report-level SV assessment, as reviewed in Chapter 2. Here, we focus on several studies directly related to our current study. Yamamoto et al. [95] used Linear Discriminant Analysis, Naïve Bayes and Latent Semantic Indexing combined with an annual effect estimation to determine the CVSS-based VCs of more than 60,000 SVs in NVD. The annual effect focused on recent SVs, but still could not explicitly handle OoV terms in SV descriptions. Spanos et al. [22] worked on the same task as ours using a multi-target framework with Decision Tree, Random Forest and Gradient Boosting Tree. Note that our approach also contains the word-only model, but we select the optimal models using our time-based cross-validation to better address the *concept drift* issue. SV descriptions were also used to evaluate SV severity [20], associate the frequent terms with each VC [97], determine the type of each SV using topic modeling [156] and show SV trends [152]. Recently, Han et al. [21] have applied deep learning to predict SV severity. The existing literature has demonstrated the usefulness of description for SV analysis and assessment, but has not mentioned how to overcome its *concept drift* challenge. Our work is the first of its kind to provide a robust treatment for SVs' *concept drift*.

3.7.2 Temporal Modeling of SVs

Regarding the temporal relationship of SVs, Roumani et al. [263] proposed a time-series approach using autoregressive integrated moving average and exponential smoothing methods to predict the number of SVs in the future. Another time-series work [264] was presented to model the trend in disclosing SVs. A group of researchers published a series of studies [265, 266, 267] on stochastic models such as Hidden Markov Models, Artificial Neural Network, and Support Vector Machine to estimate the occurrence and exploitability of SVs. The focus of the above studies was on the determination of the occurrence of SVs over time. In contrast, our work aims to handle the temporal relationship to build more robust predictive (multi-class classification) models for report-level SV assessment.

3.8 Chapter Summary

We observe that the existing studies suffer from *concept drift* in SV descriptions that affect both the traditional model selection and prediction steps of report-level SV assessment. We assert that *concept drift* can degrade the robustness of these existing predictive models. We showed that time-based cross-validation should be used for SV analysis to better capture the temporal relationship of SVs. Then, our main contribution is the Character-Word Models (CWMs) to improve the robustness of automated report-level SV assessment with *concept drift*. CWMs were demonstrated to handle *concept drift* of SVs effectively for all the testing data from 2000 to 2018 in NVD even in the case of data scarcity. Our approach also performed comparably well with the existing word-only models. Our CWMs were also much less sparse and thus less prone to overfitting. We also found that Latent Semantic Analysis and sub-word embeddings like fastText help build compact and efficient CWM models (up to 94% reduction in dimension) with the ability to retain at least 90% of the predictive performance for all VCs. Besides the strong performance, we also provide implications on the use of different features and classifiers for building effective report-level SV assessment models.

3.9 Appendix - SVs with All Out-of-Vocabulary Words

64 vulnerabilities along with their CVD-IDs that had all-zero features of word-only model from 2000 to 2018, as mentioned in section 3.5.3: CVE-2013-6647, CVE-2015-1000004, CVE-2016-1000113, CVE-2016-1000114, CVE-2016-1000117, CVE-2016-1000118, CVE-2016-1000126, CVE-2016-1000127, CVE-2016-1000128, CVE-2016-1000129, CVE-2016-1000130, CVE-2016-1000131, CVE-2016-1000132, CVE-2016-1000133, CVE-2016-1000134, CVE-2016-1000135, CVE-2016-1000136, CVE-2016-1000137, CVE-2016-1000138, CVE-2016-1000139, CVE-2016-1000140, CVE-2016-1000141, CVE-2016-1000142, CVE-2016-1000143, CVE-2016-1000144, CVE-2016-1000145, CVE-2016-1000146, CVE-2016-1000147, CVE-2016-1000148, CVE-2016-1000149, CVE-2016-1000150, CVE-2016-1000151, CVE-2016-1000152, CVE-2016-1000153, CVE-2016-1000154, CVE-2016-1000155, CVE-2016-1000217, CVE-2017-10798, CVE-2017-10801, CVE-2017-14036, CVE-2017-14536, CVE-2017-15808, CVE-2017-16760, CVE-2017-16785, CVE-2017-17499, CVE-2017-17703, CVE-2017-17774, CVE-2017-6102, CVE-2017-7276, CVE-2017-8783, CVE-2018-10030, CVE-2018-10031, CVE-2018-10382, CVE-2018-11120, CVE-2018-11405, CVE-2018-12501, CVE-2018-13997, CVE-2018-14382, CVE-2018-5285, CVE-2018-5361, CVE-2018-6467, CVE-2018-6834, CVE-2018-8817, CVE-2018-9130

Chapter 4

Automated Function-Level Software Vulnerability Assessment

Related publication: This chapter is based on our paper titled "*On the Use of Fine-grained Vulnerable Code Statements for Software Vulnerability Assessment Models*", published in the 19th International Conference on Mining Software Repositories (MSR), 2022 (CORE A) [268].

The proposed approach in Chapter 3 has improved the robustness of report-level Software Vulnerability (SV) assessment models against changing data of SVs in the wild. Nevertheless, these report-level models still rely on SV descriptions that mostly require significant expertise and manual effort to create, which may cause delays for SV fixing. On the other hand, respective vulnerable code is always available before SVs are fixed. Many studies have developed Machine Learning (ML) approaches to detect SVs in functions and fine-grained code statements that cause such SVs. However, as shown in Chapter 2, there is little work on leveraging such detection outputs for data-driven SV assessment to give information about exploitability, impact, and severity of SVs. The information is important to understand SVs and prioritize their fixing. Using large-scale data from 1,782 functions of 429 SVs in 200 real-world projects, in Chapter 4, we investigate ML models for automating function-level SV assessment tasks, i.e., predicting seven Common Vulnerability Scoring System (CVSS) metrics. We particularly study the value and use of vulnerable statements as inputs for developing the assessment models because SVs in functions are originated in these statements. We show that vulnerable statements are 5.8 times smaller in size, yet exhibit 7.5-114.5% stronger assessment performance (Matthews Correlation Coefficient (MCC)) than non-vulnerable statements. Incorporating context of vulnerable statements further increases the performance by up to 8.9% (0.64 MCC and 0.75 F1-Score). Overall, we provide the initial yet promising ML-based baselines for function-level SV assessment, paving the way for further research in this direction.

4.1 Introduction

As shown in Chapter 2, previous studies (e.g., [95, 21, 22, 23, 11]) have mostly used SV reports to develop data-driven models for assigning the Common Vulnerability Scoring System (CVSS) [29] metrics to Software Vulnerabilities (SVs). Among sources of SV reports, National Vulnerability Database (NVD) [16] has been most commonly used for building SV assessment models [11]. The popularity of NVD is mainly because it has SV-specific information (e.g., CVSS metrics) and less noise in SV descriptions than other Issue Tracking Systems (ITSs) like Bugzilla [269]. The discrepancy is because NVD reports are vetted by security experts, while ITS reports may be contributed by users/developers with limited security knowledge [270]. However, NVD reports are mostly released long after SVs have been fixed. Our analysis revealed that less than 3% of the SV reports with the CVSS metrics on NVD had been published before SVs were fixed; on average, these reports appeared 146 days later than the fixes. Note that our findings accord with the previous studies [215, 216]. This delay renders the CVSS metrics required for SV assessment unavailable at fixing time, limiting the adoption of report-level SV assessment for understanding SVs and prioritizing their fixes.

Instead of using SV reports, an alternative and more straight-forward way is to directly take (vulnerable) code as input to enable SV assessment prior to fixing. Once a code function is confirmed vulnerable, SV assessment models can assign it the CVSS metrics before the vulnerable code gets fixed, even when its report is not (yet) available. Note that it is non-trivial to use static application security testing tools to automatically create bug/SV reports from vulnerable functions for current SV assessment techniques as these tools often have too many false positives [271, 272]. To develop function-level assessment models, it is important to obtain input information about SVs in functions detected by manual debugging or automatic means like data-driven approaches (e.g., [273, 274, 275]). Notably, recent studies (e.g., [276, 277]) have shown that an SV in a function usually stems from a very small number of code statements/lines, namely *vulnerable statements*. Intuitively, these vulnerable statements potentially provide highly relevant information (e.g., causes) for SV assessment models. Nevertheless, a large number of other (non-vulnerable) lines in functions, though do not directly contribute to SVs, can still be useful for SV assessment, e.g., indicating the impacts of an SV on nearby code. It still remains largely unknown about function-level SV assessment models as well as the extent to which vulnerable and non-vulnerable statements are useful as inputs for these models.

We conduct a large-scale study to fill this research gap. We investigate the usefulness of integrating fine-grained vulnerable statements and different types of code context (relevant/surrounding code) into learning-based SV assessment models. The assessment models employ various feature extraction methods and Machine Learning (ML) classifiers to predict the seven CVSS metrics (Access Vector, Access Complexity, Authentication, Confidentiality, Integrity, Availability, and Severity) for SVs in code functions.

Using 1,782 functions from 429 SVs of 200 real-world projects, we evaluate the use of vulnerable statements and other lines in functions for developing SV assessment models. Despite being up to 5.8 times smaller in size (lines of code), vulnerable statements are more effective for function-level SV assessment, i.e., 7.4-114.5% higher Matthews Correlation Coefficient (MCC) and 5.5-43.6% stronger F1-Score, than non-vulnerable lines. Moreover, vulnerable statements with context perform better than vulnerable lines alone. Particularly, using vulnerable and all the other lines in each function achieves the best performance of 0.64 MCC (8.9% better) and 0.75 F1-Score (8.5% better) compared to using only vulnerable statements. We obtain such improvements when combining vulnerable statements and context as a single input based on their code order, as well as when treating them as two separate inputs. Having two inputs explicitly provides models with

```
1  protected String getExecutionPreamble()
2  {
3      if (getWorkingDirectoryAsString() == null)
4      {return null;}
5      String dir = getWorkingDirectoryAsString();
6      StringBuilder sb = new StringBuilder();
7      sb.append("cd");
8  -   sb.append(unifyQuotes(dir));
9  +   sb.append(quoteOneItem(dir, false));
10     sb.append("&&");
11     return sb.toString();
12 }
```

FIGURE 4.1: A vulnerable function extracted from the fixing commit *b38a1b3* of an SV (CVE-2017-1000487) in the *Plexus-utils* project. **Notes**: Line 8 is vulnerable. Deleted and added lines are highlighted in red and green, respectively. Blue-colored code lines affect or are affected by line 8 directly.

the location of vulnerable statements and context for the assessment tasks, while single input does not. Surprisingly, we do not obtain any significant improvement of the double-input models over the single-input counterparts. These results show that function-level SV assessment models can still be effective even without knowing exactly which statements are vulnerable. Overall, our findings can inform the practice of building function-level SV assessment models.

Our key **contributions** are summarized as follows:

1. To the best of our knowledge, we are the first to leverage data-driven models for automating function-level SV assessment tasks that enable SV prioritization/planning prior to fixing.
2. We study the value of using fine-grained vulnerable statements in functions for building SV assessment models.
3. We empirically show the necessity and potential techniques of incorporating context of vulnerable statements to improve the assessment performance.
4. We release our datasets and models for future research at `https://github.com/lhmtriet/Function-level-Vulnerability-Assessment`.

Chapter organization. Section 4.2 gives a background on function-level SV assessment. Section 4.3 introduces and motivates the three RQs. Section 4.4 describes the methods used for answering these RQs. Section 4.5 presents our empirical results. Section 4.6 discusses the findings and threats to validity. Section 4.7 mentions the related work. Section 4.8 concludes the study and suggests future directions.

4.2 Background and Motivation

There have been a growing number of studies to detect vulnerable statements in code functions (e.g., [278, 276, 277]). Fine-grained detection assumes that not all statements in a function are vulnerable. We confirmed this assumption; i.e., only 14.7% of the lines in our curated functions were vulnerable (see section 4.4.1). However, it is non-trivial to manually annotate a sufficiently large dataset of vulnerable functions and statements for training SV prediction models. Instead, many studies (e.g., [276, 279, 280]) have automatically obtained vulnerable statements from modified lines in Vulnerability Fixing

Commits (VFCs) as these lines are presumably removed to fix SVs. The functions containing such identified statements are considered vulnerable. Note that VFCs are used as they can be relatively easy to retrieve from various sources like National Vulnerability Database (NVD) [16]. An exemplary function and its vulnerable statement are in Fig. 4.1. Line 8 "`sb.append(unifyQuotes(dir));`" is the vulnerable statement; this line was replaced with a non-vulnerable counterpart "`sb.append(quoteOneItem(dir, false));`" in the VFC. The replacement was made to properly sanitize the input (`dir`), preventing OS command injection.

Despite active research in SV detection, there is little work on utilizing the output of such detection for SV assessment. Previous studies (e.g., [95, 20, 22, 23, 11]) have mostly leveraged SV reports, mainly on NVD, to develop SV assessment models that alleviate the need for manually defining complex rules for assessing ever-increasing SVs. However, these SV reports usually appear long after SV fixing time. For example, the SV fix in Fig. 4.1 was done 1,533 days before the date it was reported on NVD. In fact, such a delay, i.e., disclosing SVs after they are fixed, is a recommended practice so that attackers cannot exploit unpatched SVs to compromise systems [281]. One may argue that internal bug/SV reports in Issue Tracking Systems (ITS) such as JIRA [282] or Bugzilla [269] can be released before SV fixing and have severity levels. However, ITS severity levels are often for all bug types, not only SVs. These ITSs also do not readily provide exploitability and impact metrics like CVSS [29] for SVs, limiting assessment information required for fixing prioritization. Moreover, SVs are mostly rooted in source code; thus, it is natural to perform code-based SV assessment. We propose predicting seven base CVSS metrics (i.e., Access Vector, Access Complexity, Authentication, Confidentiality, Integrity, Availability, and Severity)[1] after SVs are detected in code functions to enable thorough and prior-fixing SV assessment. We do not perform SV assessment for individual lines as for a given function, like Li et al. [283], we observed that there can be more than one vulnerable line and nearly all these lines are strongly related and contribute to the same SV (having the same CVSS metrics).

Vulnerable statements represent the core parts of SVs, but we posit that other (non-vulnerable) parts of a function may also be usable for SV assessment. Specifically, non-vulnerable statements in a vulnerable function are either *directly* or *indirectly* related to the current SV. We use program slicing [284] to define directly SV-related statements as the lines affect or are affected by the variables in vulnerable statements. For example, the blue lines in Fig. 4.1 are directly related to the SV as they define, change, or use the `sb` and `dir` variables in vulnerable line 8. These SV-related statements can reveal the context/usage of affected variables for analyzing SV exploitability, impact, and severity. For instance, lines 5-6 denote that `dir` is a directory and `sb` is a string (`StringBuilder` object), respectively; line 7 then indicates that a directory change is performed, i.e., the `cd` command. This sequence of statements suggests that `sb` contains a command changing directory. Line 11 returns the vulnerable command, probably affecting other components. Besides, indirectly SV-related statements, e.g., the black lines in Fig. 4.1, are remaining lines in a function excluding vulnerable and directly SV-related statements. These indirectly SV-related lines may still provide information about SVs. For example, lines 3-4 in Fig. 4.1 imply that there is only a `null` checking for directory without imposing any privilege requirement to perform the command, potentially reducing the complexity of exploiting the SV. It remains unclear to what extent different types of statements are useful for SV assessment tasks. Therefore, this study aims to unveil the contributions of these statement types to function-level SV assessment models.

[1]These metrics are from CVSS version 2 and were selected based on the same reasons presented in the study in Chapter 3. More details can be found in section 3.2.

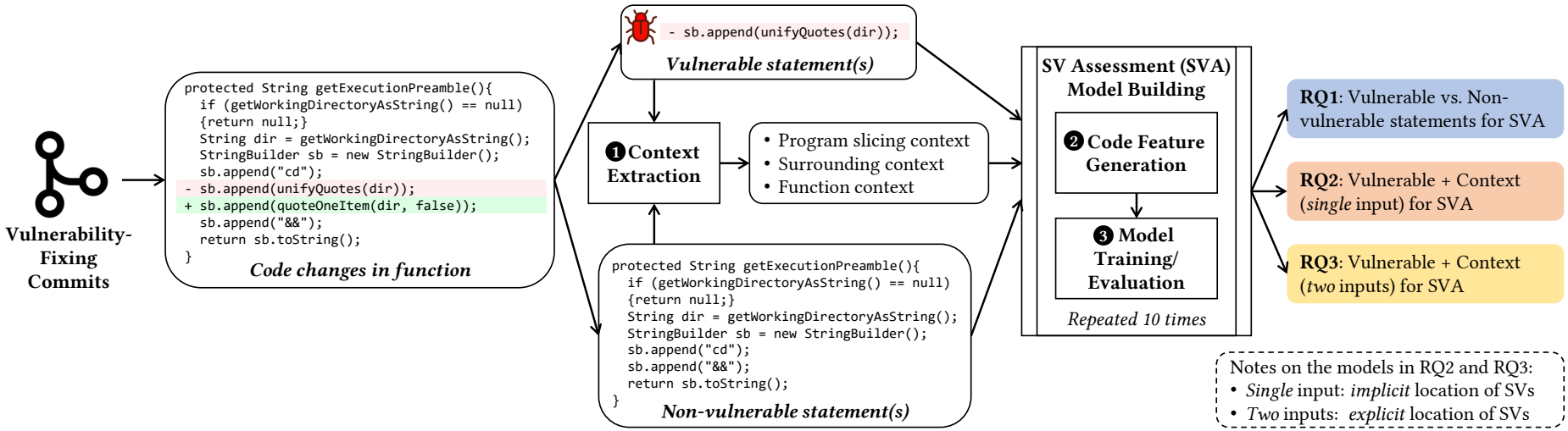


FIGURE 4.2: Methodology used to answer the research questions. **Note**: The vulnerable function is the one described in Fig. 4.1.

4.3 Research Questions

To demystify the predictive performance of SV assessment models using vulnerable and other statements in code functions, we set out to investigate the following three Research Questions (RQs).

RQ1: Are vulnerable code statements more useful than non-vulnerable counterparts for SV assessment models? Since vulnerable and non-vulnerable statements are both potentially useful for SV assessment (see section 4.2), RQ1 compares them for building function-level SV assessment models. RQ1 tests the hypothesis that vulnerable statements directly causing SVs would provide an advantage in SV assessment performance. The findings of RQ1 would also inform the practice of leveraging recent advances in fine-grained SV detection for function-level SV assessment.

RQ2: To what extent do different types of context of vulnerable statements contribute to SV assessment performance? RQ2 studies the impact of using directly and indirectly SV-related statements as context for vulnerable statements, as discussed in section 4.2, on the performance of SV assessment in functions. We compare the performance of models using different types of context lines (see section 4.4.2) that have been commonly used in the literature. RQ2 findings would unveil what types of context (if any) would be beneficial to use alongside vulnerable statements for developing function-level SV assessment models.

RQ3: Does separating vulnerable statements and context to provide explicit location of SVs improve assessment performance? For SV assessment, RQ2 combines vulnerable statements and their context as a single input following their order in functions, while RQ3 treats these two types of statements as two separate inputs. Separate inputs explicitly specify which statements are vulnerable in each function for assessment models. RQ3 results would give insights into the usefulness of the exact location of vulnerable statements for function-level SV assessment models.

4.4 Research Methodology

This section presents the experimental setup we used to perform a large-scale study on function-level SV assessment to support prioritization of SVs before fixing. We used a computing cluster with 16 CPU cores and 16GB of RAM to conduct all the experiments.

Workflow overview. Fig. 4.2 presents the workflow we followed to develop function-level SV assessment models based on various types of code inputs. The workflow has three main steps: (*i*) Collection of vulnerable and non-vulnerable statements from Vulnerability-Fixing Commits (VFCs) (section 4.4.1), (*ii*) Context extraction of vulnerable statements (section 4.4.2), and (*iii*) Model building for SV assessment (sections 4.4.3, 4.4.4, and 4.4.5). We start with VFCs containing code changes used to fix SVs. As discussed in section 4.2, we consider the deleted (–) lines in each function of VFCs as vulnerable statements, while the remaining lines are non-vulnerable statements. Details of the extracted VFCs and statements are given in section 4.4.1. Both vulnerable and non-vulnerable statements are used by the *Context Extraction* module (see section 4.4.2) to obtain the three types of context with respect to vulnerable statements that potentially provide additional information for SV assessment. The extracted statements along with their context enter the *Model Building* module. The first step in this module is to extract fixed-length feature vectors from code inputs/statements (see section 4.4.3). Subsequently, such feature vectors are used to train different data-driven models (see section 4.4.4) to support automated SV assessment, i.e., predicting the seven CVSS metrics: Access Vector, Access Complexity, Authentication, Confidentiality, Integrity, Availability, and Severity. The model training and evaluation are repeated 10 times to increase the stability of results (see section 4.4.5).

RQ-wise method. The methods to collect data, extract features as well as develop and evaluate models in Fig. 4.2 were utilized for answering all the Research Questions (RQs) in section 4.3. **RQ1** developed and compared two types of SV assessment models, namely models using only vulnerable statements and those using only non-vulnerable statements. In **RQ2**, for each of the program slicing, surrounding, and function context types, we created a single feature vector by combining the current context and corresponding vulnerable statements, based on their appearance order in the original functions, for model building and performance comparison. In **RQ3**, for each context type in RQ2, we extracted two separate feature vectors, one from vulnerable statements and another one from the context, and then fed these vectors into SV assessment models. We compared the two-input approach in RQ3 with the single-input counterpart in RQ2.

4.4.1 Data Collection

To develop SV assessment models, we need a large dataset of vulnerable functions and statements curated from VFCs, as discussed in section 4.2. This section describes the collection of such dataset.

VFC identification. We first scraped VFCs from three popular sources in the literature: NVD [16], GitHub Advisory Database,[2] and VulasDB [285], a manually curated VFC dataset. The VFCs had dates ranging from July 2000 to September 2021. We only selected VFCs that had the CVSS version 2 metrics as we needed these metrics for SV assessment. Following the recommendation of [286], we removed any VFCs that had more than 100 files and 10,000 lines of code as these VFCs are likely tangled commits, i.e., not only fixing SVs. We also discarded VFCs that were not written in the Java programming language as Java has been commonly used in both practice[3] and the literature (e.g., [287, 288, 286, 289]). After the filtering process, we obtained 900 VFCs to extract vulnerable functions/statements for building SV assessment models.

Extraction of vulnerable functions and statements. For each VFC, we obtained all the affected files (i.e., containing changed lines), excluding test files because we focused on production code. We followed a common practice [276, 279, 280] to consider all the functions in each affected file as *vulnerable functions* and the deleted lines in these functions as *vulnerable statements*. We removed functions having only added changes and non-functional/cosmetic changes such as removing/changing inline/multi-line comments, spaces, or empty lines. For the former, added lines only exist in fixed code, making it hard to pinpoint the exact vulnerable statements or root causes leading to such code additions [290]. For the latter, cosmetic changes likely do not contribute to SV fixes [286]. We also did not use a function if its entire body was deleted because such a case did not have any non-vulnerable statements for building SV assessment models in our RQs (see section 4.3). After the filtering steps, we retrieved **1,782 vulnerable functions** and **5,179 vulnerable statements** of 429 SVs in 200 Java projects. We also obtained the seven CVSS metrics from NVD for each vulnerable function (see Fig. 4.3).

Manual validation of vulnerable functions. We randomly selected 317 functions, i.e., with 95% confidence level and 5% error [291], from our dataset. The author of this thesis and a PhD student with three-year experience in Software Engineering and Cybersecurity independently validated the functions. The manual validation was considerably labor-intensive, taking 120 man-hours. We achieved a substantial agreement with a Cohen's kappa score [292] of 0.72. Disagreements were resolved through discussion. We found that 9% of the selected functions were not vulnerable, mainly due to tangled fixes/VFCs. The functions in these VFCs fixed a non-SV related issue, e.g., the

[2] https://github.com/advisories
[3] https://bit.ly/stack-overflow-survey-2021

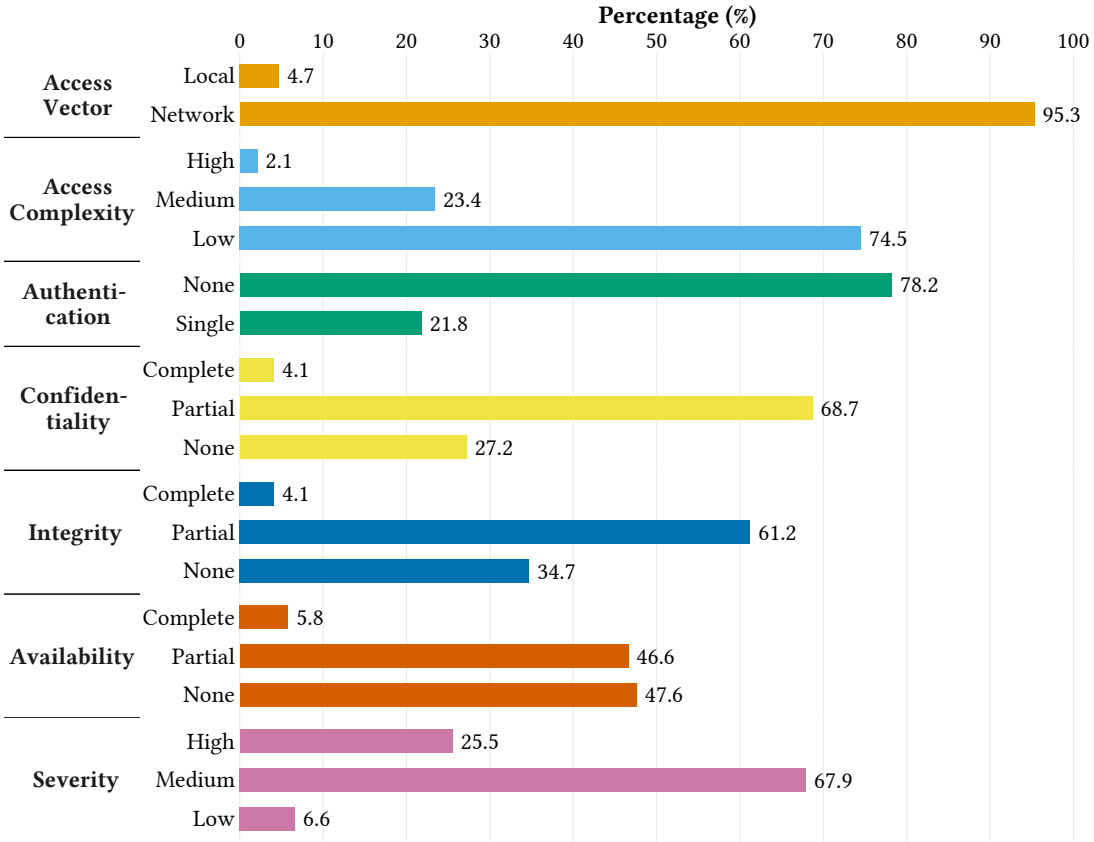


FIGURE 4.3: Class distributions of the seven CVSS metrics.

`nameContainsForbiddenSequence` function in the commit *cefbb94* of the *Eclipse Mojarra* project. The modifier of this function in the `ResourceManager.java` class was changed from `private` to `package`. This change allowed the reuse of the function in other classes like `ClasspathResourceHelper.java` for sanitizing inputs to prevent a path traversal SV (CVE-2020-6950). We assert that it is challenging to detect all of these cases without manual validation. However, such validation is extremely expensive to scale up to thousands of functions like the dataset we curated in this study.

4.4.2 Vulnerable Code Context Extraction

This section describes the *Context Extraction* module that takes vulnerable and non-vulnerable statements as inputs and then outputs program slicing, surrounding, or function context. These context types have been previously used for bug/SV-related tasks [283, 293, 275]. However, there is little known about their use and value for function-level SV assessment tasks, which are studied in this work.

Program slicing context. Program slicing captures relevant statements to a point in a program (line of code) to support software debugging [284]. This concept has been utilized for SV identification [283, 294]. However, using this context for SV assessment is fundamentally different. For SV detection, the location of vulnerable statements is unknown, so program slicing context is usually extracted for all statements including non-vulnerable ones in a function of pre-defined types (e.g., array manipulations, arithmetic operations, and function calls) [283]. In contrast, for SV assessment, vulnerable statements are known; thus, program slicing context is only obtained for these statements. With a

focus on function-level SV assessment, we considered *intra-procedural program slicing*, i.e., finding relevant lines within the boundary of a function of interest.

Following the common practice in the literature [295, 283], we used Program Dependence Graph (PDG) [296] extracted from source code to obtain program slices for vulnerable statements in each function. A PDG contains nodes that represent code statements and directed edges that capture data or control dependencies among nodes. A data dependency exists between two statements when one statement affects/changes the value of a variable used in another statement. For example, "`int b = a + 1;`" is data-dependent on "`int a = 1;`" as the variable `a` defined in the second statement is used in the first statement. A control dependency occurs when a statement determines whether/how often another statement is executed. For instance, in "`if (b == 2) func(c);`", "`func(c)`" only runs if "`b == 2`", and thus is control-dependent on the former.

Based on data and control dependencies, backward and forward slices are extracted. *Backward slices* directly change or control the execution of statements affecting the values of variables in vulnerable statements; whereas, *forward slices* are data/control-dependent on vulnerable statements [297]. In a PDG, backward slices are nodes that can go to vulnerable nodes through one or more directed edges. In Fig. 4.1, the `dir` variable is defined in line 5 and then used in vulnerable line 8, so line 5 is a backward slice. Forward slices are the nodes that can be reached from vulnerable nodes by following one or more directed edges in a PDG. In Fig. 4.1, line 11 is data-dependent on vulnerable line 8 as it uses the value of `sb`; thus, line 11 is a forward slice. The program slicing context of vulnerable statements in a function is a combination of all backward and forward slices.

Surrounding context. Another way to define context is to take a fixed number of lines (n) before and after a vulnerable statement, which is referred to as *surrounding context*. These surrounding lines may contain relevant information, e.g., values/usage of variables in vulnerable statements. This context is also based on an observation that developers usually first look at nearby code of vulnerable statements to understand how to fix SVs [293]. We discarded surrounding lines that were just code comments or blank lines as these probably do not contribute to the functionality of a function [286]. We also limited surrounding lines to be within-function.

Function context. Contrary to program slicing and surrounding context that may not use all lines in a function, *function context* uses all function statements, excluding vulnerable ones. This scope has been commonly used for SV detection models [273, 275] because vulnerable statements are unavailable at detection time. This scope contains all the lines of program slicing/surrounding context and other presumably indirectly related lines to vulnerable statements. Accordingly, the performance of using indirectly SV-related lines together with directly SV-relevant lines for SV assessment can be examined. Note that for a given function, combining function context with vulnerable statements as a single code block (RQ2 in section 4.3) is equivalent to using the whole function, which would result in the same input to SV assessment models regardless of which statements are vulnerable. This input combination allows us to evaluate the usefulness of the exact location of vulnerable statements for function-level SV assessment models.

4.4.3 Code Feature Generation

Raw code from vulnerable statements and their context are converted into fixed-length feature vectors to be consumable by learning-based SV assessment models. This step describes five techniques we used to extract features from code inputs.

Bag-of-Tokens. *Bag-of-Tokens* is based on Bag-of-Words, a popular feature extraction technique in Natural Language Processing (NLP). This technique has been commonly

investigated for developing SV assessment models based on textual SV descriptions/reports [96, 102, 129]. We extended this technique to code-based SV assessment by counting the frequency of code tokens. We also applied code-aware tokenization to preserve code syntax and semantics. For instance, `var++` was tokenized into `var` and `++`, explicitly informing a model about incrementing the variable `var` by one using the operator (`++`).

Bag-of-Subtokens. *Bag-of-Subtokens* extends Bag-of-Tokens by splitting extracted code tokens into sequences of characters (sub-tokens). These characters help a model learn less frequent tokens better. For instance, an infrequent variable like `ThisIsAVeryLongVar` is decomposed into multiple sub-tokens; one of which is `Var`, telling a model that this is a potential variable. We extracted sub-tokens of lengths ranging from two to six. Such values have been previously adopted for SV assessment [133, 23]. We did not use one-letter characters as they were too noisy, while using more than six characters would significantly increase feature size and computational cost.

Word2vec. Unlike Bag-of-Tokens and Bag-of-Subtokens that do not consider token context, *Word2vec* [202] extracts features of a token based on its surrounding counterparts. The contextual information from surrounding tokens helps produce similar feature vectors in an embedding space for tokens with (nearly) identical functionality/usage (e.g., `average` and `mean` variables). Word2vec generates vectors for individual tokens, so we averaged the vectors of all input tokens to represent a code snippet. This averaging method has been demonstrated to be effective for various NLP tasks [298]. Table 4.1 lists different values for the window and vector sizes of Word2vec used for tuning the performance of learning-based SV assessment models.

fastText. *fastText* [196] enhances Word2vec by representing each token with an aggregated feature vector of its constituent sub-tokens. Technically, fastText combines the strengths of semantic representation of Word2vec and subtoken-augmented features of Bag-of-Subtokens. fastText has been shown to build competitive yet compact report-level SV assessment models [23]. Like Word2vec, the feature vector of a code snippet was averaged from the vectors of all the input tokens. The length of sub-tokens also ranged from two to six, resembling that of Bag-of-Subtokens. Other hyperparameters of fastText for optimization are listed in Table 4.1.

CodeBERT. *CodeBERT* [299] is an adaptation of BERT [80], the current state-of-the-art feature representation technique in NLP, to source code modeling. CodeBERT is a pre-trained model using both natural language and programming language data to produce contextual embedding for code tokens. The same code token can have different CodeBERT embedding vectors depending on other tokens in an input; whereas, word2vec/fastText produces a single vector for every token regardless of its context. In addition, the source code tokenizer of CodeBERT is built upon Byte-Pair Encoding (BPE) [300]. This tokenizer smartly retains sub-tokens that frequently appear in a training corpus rather than keeping all of them as in Bag-of-Subtokens and fastText, balancing between performance and cost. CodeBERT also preserves a special token, *[CLS]*, to represent an entire code input. We leveraged the vector of this *[CLS]* token to extract the features for each code snippet.

We trained all the feature models from scratch, except CodeBERT as it is a pre-trained model. We used CodeBERT's pre-trained vocabulary and embeddings as commonly done in the literature [301]. To build the vocabulary for the other feature extraction methods, we considered tokens appearing at least in two samples in a training dataset to avoid vocabulary explosion due to too rare tokens. The exact vocabulary depended on the dataset used in each of the 10 training/evaluation rounds, as described in section 4.4.5. Note that some code snippets, e.g., vulnerable lines extracted from (partial) code changes, were not compilable (i.e., did not contain complete code syntax); thus, we did not use Abstract Syntax Tree (AST) based code representation like Code2vec [288] in this study as such representation may not be robust for these cases [287, 302]. It is worth noting that

TABLE 4.1: Hyperparameter tuning for SV assessment models.

Step	Model	Hyperparameters
Feature extraction	Word2vec [298] fastText [196]	*Vector size*: 150, 300, 500 *Window size*: 3, 4, 5
CVSS metrics prediction	Logistic Regression (LR) [249] Support Vector Machine (SVM) [206]	*Regularization coefficient*: 0.01, 0.1, 1, 10, 100
	K-Nearest Neighbors (KNN) [303]	*No. of neighbors*: 5, 11, 31, 51 *Weight*: uniform, distance *Distance norm*: 1, 2
	Random Forest (RF) [251] Extreme Gradient Boosting (XGB) [78] Light Gradient Boosting Machine (LGBM) [79]	*No. of estimators*: 100, 200, 300, 400, 500 *Max depth*: 3, 5, 7, 9, unlimited *Max. no. of leaf nodes*: 100, 200, 300, unlimited (RF)

Bag-of-Tokens, Bag-of-Subtokens, Word2vec, fastText, and CodeBERT can still work with these cases as these methods operate directly on code tokens.

4.4.4 Data-Driven SV Assessment Models

Features generated from code inputs enter ML models for predicting the CVSS metrics. The predictions of the CVSS metrics are classification problems (see Fig. 4.3). We used six well-known Machine Learning (ML) models for classifying the classes of each CVSS metric: Logistic Regression (LR) [249], Support Vector Machine (SVM) [206], K-Nearest Neighbors (KNN) [303], Random Forest (RF) [251], eXtreme Gradient Boosting (XGB) [78], and Light Gradient Boosting Machine (LGBM) [79]. LR, SVM, and KNN are single models, while RF, XGB, and LGBM are ensemble models that leverage multiple single counterparts to reduce overfitting. These classifiers have been used for SV assessment based on SV reports [22, 23]. We also considered different hyperparameters for tuning the performance of the classifiers, as given in Table 4.1. These hyperparameters have been adapted from the prior studies using similar classifiers [22, 23, 304]. Here, we mainly focus on ML techniques, and thus using Deep Learning models [203] for the tasks is out of the scope of this study.

4.4.5 Model Evaluation

Evaluation technique. To develop function-level SV assessment models and evaluate their performance, we used 10 rounds of training, validation, and testing. We randomly shuffled the dataset of vulnerable functions in section 4.4.1 and then split it into 10 partitions of roughly equal size.[4] In round i, for a model, we used fold $i+1$ for validation, fold $i+2$ for testing, and all of the remaining folds for training. When $i+1$ or $i+2$ was larger than 10, its value was wrapped around. For example, if $i = 10$, then $(i+1) \bmod 10 = 11 \bmod 10 = 1$ and $(i+2) \bmod 10 = 12 \bmod 10 = 2$. A grid search of the hyperparameters in Table 4.1 was performed using the validation sets to select optimal models. The performance of such optimal models on the test sets was reported. It is important to note that our evaluation strategy improves upon 10-fold cross-validation and random splitting data into a single training/validation/test set, the two most commonly

[4]With 1,782 samples in total, folds 1-9 had 178 samples and fold 10 had 180 samples.

used evaluation techniques in (fine-grained) SV detection and report-level SV assessment studies [305, 306, 278, 276, 277, 11]. Our evaluation technique has separate test sets, which cross-validation does not, to objectively measure the performance of tuned/optimal models on unseen data. Using multiple (10) validation/test sets also increases the stability of results compared to a single set [212]. Moreover, we aim to provide baseline performance for function-level SV assessment in this study, so we did not apply any techniques like class rebalancing or feature selection/reduction to augment the data/features. Such augmentation can be explored in future work. We also did not compare SV assessment using functions with that using reports as their use cases are different; function-level SV assessment is needed when SV reports are unavailable/unusable. It may be fruitful to compare/combine these two artifacts for SV assessment in the future.

Evaluation measures. We used F1-Score[5] and Matthews Correlation Coefficient (MCC) measures to quantify how well developed models perform SV assessment tasks. F1-Score values are from 0 to 1, and MCC has values from –1 to 1; 1 is the best value for both measures. These measures have been commonly used for SV assessment (e.g., [22, 124, 23]). We used MCC as the main measure for selecting optimal models because MCC takes into account all classes, i.e., all cells in a confusion matrix, during evaluation [307].

Statistical analysis. To confirm the significance of results, we employed the one-sided Wilcoxon signed rank test [254] and its respective effect size ($r = Z/\sqrt{N}$, where Z is the Z-score statistic of the test and N is the total count of samples) [308].[6] We used the Wilcoxon signed-rank test because it is a non-parametric test that can compare two-paired groups of data, and we considered a test significant if its confidence level was more than 99% (p-value < 0.01). We did not use the popular Cohen's D [310] and Cliff's δ [311] effect sizes as they are not suitable for comparing paired data [279].

4.5 Results

4.5.1 RQ1: Are Vulnerable Code Statements More Useful Than Non-Vulnerable Counterparts for SV Assessment Models?

Based on the extraction process in section 4.4.1, we collected 1,782 vulnerable functions containing 5,179 vulnerable and 57,633 non-vulnerable statements. The proportions of these two types of statements are given in the first and second boxplots, respectively, in Fig. 4.4. On average, 14.7% of the lines in the selected functions were vulnerable, 5.8 times smaller than that of non-vulnerable lines. Interestingly, we also observed that 55% of the functions contained only a single vulnerable statement. These values show that vulnerable statements constitute a very small proportion of functions.

Despite the small size (no. of lines), vulnerable statements contributed more to the predictive performance of the seven assessment tasks than non-vulnerable statements (see Table 4.2). We considered two variants of non-vulnerable statements for comparison. The first variant, *Non-vuln (random)*, randomly selected the same number of lines as vulnerable statements from non-vulnerable statements in each function. The second variant, *Non-vuln (all)* aka. Non-vuln (All - Vuln) in Fig. 4.4, considered all non-vulnerable statements. Compared to same-sized non-vulnerable statements (Non-vuln (random)), *Vuln-only* (using vulnerable statements solely) produced 116.9%, 126.6%, 98.7%, 90.7%, 147.9%, 111.2%, 116.7% higher MCC for Access Vector, Access Complexity, Authentication, Confidentiality, Integrity, Availability, and Severity tasks, respectively. On average, Vuln-only was 114.5% and 43.6% better than Non-vuln (random) in MCC

[5] The macro version of F1-Score was used for multi-class classification.

[6] $r \leq 0.1$: negligible, $0.1 < r \leq 0.3$: small, $0.3 < r \leq 0.5$: medium, $r > 0.5$: large [309]

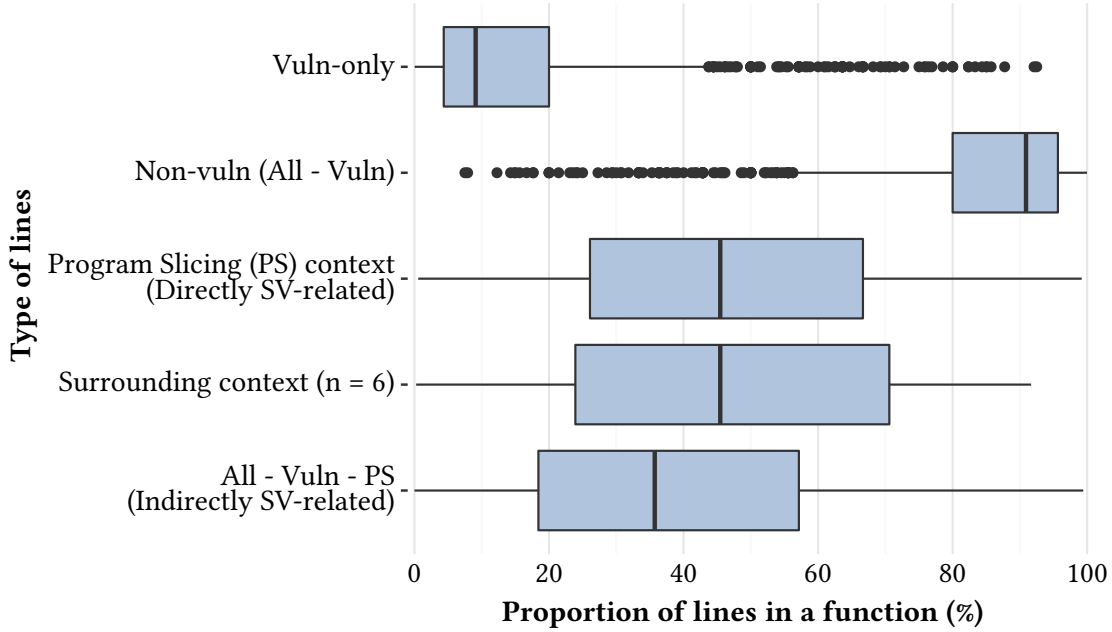


FIGURE 4.4: Proportions of different types of lines in a function. **Notes**: Min proportion of Vuln-only lines is non-zero (0.03%); thus, max of Non-vuln line proportion is $< 100\%$. Cosmetic lines were excluded from the computation of ratios.

and F1-Score, respectively. We obtained similar results of Non-vuln (random) when repeating the experiment with differently randomized lines. When using all non-vulnerable statements (Non-vuln (all)), the assessment performance increased significantly, yet was still lower than that of vulnerable statements. Average MCC and F1-Score of Vuln-only were 7.4% and 5.5% higher than Non-vuln (all), respectively. The improvements of Vuln-only over the two variants of non-vulnerable statements were statistically significant across features/classifiers with p-values < 0.01 ($p\text{-value}_{Non-vuln(random)} = 1.7 \times 10^{-36}$ and $p\text{-value}_{Non-vuln(all)} = 7.2 \times 10^{-11}$) and non-negligible effect sizes ($r_{Non-vuln(random)} = 0.62$ and $r_{Non-vuln(all)} = 0.32$). The low performance of Non-vuln (random) implies that SV assessment models likely perform worse if vulnerable statements are incorrectly identified. Moreover, the decent performance of Non-vuln (all) shows that some non-vulnerable statements are potentially helpful for SV assessment, which are studied in detail in RQ2.

4.5.2 RQ2: To What Extent do Different Types of Context of Vulnerable Statements Contribute to SV Assessment Performance?

In RQ2, we compared the performance of models using *Program Slicing* (PS), *surrounding* and *function* context. Notably, we removed 365 cases for which we could not extract PS context from the dataset in section 4.4.1. Apparently, these cases were the same as using only vulnerable statements, making comparisons biased, especially against Vuln-only itself. The vulnerable statements in these cases mainly did not contain any variable, e.g., using an unsafe function without parameter.[7] In this new dataset, PS and surrounding context approximately constituted 46%, on average, of lines in vulnerable functions (see Fig. 4.4). We used six lines before and after vulnerable statements (n = 6) as surrounding context because this value resulted in the closest average context size to that of PS context. It is impossible to have the exactly same size because PS context is dynamically derived from

[7] https://bit.ly/3pg36mp

TABLE 4.2: Testing performance for SV assessment tasks of vulnerable vs. non-vulnerable statements. **Note**: Bold and grey-shaded values are the best row-wise performance.

CVSS metric	Evaluation metric	Input type		
		Vuln-only	Non-vuln (random)	Non-vuln (all)
Access Vector	F1-Score	**0.820**	0.650	0.786
	MCC	**0.681**	0.314	0.605
Access Complexity	F1-Score	**0.622**	0.458	0.592
	MCC	**0.510**	0.225	0.467
Authentication	F1-Score	**0.791**	0.602	0.765
	MCC	**0.630**	0.317	0.614
Confidentiality	F1-Score	**0.645**	0.411	0.625
	MCC	**0.574**	0.301	0.561
Integrity	F1-Score	**0.650**	0.384	0.616
	MCC	**0.585**	0.236	0.534
Availability	F1-Score	**0.647**	0.417	0.624
	MCC	**0.583**	0.276	0.551
Severity	F1-Score	**0.695**	0.414	0.610
	MCC	**0.583**	0.269	0.523
Average	F1-Score	**0.695**	0.484	0.659
	MCC	**0.592**	0.276	0.551

vulnerable statements, while surrounding context is predefined. The roughly similar size helps test whether directly SV-related lines in PS context would be better than pre-defined surrounding lines for SV assessment. The training and evaluation processes on the dataset in RQ2 were the same as in RQ1.[8]

Adding context to vulnerable statements led to better SV assessment performance than using vulnerable statements only (see Fig. 4.5). Among the three, function context was the best, followed by PS and then surrounding context. In terms of MCC, function context working together with vulnerable statements beat Vuln-only by 6.4%, 6.5%, 9%, 8.2%, 11%, 11.4%, 9.7% for Access Vector, Access Complexity, Authentication, Confidentiality, Integrity, Availability, and Severity tasks, respectively. The higher F1-Score values when incorporating function context to vulnerable statements are also evident in Fig. 4.5. On average, combining function context and vulnerable statements attained 0.64 MCC and 0.75 F1-Score, surpassing using vulnerable lines solely by 8.9% in MCC and 8.5% in F1-Score. Although PS context + Vuln performed slightly worse than function context + Vuln, MCC and F1-Score of PS context + Vuln were still 6.7% and 7.5% ahead of Vuln-only, respectively. The improvements of function and PS context + Vuln over Vuln-only were significant across features/classifiers, i.e., p-values of 1.2×10^{-17} and 2.1×10^{-13} and medium effect sizes of 0.42 and 0.36, respectively. Compared to function/PS context + Vuln, surrounding context + Vuln outperformed Vuln-only by smaller margins, i.e., 3% for MCC and 5.2% for F1-Score (p-value $= 3.7 \times 10^{-8} < 0.01$ with a small effect size ($r = 0.27$)). These findings show the usefulness of directly SV-related (PS) lines for SV assessment models, while six lines surrounding vulnerable statements seemingly contain less related information for the SV assessment tasks.

[8]The RQ1 findings still hold when using the new dataset in RQ2, yet with a slight (≈2%) decrease in absolute model performance.

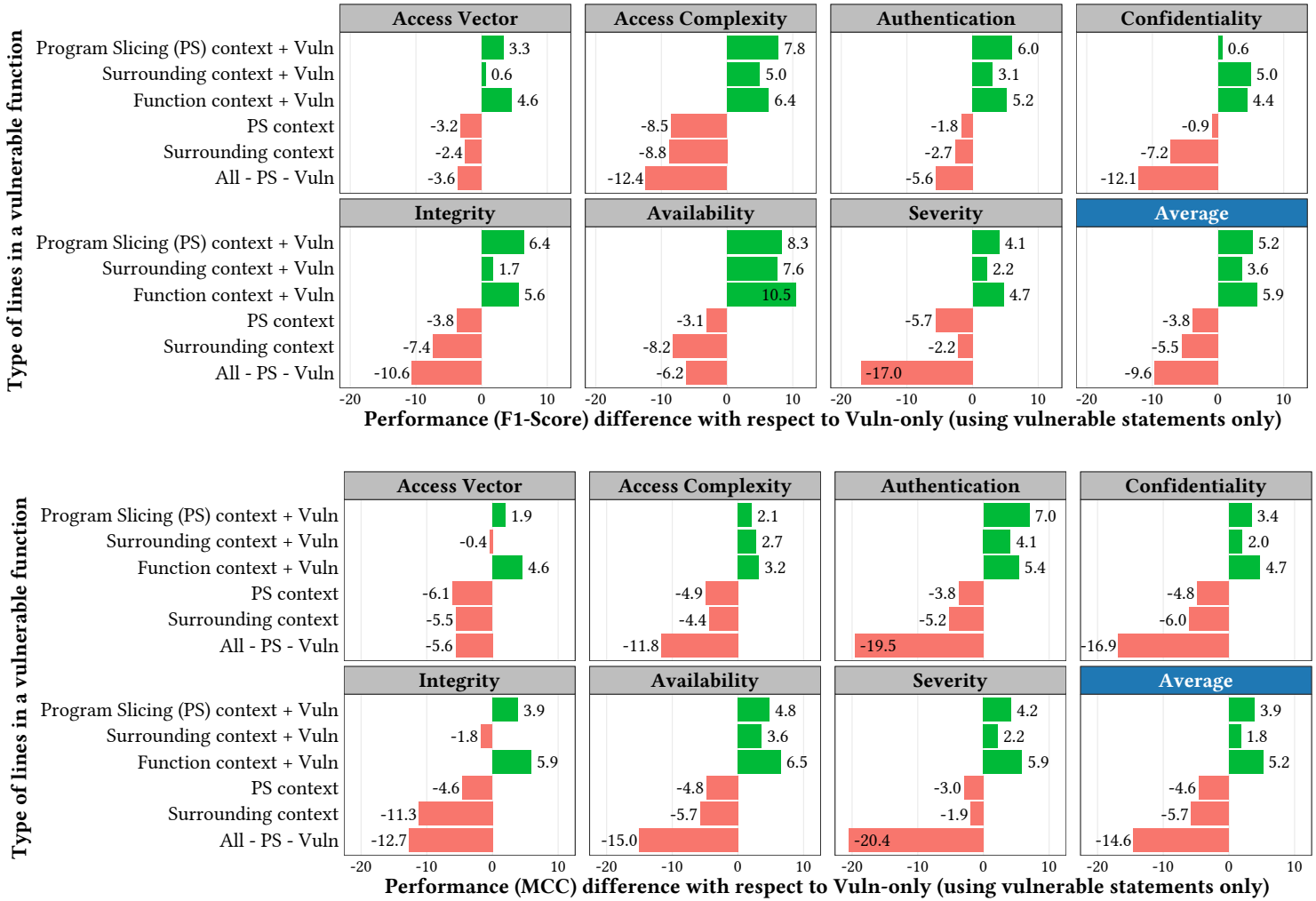


FIGURE 4.5: Differences in testing SV assessment performance (F1-Score and MCC) between models using different types of lines/context and those using only vulnerable statements. **Note**: The differences were multiplied by 100 to improve readability.

Further investigation revealed that only 49% of lines in PS context overlapped with those in surrounding context (n = 6). Note that the performance of surrounding context tended to approach that of function context as the surrounding context size increased. Using the dataset in RQ1, we also obtained the same patterns, i.e., function context + Vuln > surrounding context + Vuln > Vuln-only. This result shows that function context is generally better than the other context types, indicating the plausibility of building effective SV assessment models using only the output of function-level SV detection (i.e., requiring no knowledge about which statements are vulnerable in each function).

Although the three context types were useful for SV assessment when combined with vulnerable statements, using these context types alone significantly reduced the performance. As shown from RQ1, using only function context (i.e., Non-vuln (all) in Table 4.2) was 6.9% inferior in MCC and 5.2% lower in F1-Score than Vuln-only. Using the new dataset in RQ2, we obtained similar reductions in MCC and F1-Score values. Fig. 4.5 also indicates that using only PS and surrounding context decreased MCC and F1-Score of all the tasks. Particularly, using PS context alone reduced MCC and F1-Score by 7.8% and 5.5%, respectively; whereas, such reductions in values for using only surrounding context were 9.8% and 8%. These performance drops were confirmed significant with p-values < 0.01 and non-negligible effect sizes. Overall, the performance rankings of the context types with and without vulnerable statements were the same, i.e., function > PS > surrounding context. We also observed that all context types were better (increasing 20.1-28.2% in MCC and 6.9-17.5% in F1-Score) than non-directly SV-related lines (i.e., All - PS - Vuln in Fig. 4.5). These findings highlight the need for using context together with vulnerable statements rather than using each of them alone for function-level SV assessment tasks.

4.5.3 RQ3: Does Separating Vulnerable Statements and Context to Provide Explicit Location of SVs Improve Assessment Performance?

RQ3 evaluated the approach of separating vulnerable statements from their context as two inputs for building SV assessment models. Theoretically, this double-input method tells a model the exact vulnerable and context parts in input code, helping the model capture the information from relevant parts for SV assessment tasks more easily. To separate features, feature vectors are generated for each of the two inputs and then concatenated to form a single vector of twice the size of the vector used in RQ2. RQ3 used the same dataset from RQ2 (i.e., excluding cases without PS context) and the respective model evaluation procedure to objectively compare PS context with the other context types.

Overall, double-input models improved the performance for all types of context compared to single-input ones, but the improvements were not substantial (≈1%). Table 4.3 clearly indicated the improvement trend; i.e., a majority of the cells have green color. We noticed that the rankings of double-input models using different context types still remained the same as in RQ2, i.e., function > PS > surrounding context. Specifically, double-input models raised the MCC values of single-input models using PS, surrounding, and function context by 1.1%, 1.4%, and 0.8%, respectively. In terms of F1-Score of double-input models, PS and surrounding context had 0.26% and 0.75% increase, while function context suffered from a 0.04% decline. We found the absolute performance differences between double-input and single-input models for the seven tasks were actually small and not statistically significant with negligible effect sizes ($r_{PS\ ctx+Vuln} = 0.059$, $r_{Surrounding\ ctx+Vuln} = 0.092$, and $r_{Function\ ctx+Vuln} = 0.021$). We observed similar changes/patterns of function/surrounding context when using the full dataset in RQ1. The findings suggest that models using function context + Vuln as a single-input in RQ2 still perform competitively. This result strengthens the conclusion in

TABLE 4.3: Differences in testing performance for SV assessment tasks between double-input models (RQ3) and single-input models (RQ1/2). **Notes**: The differences were multiplied by 100 to increase readability. Green and red colors denote value increase and decrease, respectively. Darker color shows a higher magnitude of increase/decrease. Average performance values (multiplied by 100) of double-input models for the three context (ctx) types are in parentheses.

CVSS metric	Evaluation metric	Input type (double)		
		PS ctx + Vuln	Surrounding ctx + Vuln	Function ctx + Vuln
Access Vector	F1-Score	1.3	1.2	1.2
	MCC	2.6	2.7	2.4
Access Complexity	F1-Score	-0.5	0.6	-0.7
	MCC	1.5	0.5	2.4
Authentication	F1-Score	0.5	0.7	0.3
	MCC	1.0	1.7	0.7
Confidentiality	F1-Score	-0.2	0.01	-0.9
	MCC	-0.7	-0.5	-1.5
Integrity	F1-Score	0.2	0.7	0.2
	MCC	-0.07	1.1	0.5
Availability	F1-Score	-0.5	-0.2	-1.0
	MCC	0.4	0.4	0.1
Severity	F1-Score	0.7	0.9	0.7
	MCC	0.2	0.02	0.2
Average	F1-Score	0.2 (74.7)	0.5 0.5 (73.4)	-0.03 (75.2)
	MCC	0.7 (63.1)	0.8 (61.1)	0.5 (64.1)

RQ2 that SV assessment models benefit from vulnerable statements along with (in-)directly SV-related lines in functions, yet not necessarily where these lines are located.

4.6 Discussion

4.6.1 Function-Level SV Assessment: Baseline Models and Beyond

From RQ1-RQ3, we have shown that vulnerable statements and their context are useful for SV assessment tasks. In this section, we discuss the performance of various features and classifiers used to develop SV assessment models on the function level. We also explore the patterns of false positives of the models used in this work. Through these discussions, we aim to provide recommendations on building strong baseline models and inspire future data-driven advances in function-level SV assessment.

Practices of building baselines. ***Among the investigated features and classifiers, a combination of LGBM classifier and Bag-of-Subtokens features produced the best overall performance for the seven SV assessment tasks (see Fig. 4.6)***. In addition, LGBM outperformed the other classifiers, and Bag-of-Subtokens was better than the other features. However, we did not find a single set of hyperparameters that was consistently better than the others, emphasizing the need for hyperparameter tuning for function-level SV assessment tasks, as generally recommended in the literature [312, 313]. Regarding the classifiers, the ensemble ones (LGBM, RF, and XGB) were significantly better than the single counterparts (SVM, LR, and KNN) when averaging across all feature types, aligning with the previous findings for SV assessment using SV reports [23, 22].

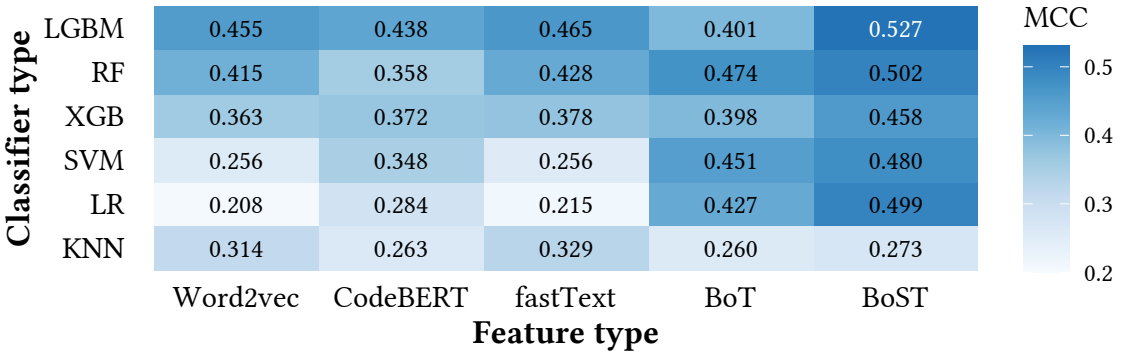


FIGURE 4.6: Average performance (MCC) of six classifiers and five features for SV assessment in functions. Notes: BoT and BoST are Bag-of-Tokens and Bag-of-Subtokens, respectively.

Regarding the features, the ones augmented by sub-tokens (Bag-of-Subtokens, fastText, and CodeBERT) had stronger performance, on average, than the respective feature types using only word-based representation (Bag-of-Tokens and Word2vec). This observation suggests that SV assessment models do benefit from sub-tokens, probably because rare code tokens are more likely to be captured by these features. This result is similar to Le et al. [23]'s finding for report-level SV assessment models. All of the aforementioned result comparisons were confirmed statistically significant with p-values < 0.01 and non-negligible effect sizes; similar patterns were also obtained for F1-Score. Surprisingly, the latest feature model, CodeBERT, did not show superior performance in this scenario, likely because the model was originally pre-trained on multiple languages, not only Java (the main language used in this study). Fine-tuning the weights of CodeBERT using SV data in a specific programming language is worthy of exploration for potentially improving the performance of this feature model. Overall, using the aforementioned baseline features and classifiers, we observed a common pattern in the performance ranking of the seven CVSS metrics across different input types, i.e., Access Vector $>$ Authentication $>$ Severity $>$ Confidentiality – Integrity – Availability $>$ Access Complexity. We speculate that the metric-wise class distribution (see Fig. 4.3) can be a potential explanation for this ranking. Specifically, Access Vector and Authentication are binary classifications, which have less uncertainty than the other tasks. In addition, Confidentiality, Integrity, and Availability are all impact metrics with roughly similar distributions, resulting in similar performance as well. Access Complexity suffers the most from imbalanced data among the tasks, and thus this task has the worst performance.

False-positive patterns. We manually analyzed the incorrect predictions by the optimal models using the best-performing (function) context from RQ2/RQ3. From these cases, we found two key patterns of false positives. ***The first pattern concerned SVs affecting implicit code in function calls***. For example, a feature requiring authentication, i.e., the synchronous mode, was run before user's password was checked in a function (`doFilter`), leading to a potential access-control related SV.[9] The execution of such mode was done by another function, `processSync`, but its implementation along with the affected components inside was not visible to the affected function. Such invisibility hinders a model's ability to fully assess SV impacts. A straightforward solution is to employ inter-procedural analysis [314], but scalability is a potential issue as SVs can affect multiple functions and even the ones outside of a project (i.e., functions in third-party libraries). Future work can leverage Question and Answer websites to retrieve and analyze SV-related information of

[9] https://bit.ly/32vyggM (CVE-2018-1000134)

third-party libraries [315]. ***The second type of false positives involved vulnerable variables with obscure context***. For instance, a function used a potentially vulnerable variable containing malicious inputs from users, but the affected function alone did not contain sufficient information/context about the origin of the variable.[10] Without such variable context, a model would struggle to assess the exploitability of an SV; i.e., through which components attackers can penetrate into a system and whether any authentication is required during the penetration. Future work can explore taint analysis [316] to supplement function-level SV assessment models with features about variable origin/flow.

4.6.2 Threats to Validity

The first threat concerns the curation of vulnerable functions and statements for building SV assessment models. We considered the recommendations in the literature to remove noise in the data (e.g., abnormally large and cosmetic changes). We also performed manual validation to double-check the validity of our data.

Another threat is about the robustness of our own implementation of the program slicing extraction. To mitigate the threat, we carefully followed the algorithms and descriptions given in the previous work [297] to extract the intra-procedural backward and forward slices for a particular code line.

Other threats are related to the selection and optimality of baseline models. We assert that it is nearly impossible to consider all types of available features and models due to limited resources. Hence, we focused on the common techniques and their respective hyperparameters previously used for relevant tasks, e.g., report-level SV assessment. We are also the first to tackle function-level SV assessment using data-driven models, and thus our imperfect baselines can still stimulate the development of more advanced and better-performing techniques in the future.

Regarding the reliability of our study, we confirmed the key findings with p-values < 0.01 using non-parametric Wilcoxon signed rank tests and non-negligible effect sizes. Regarding the generalizability of our results, we only performed our study in the Java programming language, yet we mitigated this threat by using 200 real-world projects of diverse domains and scales. The data and models were also released at `https://github.com/lhmtriet/Function-level-Vulnerability-Assessment` to support reuse and extension to new languages/applications.

4.7 Related Work

4.7.1 Code Granularities of SV Detection

SV detection has long attracted attention from researchers and there have been many proposed data-driven solutions to automate this task [3]. Neuhaus et al. [317] were among the first to tackle SV detection in code components/files. This seminal work has inspired many follow-up studies on component/file-level SV detection (e.g., [318, 319, 320]). Over time, function-level SV detection tasks have become more popular [321, 273, 322, 323, 324] as functions are usually much smaller than files, significantly reducing inspection effort for developers. For example, the number of code lines in the methods in our dataset was only 35, on average, nearly 10 times smaller than that (301) of files. Recently, researchers have begun to predict exact vulnerable statements/lines in functions (e.g., [278, 276, 277, 325]). This emerging research is based on an important observation that only a small number of lines in vulnerable functions contain root causes of SVs. Instead of detecting SVs as in these studies, we focus on SV assessment tasks after SVs are detected. Specifically, utilizing the

[10] `https://bit.ly/3plU7QP` (CVE-2012-0391)

outputs (vulnerable functions/statements) from these SV detection studies, we perform function-level SV assessment to support SV understanding/prioritization before fixing.

4.7.2 Data-Driven SV Assessment

SV assessment has been an integral step for addressing SVs. CVSS has been shown to provide one of the most reliable metrics for SV assessment [261]. According to Chapter 2, there has been a large and growing body of research work on automating SV assessment tasks, especially predicting the CVSS metrics for ever-increasing SVs. Most of the current studies (e.g., [95, 20, 22, 23, 100, 326]) have utilized SV descriptions available in bug/SV reports/databases, mostly NVD, to predict the CVSS metrics. However, according to our analysis in section 4.2, NVD reports of SVs are usually released long (up to 1k days) after SVs have been fixed, rendering report-level SV assessment potentially untimely for SV fixing. Unlike the current studies, we propose shifting the SV assessment tasks to the function level, which can help developers assess functions right after they are found vulnerable and before fixing. Note that we assess all types of SVs in source code, not only the ones in dependencies [327, 13]. Overall, our study informs the practice of developing strong baselines for function-level SV assessment tasks by combining vulnerable statements and their context. It is worth noting that function-level SV assessment does not completely replace report-level SV assessment. The latter is still useful for assessing SVs in third-party libraries/software without available source code (e.g., commercial products), to prioritize the application of security patches provided by vendors.

4.8 Chapter Summary

We motivate the need for function-level SV assessment to provide essential information for developers before fixing SVs. Through large-scale experiments, we studied the use of data-driven models for automatically assigning the seven CVSS assessment metrics to SVs in functions. We demonstrated that strong baselines for these tasks benefited not only from fine-grained vulnerable statements, but also the context of these statements. Specifically, using vulnerable statements with all the other lines in functions produced the best performance of 0.64 MCC and 0.75 F1-Score. These promising results show that function-level SV assessment tasks deserve more attention and contribution from the community, especially techniques that can strongly capture the relations between vulnerable statements and other code lines/components.

Chapter 5

Automated Commit-Level Software Vulnerability Assessment

Related publication: This chapter is based on our paper titled "*DeepCVA: Automated Commit-level Vulnerability Assessment with Deep Multi-task Learning*" published in the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021 (CORE A*) [289].

In Chapter 4, we have distilled practices of performing Software Vulnerability (SV) assessment on the code function level. While function-level assessment is useful for analyzing SVs before fixing, some SVs may be detected late in codebases and pose significant security risks for a long time. Thus, it is increasingly suggested to identify SV in code commits to give early warnings about potential security risks. However, there is a lack of effort to assess vulnerability-contributing commits right after they are detected to provide timely information about the exploitability, impact and severity of SVs. Such information is important to plan and prioritize the mitigation for the identified SVs. In Chapter 5, we propose a novel Deep multi-task learning model, DeepCVA, to automate seven Commit-level Vulnerability Assessment tasks simultaneously based on Common Vulnerability Scoring System (CVSS) metrics. We conduct large-scale experiments on 1,229 vulnerability-contributing commits containing 542 different SVs in 246 real-world software projects to evaluate the effectiveness and efficiency of our model. We show that DeepCVA is the best-performing model with 38% to 59.8% higher Matthews Correlation Coefficient than many supervised and unsupervised baseline models. DeepCVA also requires 6.3 times less training and validation time than seven cumulative assessment models, leading to significantly less model maintenance cost as well. Overall, DeepCVA presents the first effective and efficient solution to automatically assess SVs early in software systems.

5.1 Introduction

As reviewed in Chapter 2, existing techniques (e.g., [328, 21, 22, 23, 11]) to automate bug/-Software Vulnerability (SV) assessment have mainly operated on bug/SV reports, but these reports may be only available long after SVs appeared in practice. Our motivating analysis revealed that there were 1,165 days, on average, from when an SV was injected in a codebase until its report was published on National Vulnerability Database (NVD) [16]. Our analysis agreed with the findings of Meneely et al. [25]. To tackle late-detected bugs/SVs, recently, Just-in-Time (commit-level) approaches (e.g., [287, 26, 329, 330]) have been proposed to rely on the changes in code commits to detect bugs/SVs right after bugs/SVs are added to a codebase. Such early commit-level SV detection can also help reduce the delay in SV assessment.

Even when SVs are detected early in commits, we argue that existing automated techniques relying on bug/SV reports still struggle to perform *just-in-time* SV assessment. Firstly, there are significant delays in the availability of SV reports, which render the existing SV assessment techniques unusable. Specifically, SV reports on NVD generally only appear seven days after the SVs were found/disclosed [331]. Some of the detected SVs may not even be reported on NVD [332], e.g., because of no disclosure policy. User-submitted bug/SV reports are also only available post-release and more than 82% of the reports are filed more than 30 days after developers detected the bugs/SVs [333]. Secondly, code review can provide faster SV assessment, but there are still unavoidable delays (from several hours to even days) [334]. Delays usually come from code reviewers' late responses and manual analyses depending on the reviewers' workload and code change complexity [335]. Thirdly, it is non-trivial to automatically generate bug/SV reports from vulnerable commits as it would require non-code artifacts (e.g., stack traces or program crashes) that are mostly unavailable when commits are submitted [328, 336].

Performing commit-level SV assessment provides a possibility to inform committers about the exploitability, impact and severity of SVs in code changes and prioritize fixing earlier than current report-level SV assessment approaches. However, to the best of our knowledge, there is no existing work on automating SV assessment in commits. Prior SV assessment techniques that analyze text in SV databases (e.g., [21, 22, 23]) also cannot be directly adapted to the commit level. Contrary to text, commits contain deletions and additions of code with specific structure and semantics [287, 337]. Additionally, we speculate that the expert-based Common Vulnerability Scoring System (CVSS) metrics [29], which are commonly used to quantify the exploitability, impact and severity level of SVs for SV assessment, can be related. For example, an SQL injection is likely to be highly severe since attackers can exploit it easily via crafted input and compromise data confidentiality and integrity. We posit that these metrics would have common patterns in commits that can be potentially shared between SV assessment models. Predicting related tasks in a shared model has been successfully utilized for various applications [27]. For instance, an autonomous car is driven with simultaneous detection of vehicles, lanes, signs and pavement [338]. These observations motivated us to tackle a new and important research challenge, **"How can we leverage the common attributes of assessment tasks to perform effective and efficient commit-level SV assessment?"**

We present DeepCVA, a novel **Deep** multi-task learning model, to automate **C**ommit-level **V**ulnerability **A**ssessment. DeepCVA first uses attention-based convolutional gated recurrent units to extract features of code and surrounding context from vulnerability-contributing commits (i.e., commits with vulnerable changes). The model uses these features to predict seven CVSS assessment metrics (i.e., Confidentiality, Integrity, Availability, Access Vector, Access Complexity, Authentication, and Severity) simultaneously using the

multi-task learning paradigm. The predicted CVSS metrics can guide SV management and remediation processes.

Our key **contributions** are summarized as follows:

1. We are the first to tackle the commit-level SV assessment tasks that enable early security risks estimation and planning for SV remediation.
2. We propose a unified model, DeepCVA, to automate seven commit-level SV assessment tasks simultaneously.
3. We extensively evaluate DeepCVA on our curated large-scale dataset of 1,229 vulnerability-contributing commits with 542 SVs from 246 real-world projects.
4. We demonstrate that DeepCVA has 38% to 59.8% higher performance (Matthews Correlation Coefficient (MCC)) than various supervised and unsupervised baseline models using text-based features and software metrics. The proposed context-aware features improve the MCC of DeepCVA by 14.8%. The feature extractor with attention-based convolutional gated recurrent units, on average, adds 52.9% MCC for DeepCVA. Multi-task learning also makes DeepCVA 24.4% more effective and 6.3 times more efficient in training, validation, and testing than separate models for seven assessment tasks.
5. We release our source code, models and datasets for future research at `https://github.com/lhmtriet/DeepCVA`.

Chapter organization. Section 5.2 introduces preliminaries and motivation. Section 5.3 proposes the DeepCVA model for commit-level SV assessment. Section 5.4 describes our experimental design and setup. Section 5.5 presents the experimental results. Section 5.6 discusses our findings and threats to validity. Section 5.7 covers the related work. Section 5.8 concludes the work and proposes future directions.

5.2 Background and Motivation

5.2.1 Vulnerability in Code Commits

Commits are an essential unit of any version control system (e.g., Git) and record all the chronological changes made to the codebase of a software project. As illustrated in Fig. 5.1, changes in a commit consist of deletion(s) (–) and/or addition(s) (+) in each affected file.

Vulnerability-Contributing Commits (VCCs) are commits whose changes contain SVs [25], e.g., using vulnerable libraries or insecure implementation. We focus on VCCs rather than any commits with vulnerable code (in unchanged parts) since addressing VCCs helps mitigate SVs as early as they are added to a project. VCCs are usually obtained based on Vulnerability-Fixing Commits (VFCs) [329, 330]. An exemplary VFC and its respective VCC are shown in Fig. 5.1. VFCs delete, modify or add code to eliminate an SV (e.g., disabling external entities processing in the XML library in Fig. 5.1) and can be found in bug/SV tracking systems. Then, VCCs are commits that last touched the code changes in VFCs. Our work also leverages VFCs to obtain VCCs for building automated commit-level SV assessment models.

5.2.2 Commit-Level SV Assessment with CVSS

Similar to the studies in Chapters 3 and 4, we use Common Vulnerability Scoring System (CVSS) [29] version 2 of base metrics (i.e., Confidentiality, Integrity, Availability, Access

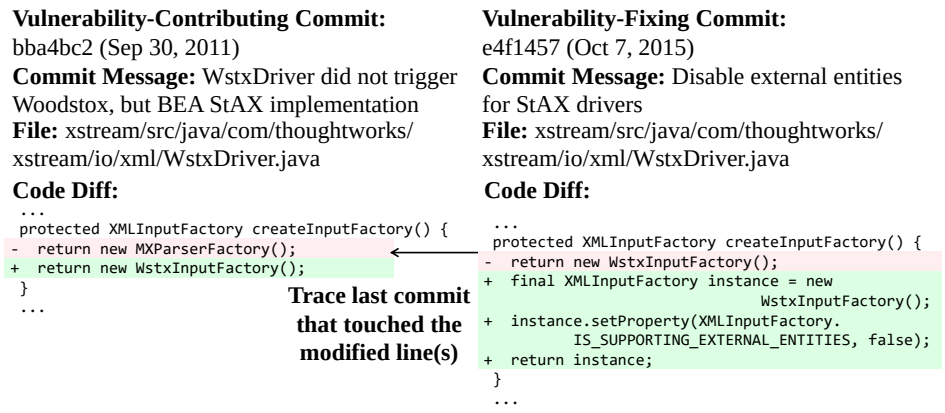


FIGURE 5.1: Exemplary SV fixing commit (right) for the XML external entity injection (XXE) (CVE-2016-3674) and its respective SV contributing commit (left) in the *xstream* project.

Vector, Access Complexity, Authentication, and Severity) to assess SVs in this study because of their popularity in practice. Based on CVSS version 2, the VCC (CVE-2016-3674) in Fig. 5.1 has a considerable impact on Confidentiality. This SV can be exploited with low (Access) complexity with no authentication via a public network (Access Vector), making it an attractive target for attackers.

Despite the criticality of these SVs, there have been delays in reporting, assessing and fixing them. Concretely, the VCC in Fig. 5.1 required 1,439 and 1,469 days to be reported[1] and fixed (in VFC), respectively. Existing SV assessment methods based on bug/SV reports (e.g., [21, 22, 23]) would need to wait more than 1,000 days for the report of this SV. However, performing SV assessment right after this commit was submitted can bypass the waiting time for SV reports, enabling developers to realize the exploitability/impacts of this SV and plan to fix it much sooner. To the best of our knowledge, there has not been any study on automated commit-level SV assessment, i.e., assigning seven CVSS base metrics to a VCC. Our work identifies and aims to bridge this important research gap.

5.2.3 Feature Extraction from Commit Code Changes

The extraction of commit features is important for building commit-level SV assessment models. Many existing commit-level defect/SV prediction models have only considered commit code changes (e.g., [287, 337, 339]). However, we argue that the nearby context of code changes also contributes valuable information to the prediction, as shown in Chapter 4. For instance, the surrounding code of the changes in Fig. 5.1 provides extra details; e.g., the method return statement is modified and the return type is `XMLInputFactory`. Such a type can help learn patterns of XXE SV that usually occurs with XML processing.

Besides the context, we speculate that SV assessment models can also benefit from the relatedness among the assessment tasks. For example, the XXE SV in Fig. 5.1 allows attackers to read arbitrary system files, which mainly affects the Confidentiality rather than the Integrity and Availability of a system. This chapter investigates the possibility of incorporating the common features of seven CVSS metrics into a single model using the multi-task learning paradigm [27] instead of learning seven cumulative individual models. Specifically, multi-task learning leverages the similarities and the interactions of the involved tasks through a shared feature extractor to predict all the tasks simultaneously. Such a unified model can significantly reduce the time and resources to train, optimize and maintain/update the model in the long run.

[1] https://github.com/x-stream/xstream/issues/25

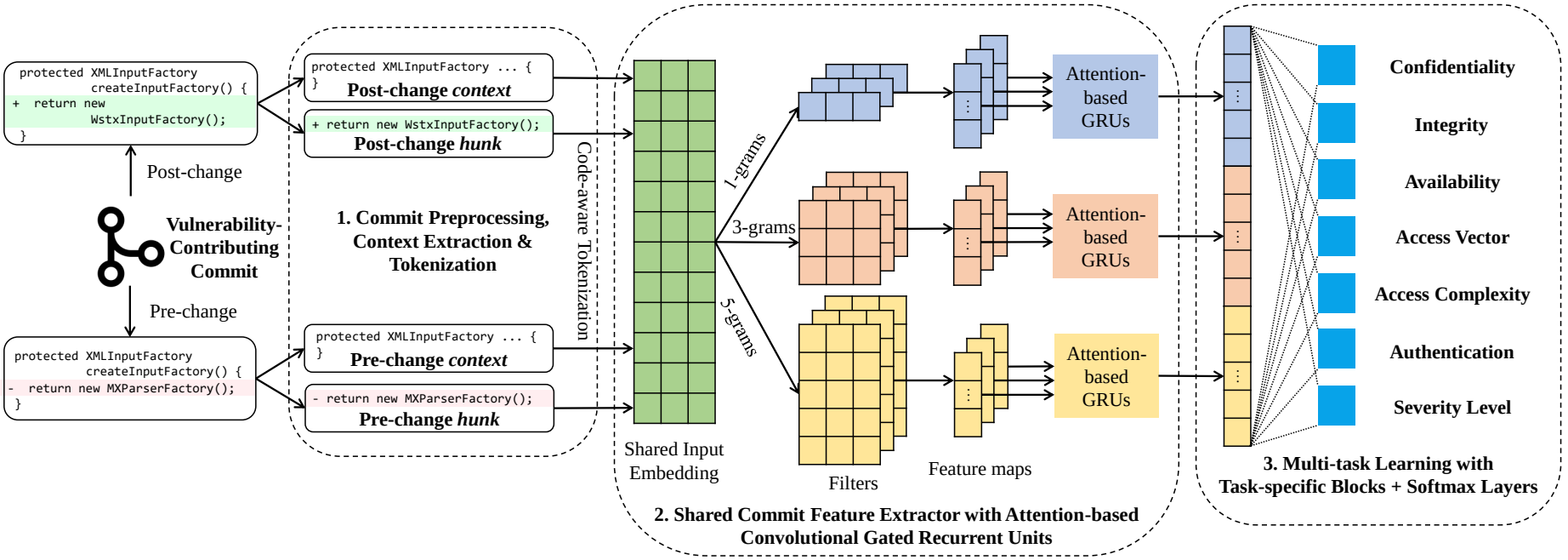


FIGURE 5.2: Workflow of DeepCVA for automated commit-level SV assessment. **Note**: The VCC is the one described in Fig. 5.1.

5.3 The DeepCVA Model

We propose **DeepCVA** (see Fig. 5.2), a novel **Deep** learning model to automate **C**ommit-level **V**ulnerability **A**ssessment. DeepCVA is a unified and end-to-end trainable model that concurrently predicts seven CVSS metrics (i.e., Confidentiality, Integrity, Availability, Access Vector, Access Complexity, Authentication, and Severity) for a Vulnerability-Contributing Commit (VCC). DeepCVA contains: (*i*) preprocessing, context extraction and tokenization of code commits (section 5.3.1), (*ii*) feature extraction from commits shared by seven assessment tasks using attention-based convolutional gated recurrent units (section 5.3.2), and (*iii*) simultaneous prediction of seven CVSS metrics using multi-task learning [27] (section 5.3.3). To assign the CVSS metrics to a new VCC with DeepCVA, we first preprocess the commit, obtain its code changes and respective context and tokenize such code changes/context. Embedding vectors of preprocessed code tokens are then obtained, and the commit feature vector is extracted using the trained feature extractor. This commit feature vector passes through the task-specific blocks and softmax layers to get the seven CVSS outputs with the highest probability values. Details of each component are given hereafter.

5.3.1 Commit Preprocessing, Context Extraction & Tokenization

To train DeepCVA, we first obtain and preprocess code changes (hunks) and extract the context of such changes. We then tokenize them to prepare inputs for feature extraction.
Commit preprocessing. Preprocessing helps remove noise in code changes and reduce computational costs. We remove newlines/spaces and inline/multi-line comments since they do not change code functionality. We do not remove punctuations (e.g., "`;`", "`(`", "`)`") and stop words (e.g., `and`/`or` operators) to preserve code syntax. We also do not lowercase code tokens since developers can use case-sensitivity for naming conventions of different token types (e.g., variable name: `system` vs. class name: `System`). Stemming (i.e., reducing a word to its root form such as `equals` to `equal`) is not applied to code since different names can change code functionality (e.g., the built-in `equals` function in Java).
Context extraction algorithm. We customize Sahal et al.'s [340] *Closest Enclosing Scope* (CES) to identify the context of vulnerable code changes for commit-level SV assessment (see section 5.2.3). Sahal et al. [340] defined an enclosing scope to be the code within a balanced amount of opening and closing curly brackets such as `if`/`switch`/`while`/`for` blocks. Among all enclosing scopes of a hunk, the one with the smallest size (lines of code) is selected as CES to reduce irrelevant code. Sahal et al. [340] found CES usually contains hunk-related information (e.g., variable values/types preceding changes). CES also alleviates the need for manually pre-defining the context size as in [329, 293]. Some existing studies (e.g., [280, 288]) only used the method/function scope, but code changes may occur outside of a method. For instance, changes in Fig. 5.3 do not have any enclosing method, but we can still obtain its CES, i.e., the `PlainNegotiator` class.

There are still two main limitations with the definition of CES in [340]. Firstly, a scope (e.g., `for`/`while` in Java) with single-line content does not always require curly brackets. Secondly, some programming languages do not use curly brackets to define scopes like Python. To address these two issues, we utilize Abstract Syntax Tree (AST) depth-first traversal (see Algorithm 2) to obtain CESs of code changes, as AST covers the syntax of all scope types and generalizes to any programming languages.

Algorithm 2 contains: (*i*) the `extract_scope` function for extracting potential scopes of a code hunk (lines 1-8), and (*ii*) the main code to obtain the CES of every hunk in a commit (lines 9-18). The `extract_scope` function leverages depth-first traversal with recursion to go through every node in an AST of a file. Line 3 adds the selected part

```
public class PlainNegotiator implements SaslNegotiator {
   ...
-  private static final String UTF8 = Standard
                                      Charsets.UTF_8.name();
+  private static final Charset UTF8 = Standard
                                      Charsets.UTF_8;
   ...
} // End of the PlainNegotiator class
```

FIGURE 5.3: Code changes outside of a method from the commit *4b9fb37* in the *Apache qpid-broker-j* project.

of an AST to the list of potential scopes (`potential_scopes`) of the current hunk. The first (root) AST is always valid since it encompasses the whole file. Line 6 then checks whether each node (sub-tree) of the current AST has one of the following types: `class`, `interface`, `enum`, `method`, `if`/`else`, `switch`, `for`/`while`/`do`, `try`/`catch`, and is surrounding the current hunk. If the conditions are satisfied, the `extract_scope` function would be called recursively in line 7 until a leaf of the AST is reached. The main code starts to extract the modified files of the current commit in line 9. For each file, we extract code hunks (code deletions/additions) in line 12 and then obtain the AST of the current file using an AST parser in line 13. Line 16 calls the defined `extract_scope` function to generate the potential scopes for each hunk. Among the identified scopes, line 17 adds the one with the smallest size (i.e., the number of code lines excluding empty lines and comments) to the list of CESs (`all_ces`). Finally, line 18 of Algorithm 2 returns all the CESs for the current commit.

We treat deleted (pre-change), added (post-change) code changes and their CESs as four separate inputs to be vectorized by the shared input embedding, as illustrated in Fig. 5.2. For each input, we concatenate all the hunks/CESs in all the affected files of a commit to explicitly capture their interactions.

Code-aware tokenization. The four inputs extracted from a commit are then tokenized with a code-aware tokenizer to preserve code semantics and help prediction models be more generalizable. For example, `a++` and `b++` are tokenized as `a`, `b` and `++`, explicitly giving a model the information about one-increment operator (`++`). Tokenized code is fed into a shared Deep Learning model, namely Attention-based Convolutional Gated Recurrent Unit (AC-GRU), to extract commit features.

5.3.2 Feature Extraction with Deep AC-GRU

Deep AC-GRU has a *three-way Convolutional Neural Network* to extract n-gram features and *Attention-based Gated Recurrent Units* to capture dependencies among code changes and their context. This feature extractor is shared by four inputs, i.e., deleted/added code hunks/context. Each input has the size of $N \times L$, where N is the no. of code tokens and L is the vector length of each token. All inputs are truncated or padded to the same length N to support parallelization. The feature vector of each input is obtained from a shared *Input Embedding* layer that maps code tokens into fixed-length arithmetic vectors. The dimensions of this embedding layer are $|V| \times L$, where $|V|$ is the code vocabulary size, and its parameters are learned together with the rest of the model.

Three-way Convolutional Neural Network. We use a shared three-way Convolutional Neural Network (CNN) [120] to extract n-grams (n = 1,3,5) of each input vector. The three-way CNN has filters with three sizes of one, three and five, respectively, to capture

Algorithm 2: AST-based extraction of the Closest Enclosing Scopes (CESs) of commit code changes.

Input: Current Vulnerability-Contributing Commit (VCC): *commit*
Scope type: *scope_types*
Output: CESs of code changes in the current commit: *all_ces*

Function `extract_scope(`$AST, hunk, visited = \emptyset$`):`
  global *potential_scopes*
  $potential\_scopes \longleftarrow potential\_scopes + AST$
  $visited \longleftarrow visited + AST$
  foreach $node \in AST$ **do**
    if $node \notin visited$ **and** $type(node) \in scope\_types$ **and** $start_{node} \leq start_{hunk}$ **and** $end_{node} \geq end_{hunk}$ **then**
      $extract\_scope(AST, hunk, visited)$
  return
$files \longleftarrow$ extract_files($commit$)
$all\_ces \longleftarrow \emptyset$
foreach $f_i \in files$ **do**
  $hunks \longleftarrow$ extract_hunk($commit, f_i$)
  $AST_i \longleftarrow$ extract_AST(f_i)
  foreach $h_i \in hunks$ **do**
    $potential\_scopes \longleftarrow \emptyset$
    extract_scopes(AST_i, h_i)
    $all\_ces \longleftarrow all\_ces + \underset{\text{size}}{\text{argmin}}(potential\_scopes)$
return *all_ces*

common code patterns, e.g., `public class Integer`. The filters are randomly initialized and jointly learned with the other components of DeepCVA. We did not include 2-grams and 4-grams to reduce the required computational resources without compromising the model performance, which has been empirically demonstrated in section 5.5.2. To generate code features of different window sizes with the three-way CNN, we multiply each filter with the corresponding input rows and apply non-linear ReLU activation function [341], i.e., $\text{ReLU}(x) = \max(0, x)$. We repeat the same convolutional process from the start to the end of an input vector by moving the filters down sequentially with a stride of one. This stride value is the smallest and helps capture the most fine-grained information from input code as compared to larger values. Each filter size returns feature maps of the size $(N - K + 1) \times F$, where K is the filter size (one, three or five) and F is the number of filters. Multiple filters are used to capture different semantics of commit data.

Attention-based Gated Recurrent Unit. The feature maps generated by the three-way CNN sequentially enter a Gated Recurrent Unit (GRU) [31]. GRU, defined in Eq. (5.1), is an efficient version of Recurrent Neural Networks and used to explicitly capture the order and dependencies between code blocks. For example, the `return` statement comes after the function declarations of the VCC in Fig. 5.2.

$$
\begin{aligned}
\mathbf{z}_t &= \sigma(\mathbf{W}_z \mathbf{x}_t + \mathbf{U}_z \mathbf{h}_{t-1} + b_z) \\
\mathbf{r}_t &= \sigma(\mathbf{W}_r \mathbf{x}_t + \mathbf{U}_r \mathbf{h}_{t-1} + b_r) \\
\hat{\mathbf{h}}_t &= \tanh(\mathbf{W}_h \mathbf{x}_t + \mathbf{U}_h(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + b_h) \\
\mathbf{h}_t &= (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \hat{\mathbf{h}}_t
\end{aligned}
\tag{5.1}
$$

where $\mathbf{W}_z$, $\mathbf{W}_r$, $\mathbf{W}_h$, $\mathbf{U}_z$, $\mathbf{U}_r$, $\mathbf{U}_h$ are learnable weights, b_z, b_r, b_h are learnable biases, $\odot$ is element-wise multiplication, σ is the sigmoid function and tanh() is the hyperbolic tangent function.
To determine the information ($\mathbf{h}_t$) at each token (time step) t, GRU combines the current input ($\mathbf{x}_t$) and the previous time step ($\mathbf{h}_{t-1}$) using the *update* ($\mathbf{z}_t$) and *reset* ($\mathbf{r}_t$) gates. $\mathbf{h}_t$ is then carried on to the next token until the end of the input to maintain the dependencies of the whole code sequence.

The last token output of GRU is often used as the whole sequence representation, yet it suffers from the *information bottleneck* problem [117], especially for long sequences. To address this issue, we incorporate the *attention mechanism* [117] into GRU to explicitly capture the contribution of each input token, as formulated in Eq. (5.2).

$$\begin{aligned} \mathbf{out}_{attention} &= \sum_{i=1}^{m} w_i \mathbf{h}_i \\ w_i &= \text{softmax}(\mathbf{W}_s \tanh(\mathbf{W}_a \mathbf{h}_i + b_a)) \\ &= \frac{\exp(\mathbf{W}_s \tanh(\mathbf{W}_a \mathbf{h}_i + b_a))}{\sum_{j=1}^{m} \exp(\mathbf{W}_s \tanh(\mathbf{W}_a \mathbf{h}_j + b_a))} \end{aligned} \tag{5.2}$$

where w_i is the weight of $\mathbf{h}_i$; $\mathbf{W}_s$, $\mathbf{W}_a$ are learnable weights, b_a is learnable bias, and m is the number of code tokens.

The attention-based outputs ($\mathbf{out}_{attention}$) of the three GRUs (see Fig. 5.2) are concatenated into a single feature vector to represent each of the four inputs (pre-/post-change hunks/contexts). The commit feature vector is a concatenation of the vectors of all four inputs generated by the shared AC-GRU feature extractor. This feature vector is used for multi-task prediction of seven CVSS metrics.

5.3.3 Commit-Level SV Assessment with Multi-task Learning

This section describes the multi-task learning layers of DeepCVA for efficient commit-level SV assessment (i.e., learning/predicting seven CVSS tasks simultaneously) using a single model as well as how to train the model end-to-end.
Multi-task learning layers. The last component of DeepCVA consists of the multi-task learning layers that simultaneously give the predicted CVSS values for seven SV assessment tasks. As illustrated in Fig. 5.2, this component contains two main parts: *task-specific blocks* and *softmax layers*. On top of the shared features extracted by AC-GRU, task-specific blocks are necessary to capture the differences among the seven tasks. Each task-specific block is implemented using a fully connected layer with non-linear ReLU activations [341]. Specifically, the output vector ($\mathbf{task}_i$) of the task-specific block for assessment task i is defined in Eq. (5.3).

$$\mathbf{task}_i = \text{ReLU}(\mathbf{W}_t \mathbf{x}_{commit} + b_t) \tag{5.3}$$

where $\mathbf{x}_{commit}$ is the commit feature vector from AC-GRU; $\mathbf{W}_t$ is learnable weights and b_t is learnable bias.

Each task-specific vector goes through the respective softmax layer to determine the output of each task with the highest predicted probability. The prediction output ($pred_i$)

of task i is given in Eq. (5.4).

$$\begin{aligned} pred_i &= \text{argmax}(\mathbf{prob}_i) \\ \mathbf{prob}_i &= \text{softmax}(\mathbf{W}_p \mathbf{task}_i + b_p) \\ \text{softmax}(z_j) &= \frac{\exp(z_j)}{\sum_{c=1}^{nlabels_i} \exp(z_c)} \end{aligned} \tag{5.4}$$

where $\mathbf{prob}_i$ contains the predicted probabilities of $nlabels_i$ possible outputs of task i; $\mathbf{W}_p$ is learnable weights and b_p is learnable bias.

Training DeepCVA. To compare DeepCVA's outputs with ground-truth CVSS labels, we define a multi-task loss that averages the cross-entropy losses of seven tasks in Eq. (5.5).

$$\begin{aligned} loss_{DeepCVA} &= \sum_{i=1}^{7} loss_i \\ loss_i &= -\sum_{c=1}^{nlabels_i} y_i^c \log(prob_i^c),\ y_i^c = 1 \text{ if } c \text{ is true class else } 0 \end{aligned} \tag{5.5}$$

where y_i^c, $prob_i^c$, and $nlabels_i$ are the ground-truth value, predicted probability and all labels of CVSS task i, respectively.

We minimize this multi-task loss using a stochastic gradient descent method [342] to optimize the weights of learnable components in DeepCVA. We also use backpropagation [343] to automate partial differentiation with chain-rule and increase the efficiency of gradient computation throughout the model.

5.4 Experimental Design and Setup

All the experiments ran on a computing cluster that has 16 CPU cores with 16GB of RAM and Tesla V100 GPU.

5.4.1 Datasets

To develop commit-level SV assessment models, we built a dataset of Vulnerability-Contributing Commits (VCCs) and their CVSS metrics. We used Vulnerability-Fixing Commits (VFCs) to retrieve VCCs, as discussed in section 5.2.1.

VFC identification. We first obtained VFCs from three public sources: NVD [16], GitHub and its Advisory Database[2] as well as a manually curated/verified VFC dataset (VulasDB) [285]. In total, we gathered 13,310 VFCs that had dates ranging from July 2000 to October 2020. We selected VFCs in Java projects as Java has been commonly investigated in the literature (e.g., [287, 288, 286]) and also in the top five most popular languages in practice.[3] Following the practice of [286], we discarded VFCs that had more than 100 files and 10,000 lines of code to reduce noise in the data.

VCC identification with the SZZ algorithm. After the filtering steps, we had 1,602 remaining unique VFCs to identify VCCs using the SZZ algorithm [344]. This algorithm selects commits that last modified the source code lines deleted or modified to address an SV in a VFC as the respective VCCs of the same SV (see Fig. 5.1). As in [344], we first discarded commits with timestamps after the published dates of the respective SVs on NVD since SVs can only be reported after they were injected in a codebase. We

[2] https://github.com/advisories

[3] https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved

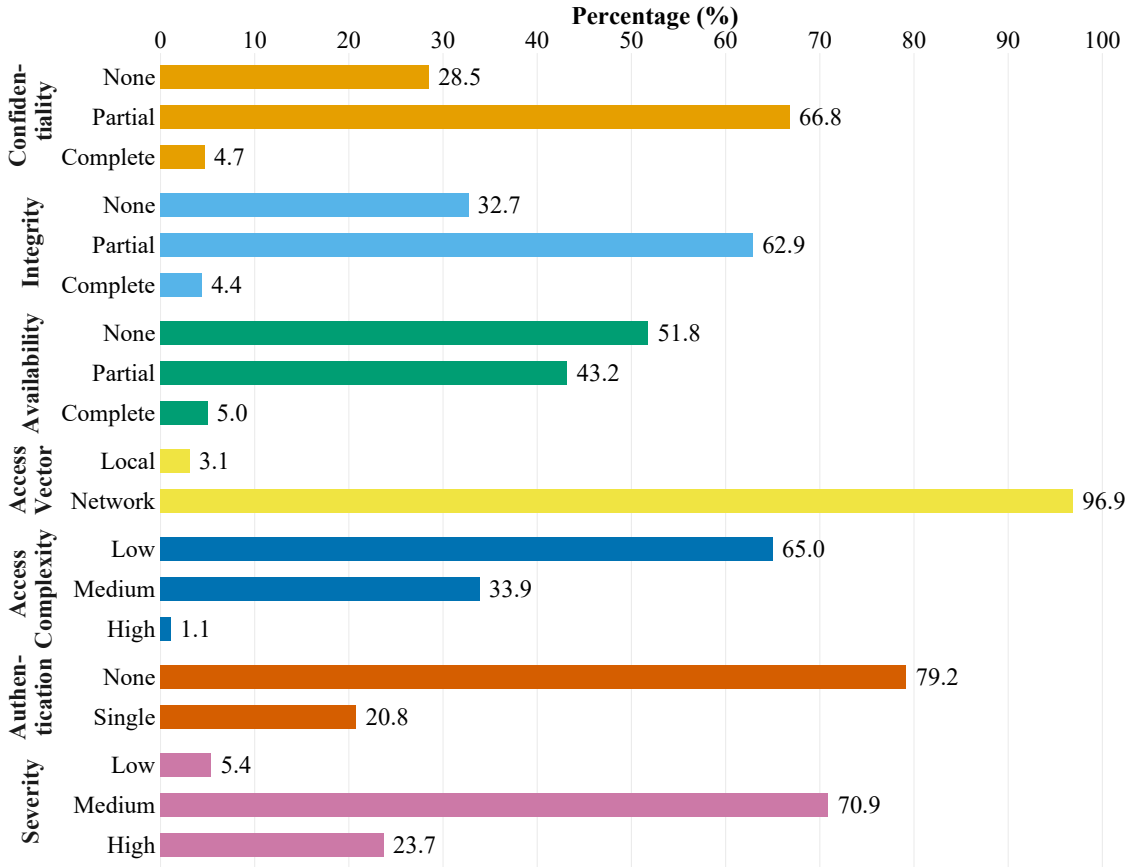


FIGURE 5.4: Class distributions of seven SV assessment tasks.

then removed cosmetic changes (e.g., newlines and white spaces) and single-line/multi-line comments in VFCs since these elements do not change code functionality [286]. Like [286], we also considered copied or renamed files while tracing VCCs. We obtained **1,229 unique VCCs**[4] of 542 SVs in 246 real-world Java projects and their corresponding expert-verified CVSS metrics on NVD. Distributions of curated CVSS metrics are illustrated in Fig. 5.4. The details of the number of commits and projects retained in each filtering step are also given in Table 5.1. Note that some commits and projects were removed during the tracing of VCCs from VFCs due to the issues coined as ghost commits studied by Rezk et al. [290]. We did not remove large VCCs (with more than 100 files and 10k lines) as we found several VCCs were large initial/first commits. Our observations agreed with the findings of Meneely et al. [25].

Manual VCC validation. To validate our curated VCCs, we randomly selected 293 samples, i.e., 95% confidence level and 5% error [291], for two researchers (i.e., the author of this thesis and a PhD student with three-year experience in Software Engineering and Cybersecurity) to independently examine. The manual VCC validation was considerably labor-intensive, which took approximately 120 man-hours. The Cohen's kappa (κ) inter-rater reliability score [345] was 0.83, i.e., "almost perfect" agreement [346]. We also involved another PhD student having two years of experience in Software Engineering and Cybersecurity in the discussion to resolve disagreements. Our validation found that 85% of the VCCs were valid. In fact, the SZZ algorithm is imperfect [347], but we assert that it is nearly impossible to obtain near 100% accuracy without exhaustive manual validation. Specifically, the main source of incorrectly identified VCCs in our dataset was that some

[4]The SV reports of all curated VCCs were not available at commit time.

TABLE 5.1: The number of commits and projects after each filtering step.

No.	Filtering step	No. of commits	No. of projects
1	All unfiltered VFCs	13,310	2,864
2	Removing duplicate VFCs	9,989	2,864
3	Removing non-Java VFCs	1,607	361
4	Removing VFCs with more than 100 files & 10k lines	1,602	358
5	Tracing VCCs from VFCs using the SZZ algorithm	3,742	342
6	Removing VCCs with null characteristics (CVSS values)	2,271	246
7	Removing duplicate VCCs	1,229	246

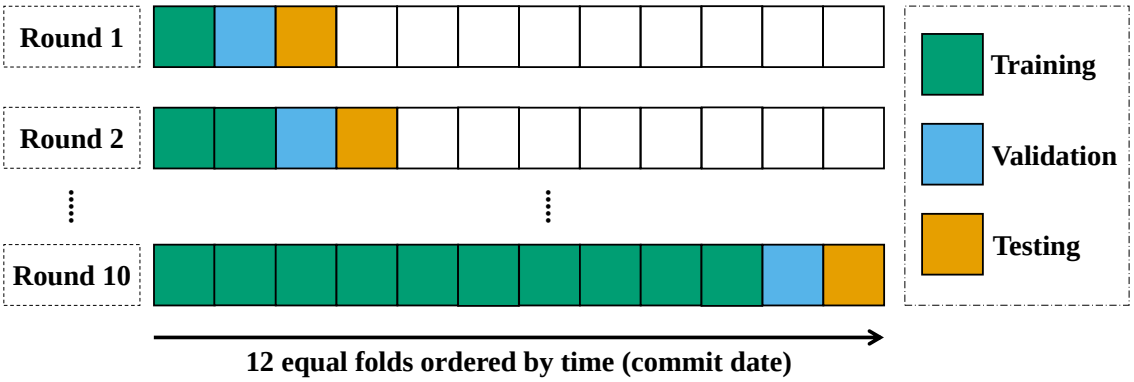


FIGURE 5.5: Time-based splits for training, validating & testing.

files in VFCs were used to update version/documentation or address another issue instead of fixing an SV. One such false positive VCC was the commit *87c89f0* in the *jspwiki* project that last modified the build version in the corresponding VFC.

Data splitting. We adopted *time-based* splits [348] for training, validating and testing the models to closely represent real-world scenarios where incoming/future unseen data is *not* present during training [286, 349]. We trained, validated and tested the models in 10 rounds using 12 equal folds split based on commit dates (see Fig. 5.5). Specifically, in round i, folds $1 \rightarrow i$, $i+1$ and $i+2$ were used for training, validation and testing, respectively. We chose an optimal model with the highest average *validation* performance and then reported its respective average testing performance over 10 rounds, which helped avoid unstable results of a single testing set [212].

5.4.2 Evaluation Metrics

To evaluate the performance of automated commit-level SV assessment, we utilized the F1-Score and Matthews Correlation Coefficient (MCC) metrics that have been commonly used in the literature (e.g., [21, 22, 349]). These two metrics are suitable for the imbalanced classes [307] in our data (see Fig. 5.4). F1-Score has a range from 0 to 1, while MCC takes values from –1 to 1, where 1 is the best value for both metrics. MCC was used to select optimal models since MCC explicitly considers all classes [307]. To evaluate the tasks with more than two classes, we used macro F1-Score [22] and the multi-class version of MCC [350]. MCC of the multi-task DeepCVA model was the average MCC of seven constituent tasks. Note that MCC is not directly proportional to F1-score.

5.4.3 Hyperparameter and Training Settings of DeepCVA

Hyperparameter settings. We used the average *validation* MCC to select optimal hyperparameters for DeepCVA's components. We also ran DeepCVA 10 times each round to reduce the impact of random initialization on model performance. We first chose 1024 for the input length of the pre-/post-change hunks/context (see Fig. 5.2), which has been commonly used in the literature (e.g., [80, 351]). Using a shorter input length would likely miss many code tokens, while a longer length would significantly increase the model complexity and training time. Shorter commits were padded with zeros, and longer ones were truncated to ensure the same input size for parallelization with GPU [287, 337]. We built a vocabulary of 10k most frequent code tokens in the Input Embedding layer as suggested by [352]. Note that using 20k-sized vocabulary only raised the performance by 2%, yet increased the model complexity by nearly two times. We selected an input embedding size of 300, i.e., a standard and usually high limit value for many embedding models (e.g., [202, 196]), and we randomly initialized embedding vectors [287, 120]. For the number of filters of the three-way CNN as well as the hidden units of the GRU, Attention and Task-specific blocks, we tried {32, 64, 128}, similar to [21]. We picked 128 as it had at least 5% better validation performance than 32 and 64.
Training settings. We used the Adam algorithm [353], the state-of-the-art stochastic gradient descent method, for training DeepCVA end-to-end with a learning rate of 0.001 and a batch size of 32 as recommended by Hoang et al. [287]. To increase the training stability, we employed Dropout [354] with a dropout rate of 0.2 and Batch Normalization [355] between layers. We trained DeepCVA for 50 epochs, and we would stop training if the *validation* MCC did not change in the last five epochs to avoid overfitting [287, 337].

5.4.4 Baseline Models

We considered three types of learning-based baselines for automated commit-level SV assessment, as learning-based models can automatically extract relevant SV patterns/features from input data for prediction without relying on pre-defined rules. The baselines were (*i*) **S-CVA**: **S**upervised single-task model using either software metrics or text-based features including Bag-of-Words (BoW or token count) and Word2vec [202]; (*ii*) **X-CVA**: supervised e**X**treme multi-class model that performed a single prediction for all seven tasks using the above feature types; and (*iii*) **U-CVA**: **U**nsupervised model using k-means clustering [356] with the same features as S-CVA/X-CVA. Note that there was no existing technique for automating commit-level SV assessment, so we could only compare DeepCVA with the compatible techniques proposed for related tasks, as described hereafter.

Software metrics (e.g., [26, 329, 330]) and text-based features (BoW/Word2vec) (e.g., [339, 357]) have been widely used for commit-level prediction. We used 84 software metrics proposed by [26, 329, 330] for defect/SV prediction. Among these metrics, we converted C/C++ keywords into Java ones to match the language used in our dataset. The list of software metrics used in this chapter can be found at `https://github.com/lhmtriet/DeepCVA`. As in [26], in each round in Fig. 5.5, we also removed correlated software metrics that had a Spearman correlation larger than 0.7 based on the training data of that round to avoid performance degradation, e.g., no. of *stars* vs. *forks* of a project. For BoW and Word2vec, we adopted the same vocabulary size of 10k to extract features from four inputs described in Fig. 5.2, as in DeepCVA. Feature vectors of all inputs were concatenated into a single vector. For Word2vec, we averaged the vectors of all tokens in an input to generate its feature vector, which has been shown to be a strong baseline [298]. Like DeepCVA, we also used an embedding size of 300 for each Word2vec token.

Using these feature types, S-CVA trained a separate supervised model for each CVSS task, while X-CVA used a single multi-class model to predict all seven tasks simultaneously.

X-CVA worked by concatenating all seven CVSS metrics into a single label. To extract the results of the individual tasks for X-CVA, we checked whether the ground-truth label of each task was in the concatenated model output. For S-CVA and X-CVA, we applied six popular classifiers: Logistic Regression (LR), Support Vector Machine (SVM), K-Nearest Neighbors (KNN), Random Forest (RF), XGBoost (XGB) [78] and Light Gradient Boosting Machine (LGBM) [79]. These classifiers have been used for SV assessment based on SV reports [22, 23]. The hyperparameters for tuning these classifiers were *regularization*: {l1, l2}; *regularization coefficient*: {0.01, 0.1, 1, 10, 100} for LR and {0.01, 0.1, 1, 10, 100, 1,000, 10,000} for SVM; *no. of neighbors*: {11, 31, 51}, *distance norm*: {1, 2} and *distance weight*: {uniform, distance} for KNN; *no. of estimators*: {100, 300, 500}, *max. depth*: {3, 5, 7, 9, unlimited}, *max. no. of leaf nodes*: {100, 200, 300, unlimited} for RF, XGB and LGBM. These hyperparameters have been adapted from relevant studies [22, 23, 304].

Unlike S-CVA and X-CVA, U-CVA did not require CVSS labels to operate; therefore, U-CVA required less human effort than S-CVA and X-CVA. We tuned U-CVA for each task with the following no. of clusters (k): {2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 35, 40, 45, 50}. To assess a new commit with U-CVA, we found the cluster with the smallest Euclidean distance to that commit and assigned it the most frequent class of each task in the selected cluster.

5.5 Research Questions and Experimental Results

5.5.1 RQ1: How does DeepCVA Perform Compared to Baseline Models for Commit-level SV Assessment?

Motivation. We posit the need for commit-level Software Vulnerability (SV) assessment tasks based on seven CVSS metrics. Such tasks would help developers to understand the SV exploitability and impacts as early as SVs are introduced in a software system and devise remediation plans accordingly. RQ1 evaluates our DeepCVA for this new and important task.

Method. We compared the effectiveness of our DeepCVA model with the S-CVA, X-CVA and U-CVA baselines (see section 5.4.4) on the *testing* sets. We trained, validated and tested the models using the time-based splits, as described in section 5.4.1. Because of the inherent randomness of GPU-based implementation of DeepCVA,[5] we ran DeepCVA 10 times in each round and then averaged its performance. The baselines were not affected by this issue as they did not use GPU. For DeepCVA, we used the hyperparameter/training settings in section 5.4.3. For each type of baseline, we used grid search on the hyperparameters given in section 5.4.4 to find the optimal model with the highest *validation* MCC (see section 5.4.2).

Results. ***DeepCVA outperformed all baselines***[6] ***(X-CVA, S-CVA and U-CVA) in terms of both MCC and F1-Score***[7] ***for all seven tasks (see Table 5.2).*** DeepCVA got average and best MCC values of 0.247 and 0.286, i.e., 38% and 59.8% better than the second-best baseline (X-CVA with Word2vec features), respectively. Task-wise, DeepCVA had 11.2%, 42%, 42.2%, 20.6%, 69.2%, 24.8% and 39.2% higher MCC than the best respective baseline models for Confidentiality, Integrity, Availability, Access Vector, Access Complexity, Authentication and Severity tasks, respectively. Notably, the best DeepCVA model achieved stronger performance than all baselines with MCC percentage gaps from 24.1% (Confidentiality) to 82.5% (Access Complexity).

[5] https://keras.io/getting_started/faq/#how-can-i-obtain-reproducible-results-using-keras-during-development

[6] MCC values of *random* and *most-frequent-class* baselines were all < 0.01.

[7] Precision (0.533)/Recall (0.445) of DeepCVA were $>$ than all baselines.

TABLE 5.2: Testing performance of DeepCVA and baseline models. **Notes**: Optimal classifiers of S-CVA/X-CVA and optimal cluster no. (k) of U-CVA are in parentheses. BoW, W2V and SM are Bag-of-Words, Word2vec and Software Metrics, respectively. The best performance of DeepCVA is from the run with the highest MCC in each round. Best row-wise values are in grey.

CVSS metric	Evaluation metric	Model									
		S-CVA			X-CVA			U-CVA			DeepCVA (Best in parentheses)
		BoW	W2V	SM	BoW	W2V	SM	BoW	W2V	SM	
Confidentiality	**F1-Score**	0.416	0.406	0.423	0.420	0.434	0.429	0.292	0.332	0.313	**0.436 (0.475)**
	MCC	0.174 (LR)	0.239 (LGBM)	0.232 (XGB)	0.188 (LR)	0.241 (LR)	0.203 (XGB)	0.003 (50)	0.092 (45)	0.017 (50)	**0.268 (0.299)**
Integrity	**F1-Score**	0.373	0.369	0.352	0.391	0.415	0.407	0.284	0.305	0.330	**0.430 (0.458)**
	MCC	0.127 (LGBM)	0.176 (LGBM)	0.146 (RF)	0.114 (LGBM)	0.160 (LR)	0.128 (LGBM)	-0.005 (25)	0.091 (30)	0.084 (25)	**0.250 (0.295)**
Availability	**F1-Score**	0.381	0.389	0.384	0.424	0.422	0.406	0.254	0.332	0.238	**0.432 (0.475)**
	MCC	0.182 (RF)	0.173 (LGBM)	0.126 (XGB)	0.187 (LR)	0.192 (LR)	0.123 (XGB)	0.064 (10)	0.092 (45)	0.016 (3)	**0.273 (0.303)**
Access Vector	**F1-Score**	0.511	0.487	0.440	0.499	0.532	0.487	0.477	0.477	0.477	**0.554 (0.578)**
	MCC	0.07 (XGB)	0.051 (LR)	0.018 (LR)	0.044 (LGBM)	0.107 (LR)	0.012 (LGBM)	0.000 (9)	0.000 (40)	0.000 (6)	**0.129 (0.178)**
Access Complexity	**F1-Score**	0.437	0.448	0.417	0.412	0.445	0.361	0.315	0.365	0.385	**0.464 (0.475)**
	MCC	0.119 (LR)	0.143 (XGB)	0.111 (LGBM)	0.131 (LR)	0.121 (XGB)	0.088 (SVM)	0.000 (4)	0.022 (30)	0.119 (15)	**0.242 (0.261)**
Authentication	**F1-Score**	0.601	0.584	0.593	0.541	0.618	0.586	0.458	0.526	0.492	**0.657 (0.677)**
	MCC	0.258 (SVM)	0.264 (XGB)	0.268 (LGBM)	0.212 (RF)	0.282 (SVM)	0.208 (XGB)	0.062 (50)	0.162 (30)	0.089 (50)	**0.352 (0.388)**
Severity	**F1-Score**	0.407	0.357	0.345	0.382	0.381	0.358	0.283	0.288	0.287	**0.424 (0.460)**
	MCC	0.144 (LR)	0.153 (XGB)	0.057 (XGB)	0.130 (LR)	0.149 (LGBM)	0.058 (XGB)	-0.018 (4)	0.010 (15)	0.026 (4)	**0.213 (0.277)**
Average	**F1-Score**	0.447	0.434	0.422	0.438	0.464	0.433	0.338	0.375	0.360	**0.485 (0.514)**
	MCC	0.153	0.171	0.137	0.144	0.179	0.117	0.015	0.067	0.050	**0.247 (0.286)**

The average and task-wise F1-Score values of DeepCVA also beat those of the best baseline (X-CVA with Word2vec features) by substantial margins. We found that DeepCVA significantly outperformed the best baseline models in terms of both MCC and F1-score averaging across all seven tasks, confirmed with p-values < 0.01 using the non-parametric Wilcoxon signed-rank tests [254]. These results show the effectiveness of the novel design of DeepCVA.

An example to qualitatively demonstrate the effectiveness of DeepCVA is the VCC *ff655ba* in the *Apache xerces2-j* project, in which a hashing algorithm was added. This algorithm was later found vulnerable to hashing collision that could be exploited with timing attacks in the fixing commit *992b5d9*. This SV was caused by the order of items being added to the hash table in the `put(String key, int value)` function. Such an order could not be easily captured by baseline models whose features did not consider the sequential nature of code (i.e., BoW, Word2vec and software metrics) [203]. More details about the contributions of different components to the overall performance of DeepCVA are covered in section 5.5.2.

Regarding the baselines, the average MCC value (0.147) of X-CVA was on par with that (0.154) of S-CVA. This result reinforces the benefits of leveraging the common attributes among seven CVSS metrics to develop effective commit-level SV assessment models. However, X-CVA was still not as strong as DeepCVA mainly because of its much lower training data utilization per output. For X-CVA, there was an average of 39 output combinations of CVSS metrics in the training folds, i.e., 31 commits per output. In contrast, DeepCVA had 13.2 times more data per output as there were at most three classes for each task (see Fig. 5.4). Finally, we found supervised learning (S-CVA, X-CVA and DeepCVA) to be at least 74.6% more effective than the unsupervised approach (U-CVA). This result shows the usefulness of using CVSS metrics to guide the extraction of commit features.

5.5.2 RQ2: What are the Contributions of the Main Components in DeepCVA to Model Performance?

Motivation. We have shown in RQ1 that DeepCVA significantly outperformed all the baselines for seven commit-level SV assessment tasks. RQ2 aims to give insights into the contributions of the key components to such a strong performance of DeepCVA. Such insights can help researchers and practitioners to build effective SV assessment models.
Method. We evaluated the performance contributions of the main components of DeepCVA: (*i*) Closest Enclosing Scope (CES) of code changes, (*ii*) CNN filter size, (*iii*) Three-way CNN, (*iv*) Attention-based GRU, (*v*) Attention mechanism, (*vii*) Task-specific blocks and (*vi*) Multi-task learning. For each component, we first removed it from DeepCVA, retrained the model variant and reported its *testing* result. When we removed Attention-based GRU, we used max-pooling [287, 120] after the three-way CNN to generate the commit vector. When we removed Multi-task learning, we trained a separate model for each of the seven CVSS metrics. We also investigated an Abstract Syntax Tree (AST) variant of DeepCVA, in which we complemented input code tokens with their syntax (e.g., `int a = 1` is a `VariableDeclarationStatement`, where `a` is an `Identifier` and `1` is a `NumberLiteral`). This AST-based variant explored the usefulness of syntactical information for commit-level SV assessment. We extracted the nodes in an AST that contained code changes and their CES. If more than two nodes contained the code of interest, we chose the one at a lower depth in the AST. We then flattened the nodes with depth-first traversal for feature extraction [321].

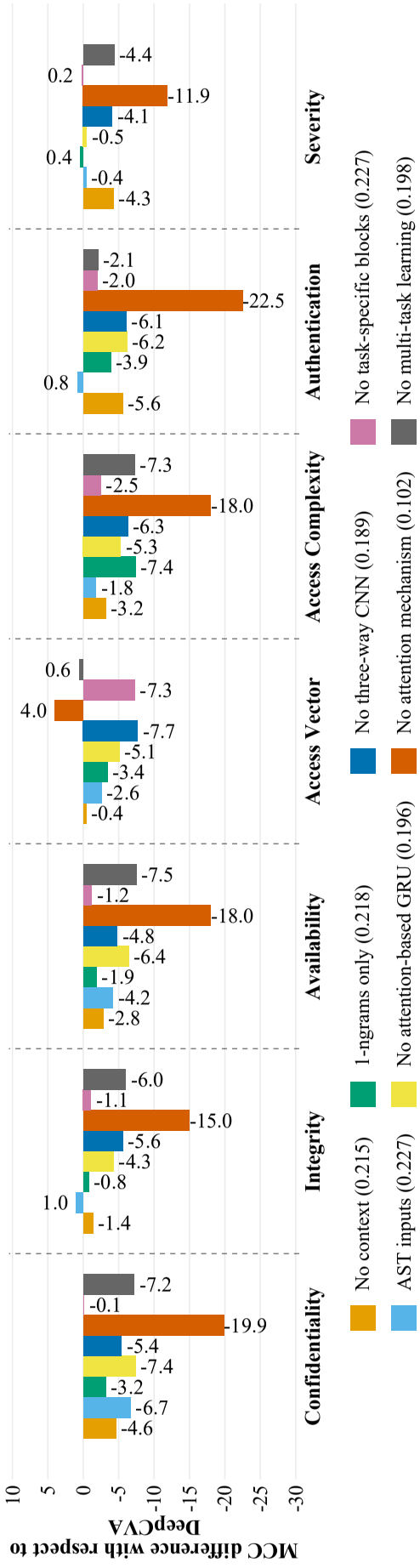


FIGURE 5.6: Differences of testing MCC (multiplied by 100 for readability) of the model variants compared to the proposed DeepCVA in section 5.3. **Note**: The average MCC values (without multiplying by 100) of the model variants are in parentheses.

Results. ***As depicted in Fig. 5.6, the main components[8] uplifted the average MCC of DeepCVA by 25.9% for seven tasks***. Note that 7/8 model variants (except the model with no attention mechanism) outperformed the best baseline model from RQ1. These results were confirmed with p-values < 0.01 using Wilcoxon signed-rank tests [254]. Specifically, the components[8] of DeepCVA increased the MCC values by 25.3%, 20.8%, 21.5%, 35.8%, 35.5%, 18.9% and 23.6% for Confidentiality, Integrity, Availability, Access Vector, Access Complexity, Authentication and Severity, respectively.

For the inputs, using the Smallest Enclosing Scope (CES) of code changes resulted in a 14.8% increase in MCC compared to using hunks only, while using AST inputs had 8.8% lower performance. This finding suggests that code context is important for assessing SVs in commits. In contrast, syntactical information is not as necessary since code structure can be implicitly captured by code tokens and their sequential order using our AC-GRU.

The key components of the AC-GRU feature extractor boosted the performance by 13.2% (3-grams vs. 1-grams), 25.6% (Attention-based GRU), 30.2% (Three-way CNN) and 142% (Attention). Note that DeepCVA surpassed the state-of-the-art 3-gram [21] and 1-gram [287] CNN-only architectures for (commit-level) SV/defect prediction. These results show the importance of combining the (1,3,5)-gram three-way CNN with attention-based GRUs rather than using them individually. We also found that 1-5 grams did not significantly increase the performance (p-value = 0.186), confirming our decision in section 5.3.2 to only use 1,3,5-sized filters.

For the prediction layers, we raised 8.8% and 24.4% MCC of DeepCVA with Task-specific blocks and Multi-task learning, respectively. Multi-task DeepCVA took 8,988 s (2.5 hours) and 25.7 s to train/validate and test in 10 rounds × 10 runs, which were 6.3 and 6.2 times faster compared to those of seven single-task DeepCVA models, respectively. DeepCVA was only 11.3% and 12.7% slower in training/validating and testing than one single-task model on average, respectively. These values highlight the efficiency of training and maintaining the multi-task DeepCVA model. Finally, obtaining Severity using the CVSS formula [22] from the predicted values of the other six metrics dropped MCC by 17.4% for this task. This result supports predicting Severity directly from commit data.

5.5.3 RQ3: What are the Effects of Class Rebalancing Techniques on Model Performance?

Motivation. Recent studies (e.g., [313, 358]) have shown that class rebalancing techniques (i.e., equalizing the class distributions in the training set) can improve model effectiveness for defect/SV prediction. However, these rebalancing techniques can only be applied to single-task models, not multi-task ones. The reason is that each task has a unique class distribution (see Fig. 5.4), and thus balancing class distribution of one task will not balance classes of the others. RQ3 is important to test whether multi-task DeepCVA still outperforms single-task baselines in RQ1/RQ2 using rebalancing techniques.

Method. We compared the *testing* performance of multi-task DeepCVA with baselines in RQ1/RQ2 using two popular oversampling techniques [313]: *Random OverSampling* (*ROS*) and *SMOTE* [359]. ROS randomly duplicates the existing samples of minority classes, while SMOTE randomly generates synthetic samples between the existing minority-class samples and their nearest neighbor(s) based on Euclidean distance. We did not consider undersampling, as such models performed poorly because of some very small minority classes (e.g., *Low* Access Complexity had only 14 samples). We applied ROS and SMOTE to *only the training set* and then optimized all baseline models again. Like [313], we also tuned SMOTE using grid search with different values of nearest neighbors: {1, 5, 10, 15,

[8] We excluded the DeepCVA variant with no attention mechanism as its performance was abnormally low, affecting the overall trend of other variants.

TABLE 5.3: Testing performance (MCC) of optimal baselines using oversampling techniques and multi-task DeepCVA. **Note**: †denotes that the oversampled models outperformed the non-oversampled one reported in RQ1/RQ2.

CVSS Task	S-CVA (ROS)	S-CVA (SMOTE)	X-CVA (ROS)	Single-task DeepCVA (ROS)	Multi-task DeepCVA
Confidentiality	0.220	0.203	0.185	0.250†	**0.268**
Integrity	0.174	0.168	0.179†	0.206†	**0.250**
Availability	0.195†	0.187†	0.182	0.209†	**0.273**
Access Vector	0.115†	0.110†	0.092	**0.156**†	0.129
Access Comp.	0.172†	0.186†	0.144†	0.190†	**0.242**
Authentication	0.325†	0.340†	0.299†	0.318	**0.352**
Severity	0.132	0.124	0.141	0.186†	**0.213**
Average	0.190†	0.188†	0.175	0.216†	**0.247**

20}. We could not apply SMOTE to single-task DeepCVA as features were trained end-to-end and unavailable prior training for finding nearest neighbors. We also did not apply SMOTE to X-CVA as there was always a single-sample class in each round, producing no nearest neighbor.

Results. ***ROS and SMOTE increased the average performance (MCC) of 3/4 baselines except X-CVA (see Table 5.3). However, the average MCC of our multi-task DeepCVA was still 14.4% higher than that of the best oversampling-augmented baseline (single-task DeepCVA with ROS)***. Overall, MCC increased by 8%, 6.9% and 9.1% for S-CVA (ROS), S-CVA (SMOTE) and single-task DeepCVA (ROS), respectively. These improvements were confirmed significant with p-values < 0.01 using Wilcoxon signed-rank tests [254]. We did not report oversampling results of U-CVA as they were still much worse compared to others. We found single-task DeepCVA benefited the most from oversampling, probably since Deep Learning usually performs better with more data [274]. In contrast, oversampling did not improve X-CVA as oversampling did not generate as many samples for X-CVA per class as for S-CVA (i.e., X-CVA had 13 times, on average, more classes than S-CVA). These results further strengthen the effectiveness and efficiency of multi-task learning of DeepCVA for commit-level SV assessment even without the overheads of rebalancing/oversampling data.

5.6 Discussion

5.6.1 DeepCVA and Beyond

DeepCVA has been shown to be effective for commit-level SV assessment in the three RQs, but our model still has false positives. We analyze several representative patterns of such false positives to help further advance this task and solutions for researchers and practitioners in the future.

Some commits were too complex and large (tangled) to be assessed correctly. For example, the VCC *015f7ef* in the *Apache Spark* project contained 1,820 additions and 146 deletions across 29 files; whereas, the untrusted deserialization SV occurred in just one line 56 in `LauncherConnection.java`. Recent techniques (e.g., [279, 360]) pinpoint more precise locations (e.g., individual files or lines in commits) of defects, especially in tangled changes. Such techniques can be adapted to remove irrelevant code in VCCs (i.e., changes

that do not introduce or contain SVs). More relevant code potentially gives more fine-grained information for the SV assessment tasks. Note that DeepCVA provides a strong baseline for comparing against fine-grained approaches.

DeepCVA also struggled to accurately predict assessment metrics for SVs related to external libraries. For instance, the SV in the commit *015f7ef* above occurs with the `ObjectInputStream` class from the `java.io` package, which sometimes prevented DeepCVA from correctly assessing an SV. If an SV happens frequently with a package in the training set, (e.g., the XML library of the VCC *bba4bc2* in Fig. 5.1), DeepCVA still can infer correct CVSS metrics. Pre-trained code models on large corpora [288, 80, 302] along with methods to search/generate code [361] and documentation [362] as well as (SV-related) information from developer Q&A forums [315] can be investigated to provide enriched context of external libraries, which would in turn support more reliable commit-level SV assessment with DeepCVA.

We also observed that DeepCVA, alongside the considered baseline models, performed significantly worse, in terms of MCC, for Access Vector compared to the remaining tasks (see Table 5.2). We speculate that such low performance is mainly because Access Vector contains the most significant class imbalance among the tasks, as shown in Fig. 5.4. For single-task models, we found that using class rebalancing techniques such as ROS or SMOTE can help improve the performance, as demonstrated in RQ3 (see section 5.5.3). However, it is still unclear how to apply the current class rebalancing techniques for multi-task learning models such as DeepCVA. Thus, we suggest that more future work should investigate specific class rebalancing and/or data augmentation to address such imbalanced data in the context of multi-task learning.

5.6.2 Threats to Validity

The first threat is the collection of VCCs. We followed the practices in the literature to reduce the false positives of the SZZ algorithm. We further mitigated this threat by performing independent manual validation with three researchers with at least two years of experience in Software Engineering and Cybersecurity.

Another concern is the potential suboptimal tuning of baselines and DeepCVA. However, it is impossible to try the entire hyperparameter space within a reasonable amount of time. For the baseline models, we lessened this threat by using a wide range of hyperparameters from the previous studies to reoptimize these models from scratch on our data. For DeepCVA, we adapted the best practices recommended in the relevant literature to our tasks.

The reliability and generalizability of our findings are also potential threats. We ran DeepCVA 10 times to mitigate the experimental randomness. We confirmed our results using non-parametric statistical tests with a confidence level $> 99\%$. Our results may not generalize to all software projects. However, we reduced this threat by conducting extensive experiments on 200+ real-world projects of different scales and domains.

5.7 Related Work

5.7.1 Data-Driven SV Prediction and Assessment

As reviewed in Chapter 2, many studies have developed data-driven approaches that can harness large-scale SV data from public security databases like NVD to determine different characteristics of SVs. Specifically, SV information on NVD has been utilized to infer the types [156], exploitation [32, 34], time-to-exploit [32] and various CVSS assessment metrics [23, 22, 100, 103] of SVs. Other studies [327, 363] have leveraged code patterns

in fixing commits of third-party libraries to assess SVs in such libraries. Our work is fundamentally different from these previous studies since we are the first to investigate the potential of performing assessment of all SV types (not only vulnerable libraries) using commit changes rather than bug/SV reports/fixes. Our approach allows practitioners to realize the exploitability/impacts of SVs in their systems much earlier, e.g., up to 1,000 days before (see section 5.2.2), as compared to using bug/SV reports/fixes. Less delay in SV assessment helps practitioners to plan/prioritize SV fixing with fresh design and implementation in their minds. Moreover, we have shown that multi-task learning, i.e., predicting all CVSS metrics simultaneously, can significantly increase the effectiveness and reduce the model development and maintenance efforts in commit-level SV assessment. It should be noted that report-level prediction is still necessary for assessing SVs in third-party libraries/software, especially the ones without available code (commits), to prioritize vendor-provided patch application, as well as SVs missed by commit-level detection.

5.7.2 SV Analytics in Code Changes

Commit-level prediction (e.g., [26, 337, 364]) has been explored to provide *just-in-time* information for developers about code issues, but such studies mainly focused on generic software defects. However, SV is a special type of defects [365] that can threaten the security properties of a software project. Thus, SV requires special treatment [366] and domain knowledge [367]. Meneely et al. [25] and Bosu et al. [368] conducted in-depth studies on how code and developer metrics affected the introduction and review of VCCs. Besides analyzing the characteristics of VCCs, other studies [329, 330, 369] also developed commit-level SV detection models that leveraged software and text-based metrics. Different from the previous studies that have detected VCCs, we focus on the assessment of such VCCs. SV assessment is as important as the detection step since assessment metrics help early plan and prioritize remediation for the identified SVs. It is worth noting that the existing SV detection techniques can be used to flag VCCs that would then be assessed by our DeepCVA model.

5.8 Chapter Summary

We introduce DeepCVA, a novel deep multi-task learning model, to tackle a new task of commit-level SV assessment. DeepCVA promptly informs practitioners about the CVSS severity level, exploitability, and impact of SVs in code changes after they are committed, enabling more timely and informed remediation. DeepCVA substantially outperformed many baselines (even the ones enhanced with rebalanced data) for the seven commit-level SV assessment tasks. Notably, multi-task learning utilizing the relationship of assessment tasks helped our model be 24.4% more effective and 6.3 times more efficient than single-task models. With the reported performance, DeepCVA realizes the first promising step towards a holistic solution to assessing SVs as early as they appear.

Chapter 6

Collection and Analysis of Developers' Software Vulnerability Concerns on Question and Answer Websites

Related publications: This chapter is based on two of our papers: (1) "*PUMiner: Mining Security Posts from Developer Question and Answer Websites with PU Learning*" published in the 17th International Conference on Mining Software Repositories (MSR), 2020 (CORE A) [304], and (2) "*A Large-scale Study of Security Vulnerability Support on Developer Q&A Websites*" published in the 25th International Conference on Evaluation and Assessment in Software Engineering (EASE), 2021 (CORE A) [315].

In the previous Chapters 3, 4, and 5, we have proposed different automated solutions to perform Software Vulnerability (SV) assessment using the knowledge gathered from software artifacts, i.e., SV reports and source code. In practice, besides relying on these software artifacts for SV assessment, developers also seek information about SVs, e.g., experience/solutions of fixing similar SVs, on developer Question and Answer (Q&A) websites. The SV information provided on Q&A sites also sheds light on developers' real-world concerns/challenges with addressing SVs, which can complement other assessment metrics like CVSS to support better SV understanding and fixing. However, there is still little known about these SV-specific discussions on different Q&A sites. Chapter 6 presents a large-scale empirical study to understand developers' SV discussions and how these discussions are being supported by Q&A sites. We curate 71,329 SV posts from two large Q&A sites, namely Stack Overflow (SO) and Security StackExchange (SSE), and then use topic modeling to uncover the key developers' SV topics of concern. We then analyze the popularity, difficulty, and level of expertise for each topic. We also perform a qualitative analysis to identify the types of solutions to SV-related questions. We identify 13 main SV discussion topics. Many topics do not follow the distributions and trends in expert-based security sources, e.g., Common Weakness Enumeration (CWE) and Open Web Application Security Project (OWASP). We also discover that SV discussions attract more experts to answer than many other domains. Nevertheless, some difficult SV topics/types still receive quite limited support from experts, which may suggest it is challenging to fix them in practice. Moreover, we identify seven key types of answers given to SV questions, in which SO often provides code and instructions, while SSE usually gives experience-based advice and explanations. These solutions on Q&A sites may be reused to fix similar SVs, reducing the SV fixing effort. Overall, the findings of this chapter enable researchers and practitioners to effectively leverage SV knowledge on Q&A sites for (data-driven) SV assessment.

6.1 Introduction

It is important to constantly track and resolve Software Vulnerabilities (SVs) to ensure the availability, confidentiality and integrity of software systems [3]. Developers can seek assessment information for resolving SVs from sources verified by security experts such as Common Weakness Enumeration (CWE) [28], National Vulnerability Database (NVD) [16] and Open Web Application Security Project (OWASP) [167]. However, these expert-based SV sources do not provide any mechanisms for developers to promptly ask and answer questions about issues in implementing/understanding the reported SV solutions/concepts. On the other hand, developer Questions and Answer (Q&A) websites contain a plethora of such SV-related discussions. Stack Overflow (SO)[1] and Security StackExchange (SSE)[2] contain some of the largest numbers of SV-related discussions among developer Q&A sites with contributions from millions of users [304].

The literature has analyzed different aspects of discussions on Q&A sites, but there is still no investigation of how SO and SSE are supporting SV-related discussions. Specifically, the main concepts [370], the top languages/technologies and user demographics [371], as well as user perceptions and interactions [372] of general security discussions on SO have been studied. However, from our analysis (see section 6.3.2), only about 20% of the available SV posts on SO were investigated in the previous studies, limiting a thorough understanding of SV topics (developers' concerns when tackling SVs in practice) on Q&A sites. Moreover, the prior studies only focused on SO, and little insight has been given into the support of SV discussions on different Q&A sites. Such insight would potentially affect the use of a suitable site (e.g., SO vs. SSE) to obtain necessary SV assessment information for SV prioritization and fixing.

To fill these gaps, we conduct a large-scale empirical study using 71,329 SV posts curated from SO and SSE. Specifically, we use Latent Dirichlet Allocation (LDA) [30] topic modeling and qualitative analysis to answer the following four Research Questions (RQs) that measure the support of Q&A sites for different SV discussion topics:
RQ1: What are SV discussion topics on Q&A sites?
RQ2: What are the popular and difficult SV topics?
RQ3: What is the level of expertise for supporting SV questions?
RQ4: What types of answers are given to SV questions?
Our findings to these RQs can help raise developers' awareness of common SVs and enable them to seek solutions to such SVs more effectively on Q&A sites. We also identify the areas to which experts can contribute to assist the secure software engineering community. Moreover, these common developers' SV concerns and their characteristics can be leveraged for making data-driven SV assessment models more practical (closer to developers' real-world needs) and enabling more effective understanding and fixing prioritization of commonly encountered SVs. Furthermore, we release one of the largest datasets of SV discussions that we have carefully curated from Q&A sites for replication and future work at https://github.com/lhmtriet/SV_Empirical_Study.
Chapter organization. Section 6.2 covers the related work. Section 6.3 describes the four research questions along with the methods and data used in this chapter. Sections 6.4 presents the results of each research question. Section 6.5 discusses the findings including how they can be used for SV assessment, and then mentions the threats to validity. Section 6.6 concludes and suggests several future directions.

[1] https://stackoverflow.com/
[2] https://security.stackexchange.com/

6.2 Related Work

6.2.1 Topic Modeling on Q&A Websites

Q&A websites such as SO and SSE contain a large number of discussion posts. LDA [30] has been frequently used to extract the taxonomy/topics of various software-related domains from such posts. In 2014, a seminal work of Barua et al. [373] discovered the topics of all SO posts. They also found that LDA could find more consistent topics than the tags on SO. Many subsequent studies have leveraged LDA to investigate discussions of specific domains, such as general security [370], concurrent computing [374], mobile computing [375], big data [376], machine learning [377] and deep learning [378]. Among the aforementioned studies, Yang et al. [370] is the closest to our work. However, our work is still fundamentally different from this previous study. Despite sharing a similar security context to Yang et al. [370], we focus specifically on the flaws of security implementation/features since exploitation of such flaws can disclose user's data and interrupt system operations. Moreover, we consider the content of both questions and answers of SV posts on two Q&A sites (SO and SSE) rather than just questions on SO as in [370]. This gives more in-depth insights into how different Q&A sites are supporting on-going SV discussions. Detailed discussion on these differences is given in section 6.5.1.

6.2.2 SV Assessment Using Open Sources

SV assessment has long been of interest to researchers. Shahzad et al. [379] conducted a large-scale study on the characteristics (e.g., risk metrics, exploitation, affected vendors and products) of reported SVs on NVD. Besides empirical study, there is another active research trend to build data-driven models to analyze SVs, as mentioned in Chapter 2. Bozorgi et al. [32] used Support Vector Machine to predict the probability and time-to-exploit of SVs. There have been many follow-up studies since then on developing learning-based models (e.g., [23, 21, 132]) to determine various properties of SVs using expert-based SV sources (e.g., CWE and NVD). A recent study [183] leveraged security mentions on social media (i.e., Twitter and Reddit) to forecast the SV-related activities on GitHub. Unlike the above studies, we focus on SV analytics on developer Q&A sites. Several studies (e.g., [380, 381]) analyzed SVs of different programming languages using code snippets on SO. Contrary to these studies, we do not limit our investigation to any specific programming language, and we consider every type of SV-related posts, not just the ones with code snippets.

6.3 Research Method

6.3.1 Research Questions

We investigated four RQs to study the support of Q&A websites for SV-related discussions. To answer these RQs, we retrieved 71,329 SV posts from a general Q&A website (SO) and a security-centric one (SSE) using both the tags and content of posts (see section 6.3.2).

RQ1: What are SV discussion topics on Q&A sites?

Motivation: To provide fine-grained information about the support of SO and SSE for different types of SV discussions, we first needed to identify the taxonomy of commonly discussed SV topics in RQ1. Our taxonomy does not aim to replace the existing ones provided by experts (e.g., CWE or OWASP), but rather helps to highlight the important aspects of SVs from developers' perspectives. Such taxonomy would also provide developers' real-world needs to help make SV assessment effort more practical.

Method: Following the standard practice in [373, 370, 375, 374, 376, 377, 378], RQ1 used the Latent Dirichlet Allocation (LDA) [30] topic modeling technique (see section 6.3.3) to

select SV discussion topics based on the titles, questions and answers of SV posts on both SO and SSE. LDA is commonly used since it can produce topic distribution (assigning multiple topics with varying relevance) for a post, providing more flexibility/scalability than manual coding. We also used the topic share metric [373] in Eq. (6.1) to compute the proportion (share_i) of each SV topic and their trends over time.

$$\text{share}_i = \frac{1}{N} \sum_{p \in D} \text{LDA}(p,\ \text{T}_i) \tag{6.1}$$

where p, D and N are a single SV post, the list of all SV posts and the number of such posts, respectively; T_i is the i^{th} topic and LDA is the trained LDA model.

RQ2: What are the popular and difficult SV topics?

Motivation: After the SV topics were identified, RQ2 identified the popular and difficult topics on Q&A websites. The results of RQ2 can aid the selection of a suitable (i.e., more popular and less difficult) Q&A site for respective SV topics.

Method: To quantify the topic popularity, we used four metrics from [375, 370, 376, 374], namely the average values of (*i*) views, (*ii*) scores (upvotes minus downvotes), (*iii*) favorites and (*iv*) comments. Intuitively, a more popular topic would attract more attention (views), interest (scores/favorites) and activities (comments) per post from users. We also obtained the geometric mean of the popularity metrics to produce a more consistent result across different topics. Geometric mean was used instead of arithmetic mean here since the metrics could have different units/scales. To measure the topic difficulty, we used the three metrics from [375, 370, 376, 374]: (*i*) percentage of getting accepted answers, (*ii*) median time (hours) to receive an accepted answer since posted, and (*iii*) average ratio of answers to views. A more difficult topic would, on average, have a lower number of accepted answers and ratio of answers to views, but a higher amount of time to obtain accepted answers. To achieve this, we took reciprocals of the difficulty metrics (*i*) and (*iii*) so that a more difficult topic had a higher geometric mean of the metrics.

RQ3: What is the level of expertise to answer SV questions?

Motivation: RQ3 checked the expertise level available on Q&A websites to answer SV questions, especially the ones of difficult topics. The findings of RQ3 can shed light on the amount of support each topic receives from experienced users/experts on Q&A sites and which topic may require more attention from experts. Note that experts here are users who frequently contribute helpful (accepted) answers/knowledge.

Method: We measured both users' general and specific expertise for SV topics on Q&A sites. For the general expertise, we leveraged the commonly used metric, the reputation points [382, 380, 381], of users who got accepted answers since reputation is gained through one's active participation and appreciation from the Q&A community in different topics. A higher reputation received for a topic usually implies that the questions of that topic are of more interest to experts. Similar to [380], we did not normalize the reputation by user's participation time since reputation may not increase linearly, e.g., due to users leaving the sites. However, reputation is not specific to any topic; thus, it does not reflect whether a user is experienced with a topic. Hence, we represented developers' specific expertise with the SV content in their answers on Q&A sites. This was inspired by Dey et al.'s findings that developers' expertise/knowledge could be expressed through their generated content [383]. We determined a user's expertise in SV topics using the topic distribution generated by LDA applied to the concatenation of all answers to SV questions given by that user. The specific expertise of an SV topic (see Eq. (6.2)) was then the total correlation between LDA outputs of the current topic in SV questions and the specific expertise of users who got the respective accepted answers. The correlation of LDA values could reveal the knowledge (SV topics) commonly used to answer questions of a certain (SV) topic [373].

$$Specific\_Expertise_i = \sum_{p \in D} \text{LDA}(Q(p),\ \text{T}_i) \odot \text{LDA}(K(U_{\text{Accept.}}))$$
$$K(U_{\text{Accept.}}) = A^1_{U_{\text{Accept.}}} + A^2_{U_{\text{Accept.}}} + ... + A^k_{U_{\text{Accept.}}} (k = \left|A_{U_{\text{Accept.}}}\right|) \tag{6.2}$$

where D is the list SV posts and T_i is the i^{th} topic, while $Q(p)$ and $K(U_{\text{Accept.}})$ are the question content and SV knowledge of the user $U_{\text{Accept.}}$ who gave the accepted answer of the post p, respectively. $\odot$ is the topic-wise multiplication. $\left|A_{U_{\text{Accept.}}}\right|$ is all SV-related answers given by user $U_{\text{Accept.}}$. Note that we only considered posts with accepted answers to make it consistent with the general expertise.
Specifically, for each question, we first extracted the user that gave the accepted answer ($U_{\text{Accept.}}$). We then gathered all answers, not necessarily accepted, of that user in SV posts ($\left|A_{U_{\text{Accept.}}}\right|$). Such answer list was the SV knowledge of $U_{\text{Accept.}}$ ($K(U_{\text{Accept.}})$). Finally, we computed the LDA topic-wise correlation between the topic T_i in the current SV question ($\text{LDA}(Q(p),\ \text{T}_i)$) and the user knowledge ($\text{LDA}(K(U_{\text{Accept.}}))$) to determine the specific expertise for post p.

RQ4: What types of answers are given to SV questions?

Motivation: RQ4 extended RQ2 in terms of the solution types given if an SV question is satisfactorily answered. We do not aim to provide solutions for every single SV. Rather, we analyze and compare the types of support for different SV topics on SO and SSE, which can guide developers to a suitable site depending on their needs (e.g., looking for certain artefacts). To the best of our knowledge, we are the first to study answer types of SVs on Q&A sites.

Method: We employed an open coding procedure [384] to inductively identify answer types. LDA is *not suitable* for this purpose since it relies on word co-occurrences to determine categories. In contrast, the same type of solutions may not share any similar words. In RQ4, we only considered the posts with accepted answer to ensure the high quality and relevance of the answers. We then used stratified sampling to randomly select 385 posts (95% confidence level with 5% margin error [291]) each from SO and SSE to categorize the answer types. Stratification ensured the proportion of each topic was maintained. Following [385], the author of this thesis and a PhD student with three years of experience in Software Engineering and Cybersecurity first conducted a pilot study to assign initial codes to 30% of the selected posts and grouped similar codes into answer types. For example, the accepted answers of SO posts 32603582 (PostgreSQL code), 20763476 (MySQL code) and 12437165 (Android/Java code) were grouped into the *Code Sample* category. Similarly to [386], we also allowed one post to have more than one answer type. The two same people then independently assigned the identified categories to the remaining 70% of the posts. The Kappa inter-rater score (κ) [345] was 0.801 (strong agreement), showing the reliability of our coding. Another PhD student with two-year experience in Software Engineering and Cybersecurity was involved to discuss and resolve the disagreements. We also correlated the answer types with the question types on Q&A sites [386].

6.3.2 Software Vulnerability Post Collection

To study the support of Q&A sites for SV discussions, we proposed a workflow (see Fig. 6.1) to obtain, to the best of our knowledge, the largest and most contemporary set of SV posts on both SO and SSE. To identify SV-related posts, we started from security posts on Q&A sites as every SV is a security issue by definition. This decision helped to increase the relevance of our retrieved SV posts. Specifically, all the posts on SSE were presumably relevant to security as SSE is a security-centric site; whereas, on SO, we used security posts automatically collected using our novel tool, PUMiner [304]. PUMiner alleviates the need

for the non-security (negative) class to predict security posts on Q&A sites based on a two-stage PU learning framework [387]. Retrieving non-security posts in practice is challenging since these posts should not contain any security context, which requires significant human effort to define and verify. It is also worth noting that manual selection of security/non-security posts also does not scale to millions of posts on Q&A sites. PUMiner has been demonstrated to be more effective in retrieving security posts on SO than many learning-based baselines such as one-class SVM [388, 389] and positive-similarity filtering [390] on unseen posts. PUMiner can also successfully predict the cases where keyword matching totally missed with an MCC of 0.745. Notably, with only 1% labelled positive posts, PUMiner is still 160% better than fully-supervised learning. More details of PUMiner can be found in Appendix 6.7. Using the curated security posts on SO and SSE, we then employed *tag-based* and *content-based filtering* to retrieve SV posts based on their tags and content of other parts (i.e., title, body and answers), respectively. We considered a post to be related to SV when it mainly discussed a security flaw and/or exploitation/testing/fixing of such flaw to compromise a software system (e.g., SO post 29098142[3]). A post was not SV-related if it just asked how to implement/use a security feature (e.g., SO post 685855) without any explicit mention of a flaw. All the tags, keywords and posts collected were released at https://github.com/lhmtriet/SV_Empirical_Study.

Tag-based filtering. We had a *vulnerability* tag on SSE but not on SO to obtain SV-related posts, and the *security* tag on SO used by [370] was too coarse-grained for the SV domain. Many posts with the *security* tag did not explicitly mention SV (e.g., SO post 65983245 about privacy or SO post 66066267 about how to obtain security-relevant commits). Therefore, we used Common Weakness Enumeration (CWE), which contains various SV-related terms, to define relevant SV tags. However, the full CWE titles were usually long and uncommonly used in Q&A discussions. For example, the fully-qualified CWE name of SQL-injection (CWE-89) is "*Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')*", which appeared only nine times on SO and SSE. Therefore, we needed to extract shorter and more common terms from the full CWE titles. We adopted Part-of-Speech (POS) tagging for this purpose, in which we only considered consecutive (n-grams of) verbs, nouns and adjectives since most of them conveyed the main meaning of a title. For instance, we obtained the following 2-grams for CWE-89: *improper neutralization*, *special elements*, *elements used*, *sql command*, *sql injection*. We obtained 2,591 n-gram ($1 \leq n \leq 3$) terms that appeared at least once on either SO or SSE. To ensure the relevance of these terms, we manually removed the irrelevant terms without any specific SV context (e.g., *special elements*, *elements used* and *sql command* in the above example). We found 60 and 63 SV-related tags on SO and SSE that matched the above n-grams, respectively. We then obtained the initial set_{tag} of SV posts that had at least one of these selected tags.

Content-based filtering. As recommended by some recent studies [391, 304], tag-based filtering was not sufficient for selecting posts due to wrong tags (e.g., non SV-post 38539393 on SO with *stack-overflow* tag) or general tags (e.g., SV post 15029849 on SO with only *php* tag). Therefore, as depicted in Fig. 6.1, we customized content-based filtering, which was based on keyword matching, to refine the set_{tag} obtained from the tag-based filtering step and select missing SV posts that were not associated with SV tags. First, we presented the up-to-date list of 643 SV keywords for matching at https://github.com/lhmtriet/SV_Empirical_Study. These keywords were preprocessed with stemming and augmented with American/British spellings, space/hyphen to better handle various types of (mis-)spellings/plurality.

[3] stackoverflow.com/questions/29098142 (postid: 29098142). SSE format is security.stackexchange.com/questions/postid. Posts in our paper follow these formats.

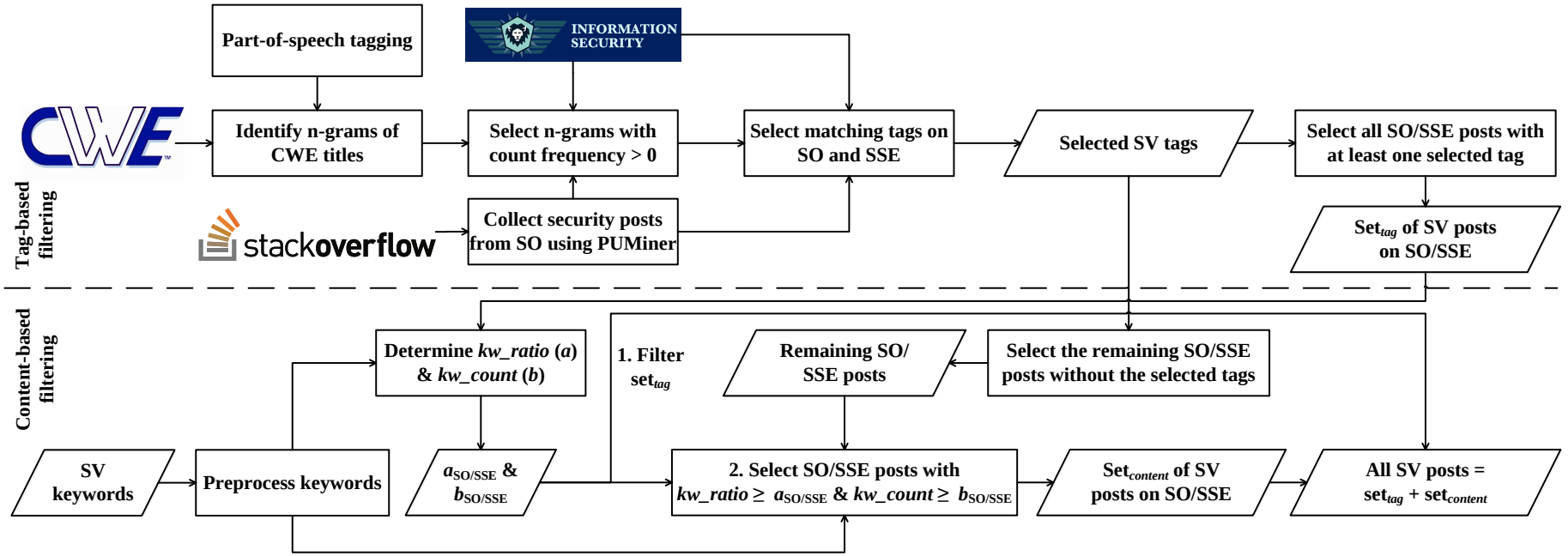


FIGURE 6.1: Workflow of retrieving posts related to SV on Q&A websites using tag-based and content-based filtering heuristics.

TABLE 6.1: Content-based thresholds ($a_{\text{SO/SSE}}$ & $b_{\text{SO/SSE}}$) for the two steps of the content-based filtering as shown in Fig. 6.1.

Threshold	Stack Overflow (SO)		Security StackExchange (SSE)	
	Step 1	**Step 2**	**Step 1**	**Step 2**
a	1	3	2	3
b	0.011	0.017	0.017	0.025

TABLE 6.2: The obtained SV posts using our tag-based and content-based filtering heuristics.

	Stack Overflow (SO)	**Security StackExchange (SSE)**	**SO + SSE**
Set$_{tag}$	46,212	9,677	55,889
Set$_{content}$	12,660	2,780	15,440
Total	58,872	12,457	71,329

For instance, we considered the following variants: *input(-)sanitization/sanit/sanitisation/sanitis* for "*input sanitization*". Similar to [391, 304], we also performed exact matching for three-character keywords and subword matching for longer ones to reduce false positives. Subsequently, for each set$_{tag}$ (SO and SSE) obtained in the tag-based filtering step, we computed two content-based metrics (see Eq. (6.3)) [391, 304]: *kw_count* and *kw_ratio*, denoting the count and appearance proportion of SV keywords in a post, respectively. *Kw_count* ensured diverse SV-related content in a post, while *kw_ratio* increased the confidence that these relevant words did not appear by chance.

$$kw\_count_p = |SV\_KWs_p| \ , \ \ kw\_ratio_p = \frac{|SV\_KWs_p|}{|Words_p|} \tag{6.3}$$

where $|SV\_KWs_p|$ and $|Words_p|$ are the numbers of SV keywords and total number of words in post p, respectively.

Based on the post content and human inspection, the thresholds $a_{\text{SO/SSE}}$ and $b_{\text{SO/SSE}}$ for filtering set$_{tag}$ (step 1) as well as selecting extra posts based on their content (step 2) were found, as given in Table 6.1. Using these thresholds, we obtained set$_{tag}$ and set$_{content}$ of SV posts on SO and SSE, respectively, as shown in Fig. 6.1.

SV datasets and validation. As of June 2020, we retrieved 285,720 and 58,912 security posts from SO using PUMiner [304] and SSE using Stack Exchange Data Explorer, respectively. We then applied the tag-based and content-based filtering steps in Fig. 6.1 and obtained **71,329 SV posts** (see Table 6.2) in total including 55,883 and 15,436 ones for set$_{tag}$ and set$_{content}$, respectively. We manually validated four different sets of SV posts, i.e., set$_{tag}$ and set$_{content}$ for SO and SSE, respectively. Specifically, we randomly sampled 385 posts (significant size [291]) in each set for two researchers (i.e., the author of this thesis and a PhD student with three years of experience in Software Engineering and Cybersecurity) to examine independently.

For set$_{tag}$, we disagreed on 7/770 cases and only two posts were not related to SV. The main issue was still the incorrect tag assignment (e.g., SSE post 175264 was about dll injection but tagged with *malware*[4]), though this issue had been significantly reduced by the content-based filtering. For set$_{content}$, the relevance of the posts was very high as there was no discrepant case.

[4]This post was short yet contained many SV keywords (e.g., "*injection*" and "*hijack*"), resulting in high *kw_count* and *kw_ratio* of the content-based filtering.

Table 6.3: Top-5 tags of SV, security and general posts on SO and SSE (in parentheses).

No.	SV posts	Security posts	General posts
1	memory-leaks (malware)	security (encryption)	javascript (encryption)
2	segmentation-fault (web-application)	encryption (tls)	java (tls)
3	php (xss)	php (authentication)	python (authentication)
4	c (exploit)	java (passwords)	c# (passwords)
5	security (penetration-test)	cryptography (web-application)	php (certificates)

Our SV dataset was only 20% overlapping with the existing security dataset [370], implying that there were significant differences in the nature of the two studies. Note that we followed the settings in [370] to retrieve the updated security posts from the same SO data we used in our study. We also reported the top tags of SV posts (see Table 6.3) and compared them with the ones of security posts [370] and a subset of all the posts containing an equal number of posts to the SV posts on SO and SSE. SV posts were associated with many SV-related tags (e.g., *memory-leaks*, *malware*, *segmentation-fault*, *xss*, *exploit* and *penetration-test*). Conversely, security posts were tagged with general terms that may not explicitly discuss security flaws such as *encryption*, *authentication* and *passwords*. The tags of general posts were mostly programming languages on SO and general security terms on SSE. These findings highlight the importance of obtaining SV-specific posts instead of reusing the security posts to study the support of Q&A sites for SV-related discussions.

6.3.3 Topic Modeling with LDA

Following the common practice of the existing work (e.g., [370, 373, 376]), we extracted the topics of the identified SV-related posts on both SO and SSE using Latent Dirichlet Allocation (LDA) [30].

Preprocessing of SV posts. Following the previous practices of [370, 374], we first removed the HTML tags and code snippets in each post as these elements were not informative for topic modeling. We also converted the text to lowercase, removed punctuations, and then eliminated stop words and performed stemming (reducing a word to its root form) to avoid irrelevant and multi-form words.

Topic modeling with LDA. We applied LDA to the title, question body and all answers of each Q&A post. Regarding the number of topics (k) of LDA, we examined an inclusive range from 2 to 80, with an increment of one topic at a time. As suggested in [373, 375, 374, 376], alongside k, we also tried different values of α ($1/k$ or $50/k$) and β (0.01 or same as α) hyperparameters to optimize the performance of LDA. α controls the sparsity of the topic-distribution per post and β determines the sparsity of the word-distribution per topic. For each tuple of (k, α and β), we ran LDA with 1,000 iterations, then evaluated the coherence metric [392] of the identified topics. Coherence metric has been recommended by many previous studies (e.g., [393, 394]) to select the optimal number of LDA topics since it usually highly correlates with human understandability. Topic coherence is the average correlation between pairs of words that appear in the same topic. The higher value of the coherence metric means the more coherent content of the posts

TABLE 6.4: SV topics on SO and SSE identified by LDA along with their proportions and trends over time. **Notes**: The topic proportions on SSE are in parenthesis. The trends of SO are the top solid sparklines, while the trends of SSE are the bottom dashed sparklines. Unit of proportion: %.

Topic Name	Proportion	Trend
Malwares (T1)	1.39 (8.18)	
SQL Injection (T2)	11.0 (4.17)	
Vulnerability Scanning Tools (T3)	5.42 (3.15)	
Cross-site Request Forgery (CSRF) (T4)	9.49 (5.09)	
File-related Vulnerabilities (T5)	2.88 (3.24)	
Synchronization Errors (T6)	3.79 (0.47)	
Encryption Errors (T7)	1.82 (7.81)	
Resource Leaks (T8)	10.6 (0.42)	
Network Attacks (T9)	1.37 (8.79)	
Memory Allocation Errors (T10)	**21.6** (2.82)	
Cross-site Scripting (XSS) (T11)	7.73 (8.09)	
Vulnerability Theory (T12)	10.7 (**33.7**)	
Brute-force/Timing Attacks (T13)	1.08 (1.28)	

within the same topic. To avoid insignificant topics like [373], we only considered topics with a probability of at least 0.1 in a post. We manually read the top-20 most frequent words and 15 random posts of each topic per site (SO/SSE) obtained by the trained LDA models to label the name of that topic as done in [376, 374]. The LDA model with the most relevant set of topics would be used for answering the four RQs.

6.4 Results

6.4.1 RQ1: What are SV Discussion Topics on Q&A Sites?

Following the procedure in section 6.3.3, we identified *13* SV topics (see Table 6.4) on SO and SSE using the optimal LDA model with $\alpha = \beta = 0.08$. We found LDA models having from 11 to 17 topics produced similar coherence metrics. Three of the authors manually examined these cases, as in [393]. Duplicate and/or platform-specific topics (e.g., web and mobile) appeared from 14 topics, making the taxonomy less generalizable. 11 and 12 topics also had high-level topics (e.g., combining XSS and CSRF). Thus, 13 was chosen as the optimal number of SV topics. All the terms/posts of each SV topic can be found at `https://github.com/lhmtriet/SV_Empirical_Study`. We describe each topic hereafter with example SO/SSE posts. We examined 15 random posts per topic per site. If we identified some common patterns of discussions (e.g., attack vectors or assets) on a site, we would extract another 15 random posts of the respective site to confirm our observations. If a pattern was no longer evident in the latter 15 posts, we would not report it.

Malwares (T1). This topic referred to the detection and removal of malicious software. T1 posts on SO were usually about malwares in content management systems such as Wordpress or Joomla (e.g., post 16397854: "*How to remove wp-stats malware in wordpress*" or post 11464297: "*How to remove .htaccess virus*'). In contrast, SSE often discussed malwares/viruses coming from storage devices such as SSD (e.g., post 227115: "*Can viruses of*

one ssd transfer to another ssd?") or USB (e.g., post 173804: "*Can Windows 10 bootable USB drive get infected while trying to reinstall Windows?*").

SQL Injection (T2). This topic concerned tactics to properly sanitize malicious inputs that could modify SQL commands and pose threats (e.g., stealing or changing data) to databases in various programming languages (e.g., PHP, Java, C#). A commonly discussed tactic was to use prepared statements, which also helped increase the efficiency of query processing. For example, developers asked questions like "*How to parameterize complex oledb queries?*" (SO post 9650292) or "*How to make this code safe from SQL injection and use bind parameters*" (SSE post 138385).

Vulnerability Scanning Tools (T3). This topic was about issues related to tools for automated detection/assessment of potential SVs in an application. Discussions of T3 mentioned different tools, and OWASP ZAP was a commonly discussed one. For example, post 62570277 on SO discussed "*Jenkins-zap installation failed*", while post 126851 on SSE asked "*How do I turn off automated testing in OWASP ZAP?*" One possible explanation is that OWASP ZAP is a free and easy-to-use tool for detecting and assessing SVs that appear in the well-known top-10 OWASP list for web applications.

Cross-site Request Forgery (CSRF) (T4). This topic contained discussions on proper setup and configuration of web application frameworks to prevent CSRF SVs. These SVs could be exploited to send requests to perform unauthorized actions from an end-user that a web application trusts. Discussions covered various issues in implementing different CSRF prevention techniques recommended by OWASP.[5] Some commonly discussed techniques were anti-CSRF token (e.g., SO post 59664094: "*Why Laravel 4 CSRF token is not working?*"), double submit cookie (e.g., SSE post 203996: "*What is double submit cookie? And how it is used in the prevention of CSRF attack?*"), and SameSite cookie attribute (e.g., SO post 41841880: "*What is the benefit of blocking cookie for clicked link? (SameSite=strict)*").

File-related Vulnerabilities (T5). Discussions of this topic were about SVs in files that could be exploited to gain unauthorized access. The common SV types were Path/Directory Traversal via Symlink (e.g., SSE post 165860: "*Symlink file name - possible exploit?*"), XML External Entity (XXE) Injection (e.g., SO post 51860873: "*Is SAXParserFactory susceptible to XXE attacks?*"), and Unrestricted File Upload (e.g., SSE post 111935: "*Exploiting a PHP server with a .jpg file upload*"). These SVs usually occurred for Linux-based systems, suggesting that Linux is more popular for servers.

Synchronization Errors (T6). This topic involved SVs produced through errors in synchronization logic (usually related to threads), which could slow down system performance. Some common SV types being discussed were deadlocks (e.g., SO post 38960765: "*How to avoid dead lock due to multiple oledb command for same table in ssis*") and race conditions (e.g., SSE post 163209: "*What's the meaning of 'the some sort of race condition' here?*").

Encryption Errors (T7). This topic included cryptographic issues leading to falsified authentication or retrieval of sensitive data, e.g., Man-in-the-middle (MITM) attack. Many posts discussed public/private keys for encryption/decryption, especially using SSL/TLS certificates to defend against MITM attacks (attempts to steal information sent between browsers and servers). Some example discussions are post 23406005 on SO ("*Man In Middle Attack for HTTPS*") or post 105773 on SSE ("*How is it that SSL/TLS is so secure against password stealing?*"). This may imply that many developers are still not familiar with these certificates in practice.

[5] https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

Resource Leaks (T8). This topic considered SVs arising from improper releases of unused memory which could deplete resources and decrease system performance. Many discussions of T8 were about memory leaks in mobile app development. Issues were usually related to Android (e.g., SO post 58180755: "*Deal with Activity Destroying and Memory leaks in Android*") or IOS (e.g., SO post 47564784: "*iOS dismissing a view controller doesn't release memory*").

Network Attacks (T9). This topic discussed attacks carried out over an online computer network, e.g., Denial of Service (DoS) and IP/ARP Spoofing, and potential mitigations. These network attacks directly affected the availability of a system. For instance, SSE post 86440 discussed "*VPN protection against DDoS*" or SO post 31659468 asked "*How to prevent ARP spoofing attack in college?*".

Memory Allocation Errors (T10). T10 and T8 were both related to memory issues, but T10 did not consider memory release. Rather, this topic more focused on SVs caused by accessing or using memory outside of what allocated that could be exploited to access restricted memory location or crash an application. In this topic, segmentation faults (e.g., SO post 31260018: "*Segmentation fault removal duplicate elements in unsorted linked list*") and buffer overflows (e.g., SSE post 190714: "*buffer overflow 64 bit issue*") were commonly discussed by developers.

Cross-site Scripting (XSS) (T11). This topic mentioned tactics to properly neutralize user inputs to a web page to prevent XSS attacks. These attacks could exploit users' trust in web servers/pages to trick them to execute malicious scripts and perform unwanted actions. XSS (T11) and CSRF (T4) are both client-side SVs, but XSS is more dangerous since it can bypass all countermeasures of T4.[5] On SO and SSE, discussions covered all three types of XSS: (*i*) reflected XSS (e.g., SSE post 57268: "*How does the anchor tag (<a>) let you do an Reflected XSS?*"), (*ii*) stored/persistent XSS (e.g., SO post 54771897: "*How to defend against stored XSS inside a JSP attribute value in a form*"), and (*iii*) DOM-based XSS (e.g., SO post 44673283: "*DOM XSS detection using javascript(source and sink detection)*").

Vulnerability Theory (T12). This topic focused on theoretical/social aspects and best practices in the SV life cycle. Many posts compared different SV-related terminologies, e.g., SSE post 103018 asked about "*In CIA triad of information security, what's the difference between confidentiality and availability?*" or SO post 402936 discussed "*Bugs versus vulnerabilities?*". Several other posts asked about internal SV reporting process (e.g., SO post 3018198: "*How best to present a security vulnerability to a web development team in your own company?*") or public SV disclosure policy (e.g., SSE post: "*How to properly disclose a security vulnerability anonymously?*").

Brute-force/Timing Attacks (T13). T13 and T7 both exploited cryptographic flaws, but these two topics used different attack vectors/methods. T7 focused on MITM attacks, while T13 was about attacks making excessive attempts or capturing the timing of a process to gain unauthorized access. Some example posts of T13 are SO post 3009988 ("*What's the big deal with brute force on hashes like MD5*") or SSE post 9192 ("*Timing attacks on password hashes*").

Proportion and Evolution of SV Topics. We analyzed the proportion (share metric in Eq. (6.1)) and the evolution trend of SV topics from their inception on SO (2008) and SSE (2010) to 2020 (see Table 6.4). The topic patterns and dynamics of SO were different from those of SSE. Specifically, Memory Allocation Errors (T10) had the greatest number of posts on SO, while Vulnerability Theory (T12) had the largest proportion on SSE. Apart from XSS (T11) and Brute-force/Timing Attacks (T13), topics with many posts in one source were not common in the other source. Moreover, we discovered three consistent topic trends on both SO and SSE: Malwares (T1) (↗), CSRF (T4) (↗), File-related SVs (T5) (↗) and Vulnerability Theory (T12) (↘). Among them, CSRF had the fastest changing

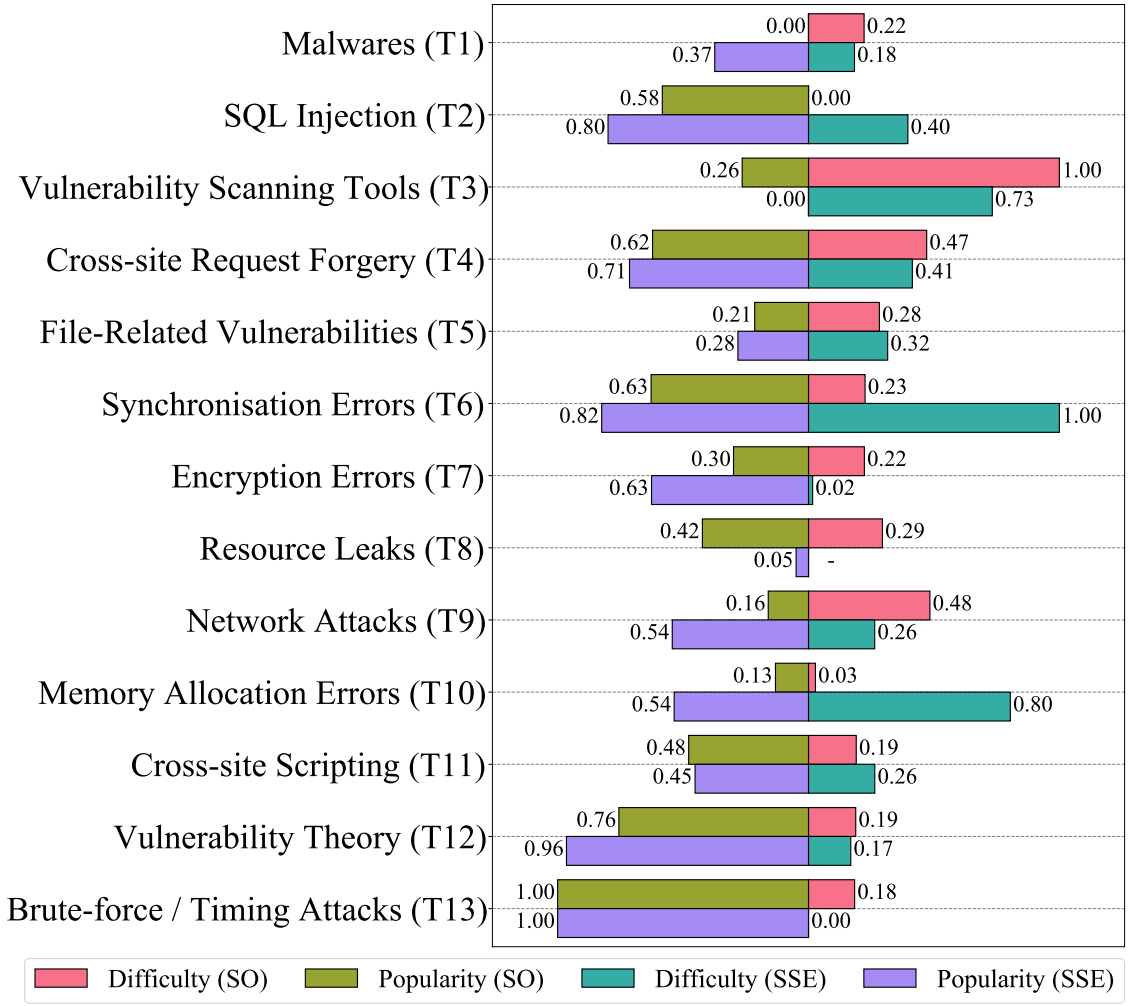


FIGURE 6.2: Popularity and difficulty of 13 SV topics on SO and SSE. **Notes**: The values were normalized by the max and min values of each category. Difficulty of T8 on SSE was excluded since it did not have any accepted answer.

pace. These trends were confirmed significant with p-values < 0.01 using Mann-Kendall non-parametric trend test [395].

6.4.2 RQ2: What are the Popular and Difficult SV Topics on Q&A Sites?

As shown in Fig. 6.2, the popularity and difficulty of 13 identified SV topics were different between SO and SSE. For conciseness, we only report the geometric means of the popularity and difficulty metrics in this section. The complete values of individual metrics (see section 6.3.1) can be found at `https://github.com/lhmtriet/SV_Empirical_Study`.

Topic Popularity. Brute-force/Timing attacks (T13) and Vulnerability Theory (T12) were the top-2 most popular topics. Despite being the most popular topic, T13 only had 1.1% and 1.3% posts on SO and SSE, respectively. Conversely, Memory Allocation Errors (T10) had the most posts on SO (RQ2), but T10 was only the second least popular topic. We found no significant correlation between the topic popularity and share metric with Kendall's Tau correlation test [396] at 95% confidence level. These findings suggest that

the share metric does not necessarily reflect the topic popularity since it does not consider users' activities on Q&A sites.

Topic Difficulty. The most difficult topics were not popular or associated with many posts, i.e., Vulnerability Scanning Tools (T3) and Network Attacks (T9) on SO as well as T3, Synchronization Errors (T6) and Memory Allocation Errors (T10) on SSE. The high difficulty of T3 on both sites was potentially caused by the low familiarity with a wide array of vendors and tools available for SV detection and assessment [13]. Some topics with many posts (high share metric) like Memory Allocation Errors (T10) and SQL Injection (T2) were the two easiest ones on SO despite being significantly more difficult on SSE. On the contrary, Malwares (T1) and Network Attacks (T9) were more popular yet easier on SSE. These numbers suggest that it may be better to ask the topics T2, T8 (only a few posts on SSE) and T10 on SO to obtain answers faster, while asking T1 and T9 on SSE would be more optimal. However, the topic difficulty did not correlate with either the topic popularity or share metric on both SO and SSE, confirmed using Kendall's Tau test [396] with a confidence level of 95%. With the same confidence level, no significant differences in terms of average topic-wise popularity and difficulty between SO and SSE were recorded using non-parametric Mann-Whitney U-test [397].

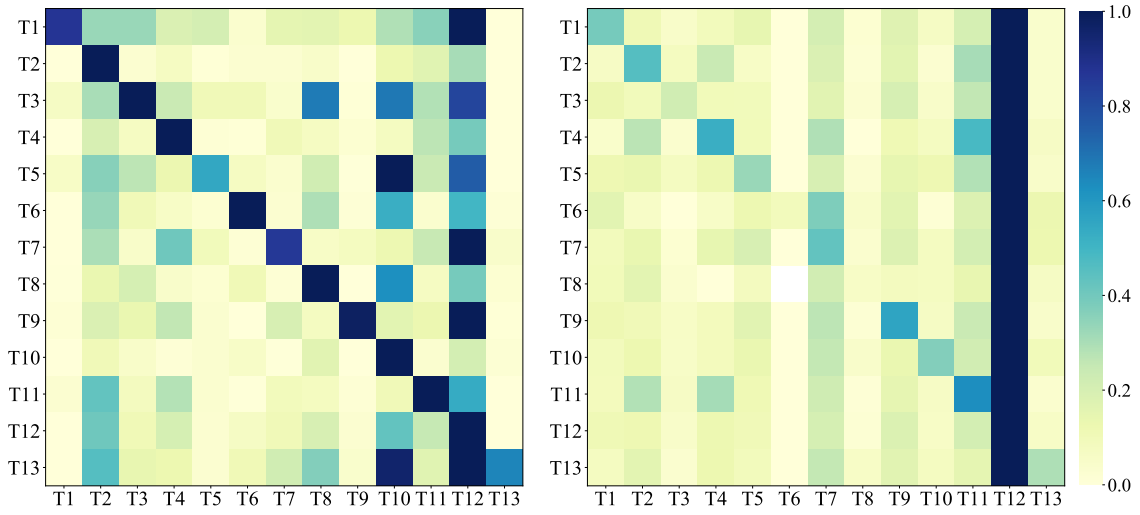
6.4.3 RQ3: What is the Level of Expertise to Answer SV Questions on Q&A Sites?

General Expertise. The average general expertise (reputation) of the accepted answerers in SV posts was 1.3 to 5.8 times higher than those of generic posts [373], general security [370], mobile development [375], concurrency [374], machine learning [377] and deep learning [378]. The higher reputation values were confirmed with p-values < 0.01 (significance level) using non-parametric Mann-Whitney U-test [397]. However, the average percentage of the same users who got accepted answers on both SO and SSE was quite small across topics, i.e., 1% to 18%, implying not much SV knowledge sharing between the two sites. The average topic-wise reputation on SO was higher than that of SSE with a p-value of 0.001 (Mann-Whitney U-test). This might be because SO users could engage in many more posts of different topics (not only security). Table 6.5 reports the general expertise of 13 SV topics. On SO, Brute-force/Timing Attacks (T13), SQL Injection (T2), Synchronization Errors (T6) and XSS (T11) were the topics that experts focused on the most. On SSE, T13 again and Encryption Errors (T7) were the topics of interest for experts. In contrast, Malwares (T1), Vulnerability Scanning Tools (T3) and Network Attacks (T9) on SO did not attract as much attention from experts. On SSE, T3 was also of the least interest to experts. Overall, experts on Q&A sites tended to favor the SV topics with high popularity and low difficulty, confirmed with p-values < 0.01 using Kendall's Tau correlation test [396].

Specific Expertise. Fig. 6.3 shows the correlation between the pairs of question SV topics and answerers' SV topics (see Eq. (6.2)). The most frequent answerers' SV topic was Vulnerability Theory (T12). On SSE, frequent answerers for T12 could answer every question topic. On SO, besides T12, users specialized in Memory-related Errors (T8 and T10) also answered the questions of other SV topics. These patterns might be because of the prevalence (RQ2) of topics T8 and T10 on SO as well as T12 on SSE. Conversely, Malwares (T1), Network Attacks (T9) and Brute-force/Timing Attacks (T13) on SO as well as Synchronization Errors (T6), Resource Leaks (T8) and T13 on SSE had unique answerers (i.e., users who usually answered questions of only one topic in the SV domain). Furthermore, on SO, most answerers were relevant for each SV topic (dark color on the diagonal in Fig. 6.3a), but it was not always the case on SSE (see Fig. 6.3b). Such results suggest that it may be easier to find relevant answerers for different SV topics on SO.

TABLE 6.5: General expertise in terms of average reputation of each topic on SO and SSE (in parentheses). **Notes**: The values were normalized by the max and min values of each category. T8 on SSE was excluded since it did not have any accepted answer.

General expertise	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**	**T11**	**T12**	**T13**
Reputation	**0.00** (0.16)	0.93 (0.05)	**0.01** (**0.00**)	0.28 (0.18)	0.43 (0.14)	0.88 (0.04)	0.51 (0.72)	0.49 (–)	0.07 (0.24)	0.62 (0.34)	0.84 (0.22)	0.59 (0.29)	**1.00** (**1.00**)



(a) Specific Expertise: Question topics (y-axis) vs. Answerers' topics (x-axis) (SO)

(b) Specific Expertise: Question topics (y-axis) vs. Answerers' topics (x-axis) (SSE)

FIGURE 6.3: Topic correlations between SV questions & answerers' SV specific knowledge on SO (a) & SSE (b). **Notes**: Light to dark color shows weak to strong correlation. Each cell was normalized by the max and min values of each question topic.

6.4.4 RQ4: What Types of Answers are Given to SV Questions on Q&A Sites?

Our open coding process in RQ4 identified *seven* answer categories of SV discussions on Q&A sites, as shown in Table 6.6. Some answer types provided experience (DC/Co and Er) or language/platform-specific support (AT, ES and CS), which is hardly found on expert-based security sites (e.g., CWE or OWASP). We correlated such answer types with the question categories of Treude et al. [386]. We found reasonable matches between answer and question types, e.g., (dis)agreeing (DC/Co) with a decision (Decision Help), explaining (Ex) a concept (Conceptual), and providing different solutions to resolve unexpected situations (Discrepancy). Discrepancy and Error were also among the most frequent question types, supporting that our posts were about issues/errors in addressing SVs.

Site-wise answer types. According to Table 6.6, Action to Take (AT) and External Sources (ES) were the most common answer types on SO and SSE, respectively; whereas, Self-Answer (SA) was the least frequent one on both sites. We also noticed that both sites usually referred to external sources (ES). The most common sources included Wikipedia, other posts (e.g., related answers), GitHub issues/commits, product documentation (e.g., PHP, MySQL and Android) and SV sources (e.g., CVE (Common Vulnerabilities and Exposures) [66], NVD [16], CWE [28], OWASP [167], CVE Details [187] and Exploit-DB [398]). Note that some provided links were unavailable or no longer maintained (e.g., CVE Details). Overall, the answers to SV questions on SO frequently provided detailed instructions (AT) and/or code samples (CS), while SSE tended to share more experience (DC/Co and Ex) to help.

Topic-wise answer types. We extend the site-wise findings to individual topics to enable developers to select the respective site (SO vs. SSE) based on their preferable SV solution types, as shown in Table 6.7. Specifically, SO highlighted steps (AT) to fix Malwares (T1), Memory Allocation Errors (T10) and XSS (T11) as well as provided code snippets for SQL Injection (T2). On the other hand, SSE gave more relevant sources and explanations for such topics. One may argue that these different answer types were because

TABLE 6.6: Answer types of SV discussions identified on Q&A websites. **Note**: An answer can have more than one solution type. Proportions of SSE (the last column) are in parenthesis.

Answer type of SV discussions	Description & Example Posts	Top-3 related question types [386]	Proportion (%) on SO & SSE
(Dis-)Confirmation (DC/Co)	Confirm/agree or refute/disagree with a major point or concept made by the asker (e.g., SO post 16155188 or SSE post 31306)	Decision Help, How-to, Conceptual	11.5 (23.7)
Explanation (Ex)	Explain concepts, definitions and "why" to take certain actions (e.g., SO post 53446941 or SSE post 157240)	Decision Help, Conceptual, Discrepancy	14.6 (27.1)
Error (Er)	Point out an error in the source code or another attachment of the initial question (e.g., SO post 29750534 or SSE post 159907)	Discrepancy, Error, How-to	13.0 (2.3)
Action to Take (AT)	Describe step/action(s) ("how-to") to solve a problem (e.g., SO post 22860382 or SSE post 180053)	How-to, Discrepancy, Decision Help	**22.5** (15.4)
External Source (ES)	Provide reference/link to external source(s) (e.g., SO post 445177 or SSE post 107498)	Decision Help, How-to, Discrepancy	18.1 (**28.7**)
Code Sample (CS)	Provide an explicit example of code snippet (e.g., SO post 20763476 or SSE post 36804)	Discrepancy, How-to, Error	16.6 (2.0)
Self-Answer (SA)	Answer given by the same user who submitted the question (e.g., SO post 55784402 or SSE post 100761)	Discrepancy, Error, How-to	3.7 (1.0)

TABLE 6.7: Top-1 answer types of 13 SV topics on SO & SSE (in parenthesis). **Note**: T8 on SSE was excluded since it did not have any accepted answer.

Topic	Top-1 Answer Type	Topic	Top-1 Answer Type
T1	AT/ES (Ex)	T8	ES (–)
T2	CS (ES)	T9	DC/Co/ES/SA (Ex)
T3	ES (ES)	T10	AT (Ex)
T4	ES (DC/Co)	T11	AT (DC/Co)
T5	Ex (ES)	T12	Ex (ES)
T6	Ex/ES (DC/Co/ES)	T13	Ex/ES (Ex)
T7	Ex/ES (Ex)	–	–

of the different question types between SO and SSE, but we did not find any such significant differences for these topics. Instead, the fact that SO and SSE had quite different accepted answerers, as shown in RQ3, probably led to such different solution types. There were four topics, namely SV Scanning Tools (T3), Synchronization & Encryption Errors (T6 & T7), Brute-force/Timing Attacks (T13), sharing similar top solutions on both SO and SSE. The remaining topics (T4, T5, T8 and T9) were mostly answered with explanation (Ex) or external sources (ES) on both SO and SSE.

6.5 Discussion

6.5.1 SV Discussion Topics on Q&A Sites vs. Existing Security Taxonomies

SV-specific topics and their support on Q&A sites. Compared to Yang et al.'s taxonomy [370], we found related topics: T2, T3, T5, T10, T11 and T13, but we still had the following important differences. Firstly, our topics were emphasized more on security

flaws, e.g., issues with encryption/decryption algorithms (T7) than how to implement/use them as in [370]. Secondly, we identified SV-specific topics previously unreported in [370]: Malwares (T1), CSRF (T4), Synchronization Errors (T6), Resource Leaks (T8), Network Attacks (T9) and Vulnerability Theory (T12). These SV topics show the necessity of focusing on SV-specific posts instead of general security ones. Thirdly, unlike [370], we did not consider language-dependent topics (i.e., PHP, Flash, Javascript, Java and ASP.NET), helping our topics be more generalizable (e.g., XSS can occur in both PHP and ASP.NET). Zahedi et al. [399] also devised a security taxonomy for GitHub issues; however, they focused on security features and implementation instead of any specific SV types. Note that we studied SV posts on both SO and SSE, while the existing studies only used one source of data (SO), enhancing the generalizability of our study. Specifically, we shed light on the differences between SV discussion topics on SO and those on SSE in terms of their proportions (RQ1), popularity/difficulty (RQ2), level of expertise (RQ3) and types of answers (RQ4). Our findings can be leveraged to select a suitable site (i.e., more popular/experts, less difficult or having certain answer types) for asking different SV questions.

Disconnection between SV discussions and expert-based SV sources. Two researchers (i.e., the author of this thesis and a PhD student with three years of experience in Software Engineering and Cybersecurity) manually mapped 13 SV topics with CWEs, as shown in Table 6.8. The agreement between the two people was strong (Kappa score [345] was 0.892), and disagreements were resolved during a discussion section with another researcher (i.e., a PhD student with two-year experience in Software Engineering and Cybersecurity). We found that only seven of them were overlapping with the two well-known expert-based SV taxonomies: top-25 CWE[6] and top-10 OWASP[7]. The overlapping topics were T2 (SQL Injection), T4 (CSRF), T5 (i.e., Path-traversal and Unrestricted File Upload), T7 (i.e., Improper Certificate Validation), T8 (Resource Leaks), T10 (Memory Allocation Errors) and T11 (XSS). There was no CWE for T3 and T12 since they mainly discussed SV scanning tools and/or socio-technical issues, respectively. In fact, using keyword matching, we found that only 159 and 71 out of a total of 839 CWEs were mentioned on SO and SSE, respectively; and only 20 and two CWEs appeared more than 100 times on SO and SSE, respectively. Other popular SV sources (i.e., CVE, NVD and OWASP) were also mentioned less than 10% on these two sites. Moreover, the fast increase of CSRF (T2), as reported in RQ1, is noteworthy given that this SV type has been removed from the top-10 OWASP since 2013. Our investigation suggested that many developers were aware of CSRF standard prevention techniques, but it was not always easy to apply these techniques and/or use built-in CSRF protection of a web framework (e.g., Spring Security) in practice. These results imply a strong disconnection in the SV patterns between expert-based sources and discussions on Q&A sites, supporting our motivation to study developers' real-life concerns in addressing SVs.

6.5.2 Implications for (Data-Driven) SV Assessment

From RQ1, we found that Q&A sites (i.e., SO and SSE) do not report zero-day SVs, but instead they contain the key SV-related issues that developers have been facing in practice. We have shown that these issues constitute only a small percentage of SVs and SV types reported in expert-based SV sources such as NVD and CWE. This finding suggests that more effort should be directed towards performing assessment, e.g., predicting CVSS metrics, for these commonly encountered types of SVs. It is also recommended to develop assessment models for each of these types rather than generic models for all SV types to improve assessment performance.

[6] https://cwe.mitre.org/top25/
[7] https://owasp.org/www-project-top-ten/

Table 6.8: The mapping between 13 SV topics and their Common Weakness Enumeration (CWE) values.

Topic Name	CWE values
Malwares (T1)	506-512, 904
SQL Injection (T2)	20, 74, 89, 707, 943
Vulnerability Scanning Tools (T3)	–
Cross-site Request Forgery (CSRF) (T4)	352, 1173
File-related Vulnerabilities (T5)	23, 34-36, 61, 434
Synchronization Errors (T6)	362, 662, 667, 820, 821, 833
Encryption Errors (T7)	295, 300, 310
Resource Leaks (T8)	400, 401, 404, 772
Network Attacks (T9)	290, 291, 400
Memory Allocation Errors (T10)	119-127, 787, 822-825, 835
Cross-site Scripting (XSS) (T11)	20, 79-87, 707
Vulnerability Theory (T12)	–
Brute-force/Timing Attacks (T13)	208, 261, 307, 385, 799, 916

Moreover, from RQ1 and RQ2, we found the prevalent, popular and increasing SV types like Brute-force/Timing Attacks, Memory/File-related SVs, Malwares and CSRF. In the context of SV assessment, these common SV types can be prioritized as *custom SV types* for prediction using data-driven models (see section 2.6.1.2 in Chapter 2) because SVs of these types are commonly encountered by developers and have a high possibility of being exploited in the wild.

From RQ2 and RQ3, the SV types (topics) with low difficulty and high expertise may also indicate that it is relatively easy to address these SVs in practice, and vice versa. The difficulty and expertise metrics can be added to the list of currently used technical metrics (e.g., CVSS metrics and time-to-fix SVs in section 2.7.1.3) to approximate the amount of effort required to fix the identified SVs of such types. Such approximation using such external data from Q&A sites is particularly useful for projects that do not have sufficient project-specific SV data for training reliable models to predict SV fixing time.

RQ4 revealed different types of solutions that have been provided to address the SVs. We found that Action to Take and Code Sample are among the most actionable solution types as they can be directly applied or customized to fix SVs found in a codebase. If no solution of these two types can be found for a detected SV, it may suggest that the SV is unique and requires more effort to fix (i.e., developing a solution from scratch).

Overall, the findings demonstrate the potential of incorporating SV data from Q&A sites into (data-driven) SV assessment. However, it is also worth noting that most of the tools (topic T3 in Table 6.4) currently used by developers for detecting and assessing SVs are based on static analysis techniques rather than data-driven models. One reason can be that data-driven approaches are usually black-box compared to static analysis counterparts. For example, many questions related to these tools on Q&A sites were about configuring these tools in specific ways to suite developers' needs, but this is mostly not possible with the current data-driven models. This implies a potential disconnection between state-of-the-art and the state-of-the-practice techniques for SV assessment. Future work is required to make data-driven SV assessment models more interpretable (i.e., why a model makes certain decisions) and configurable (i.e., controling/changing a rule learned by a model) so that these models can be more widely adopted in practice.

6.5.3 Threats to Validity

Our data collection is the first threat. We might have missed some SV posts, but we followed standard techniques in the literature. It is hard to guarantee 100% relevance of the retrieved posts without exhaustive manual validation, which is nearly impossible with more than 70k posts. However, this threat was greatly reduced since the selected posts were carefully checked by three researchers with at least two years of experience in Software Engineering and Cybersecurity.

The identified taxonomies can be another concern. Topic modeling with LDA has been shown effective for processing a large amount of textual posts, but there is still subjectivity in labeling the topics. We mitigated this threat by manually examining at least 30 posts per topic and cross-checking with three of the authors. We also performed a similar manual checking for the answer types in RQ4.

The generalizability of our study may be a threat as well. The patterns we found may not be the same for other Q&A sites and domains. However, the reported patterns for SV discussions on SO and SSE were at least confirmed significant using statistical tests with p-values < 0.01. We also released our code and data at https://github.com/lhmtriet/SV_Empirical_Study for replication and extension to other domains.

6.6 Chapter Summary

Through a large-scale study of 71,329 posts on SO and SSE, we have revealed the support of SV-focused discussions on Q&A sites. Using LDA, we devised 13 commonly discussed SV topics on Q&A sites. We also characterized these topics in terms of their popularity, difficulty, level expertise and solutions received on these sites. Overall, Q&A sites do support SV discussions on SO and SSE, and these discussions shed light on the key SV concerns/types that are of particular interest to developers. Given their importance/prevalence in practice, more priority should be given to assess these SV types. Moreover, data about the popularity, difficulty, expertise level and solution availability on these Q&A sites can be considered for (automatically) assessing the effort required to fix SVs (e.g., more relevant resources/support on Q&A sites may make an SV fix easier). Overall, rich data on crowdsourcing Q&A sites open up various opportunities for the next generation of data-driven SV assessment methods that are better tailored to developers' real-world needs.

6.7 Appendix - PUMiner Overview

In this Appendix, we briefly introduce the PUMiner approach that we used in section 6.3.2 to retrieve security-related posts on SO that can be further refined to obtain the SV-related posts for the presented empirical study in this chapter. We hereby describe the context-aware two-stage PU learning model that forms the core of PUMiner. The complete details and results of PUMiner can be found in the original publication [304].

6.7.1 PUMiner - A Context-aware Two-stage PU Learning Model for Retrieving Security Q&A Posts

PUMiner is a learning model to distinguish security from non-security posts on Q&A websites. The novelty of PUMiner is that this model does not require negative (non-security) posts to perform the prediction, saving significant effort for practitioners as it is non-trivial to define and obtain such non-security posts in practice. Thus, PUMiner operates on security-related and unlabeled posts. There are two main components in PUMiner: (i)

Algorithm 3: Context-aware two-stage PU learning model building.

Input: List of posts: P_{in}
Labels of posts: $labels \in \{positive, unlabeled\}$
Size of embeddings and Window size: sz, ws
Classifier and its model configurations: C, $config$
Output: The trained feature and PU models: feature_model, model$_{PU}$

1 $word\_list, tag\_list \longleftarrow \emptyset, \emptyset$
2 **foreach** $p_i \in P_{in}$ **do**
3 $words, tags \longleftarrow$ tokenize(p_i), extract_tags(p_i)
4 $word\_list, tag\_list \longleftarrow word\_list + \{words\}, tag\_list + \{tags\}$
5 feature_model $\longleftarrow$ train_doc2vec$(word_list, tag_list, sz, ws)$
6 $\mathbf{X}_{in} \longleftarrow$ obtain_feature(feature_model, $word\_list$)
7 $P \longleftarrow \{\mathbf{x}_p | \mathbf{x}_p \in \mathbf{X}_{in} \wedge label(p) = positive\}$
8 $U \longleftarrow \{\mathbf{x}_p | \mathbf{x}_p \in \mathbf{X}_{in} \wedge label(p) = unlabeled\}$
9 $\mathbf{centroid}_P, \mathbf{centroid}_U \longleftarrow \frac{\sum_{p \in P} \mathbf{x}_p}{|P|}, \frac{\sum_{p \in U} \mathbf{x}_p}{|U|}$
10 $RN \longleftarrow \emptyset$
11 **foreach** $\mathbf{x}_i \in \mathbf{X}_{in}$ **do** `// Stage-1 PU: Identify reliable negatives`
12 **if** $d(\mathbf{x}_i, \mathbf{centroid}_U) < \alpha * d(\mathbf{x}_i, \mathbf{centroid}_P)$ **then**
13 $RN \longleftarrow RN + \{\mathbf{x}_i\}$
14 model$_{PU} \longleftarrow$ train_classifier$(C, P, RN, config)$ `// Stage-2 PU`
15 **return** feature_model, model$_{PU}$

feature generation using doc2vec [121] and (*ii*) PU model building using generated features for retrieving security posts. These two components are described hereafter.

PUMiner uses doc2vec [121] to represent input features of posts on Q&A sites for training the PU model. Doc2vec [121] jointly learns the representation/embedding of a document (paragraph vector) with its constituent words. We choose doc2vec to generate embeddings as it can capture the semantic relationship of a word, unlike traditional feature extraction methods such as BoW, n-grams, or tf-idf [21, 400]. Specifically, our work adopts the distributed memory architecture to train doc2vec models as suggested in [121]. Regarding the label of a document/post, a post may contain multiple tags to describe complex concepts, e.g., *php* and *security* tags represent security issues (sql-injection) in PHP. Therefore, besides single tags, we also combine all tags to handle the topic mixture. For example, a post with *php* and *security* tags would have *php_security* alongside *php*, *security* and *post id* as labels. We also sort labels alphabetically to avoid duplicates (e.g., *php_security* and *security_php* are the same).

Next, we present the context-aware two-stage PU learning model (see Algorithm 3) to retrieve security-related posts – the core of our PUMiner framework. This algorithm requires a list of discussion posts with their respective labels (positive or unlabeled), along with the configurations of doc2vec (size of embeddings and window size) and classification models (model hyperparameters). Details of Algorithm 3 are given hereafter.

Lines 1-8: Learning doc2vec context-aware feature vectors. Lines 1-4 tokenize text and extract tags of each post to prepare the data for training doc2vec models. Line 5 trains a doc2vec embedding model to learn the context of words and posts. Line 6 then obtains the embedding vector of each post using the trained doc2vec model. Lines 7 and 8 extract the features of both positive and unlabeled posts, respectively.

Lines 9-13: Stage one of PU learning model. Inspired by PU learning in other domains [401, 402, 403, 404], we assume that the context-aware doc2vec embeddings can

make posts of the same class stay in close proximity in the embedding space. Stage-one PU learning first identifies (reliable/pure) negative (non-security) posts in the unlabeled set that are as different as possible from the positive set. A traditional binary classifier would work poorly in this case since the negative class is not pure [403]. In line 9 of Algorithm 3, we propose to approximately locate unknown negative posts using the centroid (average) vectors of the known positive ($\mathbf{centroid}_P$) and unlabeled ($\mathbf{centroid}_U$) sets, respectively. Since the number of non-security posts is dominant (i.e., up to 96% on Stack Overflow as per our analysis), $\mathbf{centroid}_U$ would represent the negative class more than the positive one. Lines 11-13 compute and compare the cosine distances [202, 121, 403] (see Eq. (6.4)) from each post to the two centroids. If the current post is closer to $\mathbf{centroid}_U$ (i.e., more towards the negative class), it would be selected as a reliable negative.

$$\text{cosine\_distance} = d(i,\, j) = 1 - \frac{\mathbf{p}_i \cdot \mathbf{p}_j}{\|\mathbf{p}_i\| \times \|\mathbf{p}_j\|} \tag{6.4}$$

where $\mathbf{p}_i$ and $\mathbf{p}_j$ are the embedding vectors of the posts i^{th} and j^{th}, respectively. The range of cosine_distance is [0, 2].

We also propose a scaling factor (α) to increase the flexibility of our centroid-based approach, which would be jointly optimized with other hyperparameters of binary classifiers in the second stage. Besides having only one hyperparameter for tuning, this centroid-based approach can incrementally learn new post(s) very fast with a complexity time of $O(1)$, as given in Eq. (6.5).

$$\mathbf{centroid}_{new} = \frac{\mathbf{centroid}_{old} \times N + \mathbf{x}_{new}}{N + \text{size}(\mathbf{x}_{new})} \tag{6.5}$$

where $\mathbf{centroid}_{old}$, $\mathbf{centroid}_{new}$ are the centroid vectors before and after learning new post(s) ($\mathbf{x}_{new}$), while N is the original number of posts in the positive or unlabeled set.

Lines 14-15: Stage two of PU learning model. Using positive and (reliable) negative posts from the first stage, the second stage of PU learning (line 14) trains a binary classifier with its hyperparameters. In the model building process, a PU model is trained with the optimal classifier and its configurations obtained from a validation process (e.g., k-fold cross-validation). Finally, line 15 saves the trained feature and PU models to disk for future inference of whether a post is related to security.

Chapter 7

Conclusions and Future Work

Software Vulnerability (SV) assessment is an integral step of software development. There has been an increasing number of studies on data-driven SV assessment. Data-driven approaches have the benefits of extracting patterns and rules from large-scale historical SV data in the wild without manual definition from experts. Such capability of these approaches enables researchers to automate various SV assessment tasks that were previously not possible with static analysis and rule-based counterparts. Despite the rising interest in data-driven SV assessment, the application of these approaches in practice has been hindered by three key issues: (*i*) concept drift in SV data used for report-level SV assessment, (*ii*) lack of code-based SV assessment, and (*iii*) SV assessment without considering developers' real-world SV needs. This thesis has tackled these three challenges to improve the real-world applicability of data-driven SV assessment. We have first systematized the knowledge and have identified the key practices of the field. Then, we have developed various solutions to enhance the performance and applicability of SV assessment models in real-world settings. These solutions have been based on the customization of recent advances in Machine Learning, Deep Learning (DL) and Natural Language Processing (NLP), as well as the utilization of relevant software artifacts (e.g., source code) and open sources (e.g., Q&A sites) that have been little explored in the literature. We have demonstrated the potential effectiveness and value of our solutions using real-world SV data. Chapter 7 first summarizes the key contributions and findings of this thesis and then suggests several avenues for future research[1] in the emerging area of data-driven SV assessment.

[1]Some of the future research directions presented in this chapter are based on our paper "*A Survey on Data-Driven Software Vulnerability Assessment and Prioritization*", ACM Computing Surveys journal (CORE A*) [11].

7.1 Summary of Contributions and Findings

The significant contributions and key findings of the thesis are summarized as follows.

7.1.1 A Systematization of Knowledge of Data-Driven SV Assessment

In Chapter 2, we have reviewed 84 representative primary studies on data-driven SV assessment to systematize the knowledge of the field. Our review has identified a taxonomy of five key SV assessment tasks/themes that have been automated using data-driven approaches. The themes are predictions of SV (*i*) exploitation (probability, time, and characteristics), (*ii*) impact (confidentiality, integrity, availability, scope, and custom impacts), (*iii*) severity (probability, levels, and score), (*iv*) type (CWE and custom types), as well as (*v*) other tasks (e.g., retrieval of affected product names/vendors, SV knowledge graph construction, and SV fixing effort prediction).

The review has also shed light on the common data-driven practices adopted by these studies to automate SV assessment tasks. Regarding the data sources, NVD [16]/CVE [66] has been most frequently used to provide inputs (mostly SV descriptions) and outputs (e.g., CVSS metrics) for SV assessment models. Moreover, Bag-of-words, Word2vec [202] and more recently BERT [80] have been common methods for extracting features from text-based descriptions of SVs. In addition, linear Support Vector Machine has been the most commonly used for building prediction models, while ensemble Machine Learning (ML) and Deep Learning (DL) models are also on the rise. These models have been mainly evaluated on random-based splits like k-fold cross-validation using common classification metrics (e.g., F1-Score and/or Accuracy) or regression metrics (e.g., Mean absolute error and/or Correlation coefficient).

While data-driven approaches have shown great potential for various SV assessment tasks, we have identified three key issues affecting their practical applicability. Firstly, current studies have mostly used SV descriptions/reports, but they have not investigated the impacts of changing data (concept drift) of these descriptions/reports due to ever-increasing SVs on their models. Such impacts can affect the robustness and even lower the performance of these models over time when deployed in practice. Secondly, existing studies have hardly leveraged (vulnerable) source code in which SVs are usually rooted as inputs for developing data-driven SV assessment models. In practice, using source code directly for SV assessment can alleviate the need for SV reports, which in turn supports more timely SV assessment. Thirdly, the current literature on SV assessment has mainly focused on characteristics of SVs (e.g., exploitability and impacts) rather than real-world concerns encountered by developers while addressing SVs (e.g., difficulty of implementing solutions). Such real-world concerns can facilitate more thorough assessment of SVs by incorporating developers' needs. These three challenges have motivated us to devise respective solutions presented in Chapters 3, 4, 5, and 6.

7.1.2 Automated Report-Level Assessment for Ever-Increasing SVs

In Chapter 3, we have conducted a large-scale analysis of the concept drift issue on report-level SV assessment models (i.e., predicting seven CVSS version 2 base metrics [110]). Using data of 105,124 SVs on NVD, we have first shown that concept drift is indeed an issue in SV descriptions over the years due to releases or discoveries of new software products and cyber-attacks. Concept drift can make k-fold cross-validation, the most commonly used evaluation technique, inflate the validation performance up to 4.7 times compared to that of time-based validation (year-based splits). The main reason is that k-fold cross-validation mixes future SV data during training (i.e., using data of SVs that have not yet been reported at training time), while time-based validation does not. Thus,

we strongly recommend time-based validation instead of k-fold cross-validation for future work on report-level SV assessment.

We have also proposed using subwords extracted from SV descriptions as features for report-level SV assessment models to help these models be robust against Out-of-Vocabulary words caused by concept drift. We have shown that the proposed subword-based models are much more resilient to all the changes in SV descriptions over time, while exhibiting competitive or better performance compared to existing word-only models. We have also demonstrated the possibility of building compact concept-drift-aware SV assessment models (up to 94% reduction in model size while retaining at least 90% of the performance) by using fastText [196, 246]. Overall, we have raised the awareness of the concept drift issue in SV data/reports and proposed an effective data-driven solution to addressing such issue for report-level SV assessment.

7.1.3 Automated Early SV Assessment using Code Functions

In Chapter 4, we have shown the benefits of performing function-level SV assessment (automatically assigning the seven CVSS version 2 metrics to SVs in functions) over report-level SV assessment. We have found that SV reports collected from NVD required for SV assessment usually appear on average 146 days after the fixing time of respective SVs. Such a delay makes report-level SV assessment likely untimely. On the other hand, vulnerable code is available at fixing time by default, thus could be leveraged to support SV assessment even when SV reports are not (yet) available.

Using 1,782 functions of 429 SVs in 200 real-world projects, we have explored the use of vulnerable statements containing root causes of SVs together with other (non-vulnerable) lines in functions as inputs for developing ML models to automate SV assessment. The optimal function-level SV assessment models, on average, have achieved a promising performance of 0.64 Matthews Correlation Coefficient (MCC) and 0.75 F1-Score using all statements in each function (i.e., vulnerable statements alongside all the other context lines). We have highlighted the possibility of performing function-level SV assessment without knowing exactly where vulnerable statements are located in functions. We have also recommended high-performing features and classifiers for this task. Overall, we have distilled the first practices of using data-driven approaches to automate function-level SV assessment to enable earlier fixing prioritization without waiting for SV reports.

7.1.4 Automated Just-in-Time SV Assessment using Code Commits

In Chapter 5, we have motivated the need for SV assessment in code commits to avoid assessment and remediation latencies caused by hidden SVs. We have pointed out that in practice, SVs can stay hidden in codebases for a long time, up to 1,469 days, before being reported. As a result, performing SV assessment in commits, where vulnerable changes are first added to a project, provides just-in-time information about SVs for remediation planning and prioritization. This new assessment task requires a suitable approach that can directly operate with code changes in commits.

We have proposed DeepCVA, a novel deep multi-task learning model, to tackle commit-level SV assessment. Specifically, DeepCVA simultaneously predicts the seven base metrics of CVSS version 2 in a single model using shared context-aware commit features extracted by attention-based convolutional gated recurrent units. Through large-scale experiments on 1,229 vulnerability-contributing commits of 542 different SVs in 246 real-world software projects, we have demonstrated the substantially better performance (38% to 59.8% higher MCC) of DeepCVA than that of many supervised and unsupervised baseline models. Multi-task learning has also enabled DeepCVA to require 6.3 times less time for training and

maintenance than seven cumulative assessment models. With the reported effectiveness and efficiency of DeepCVA, we have made the first promising step towards a holistic solution to assessing SVs as early as they appear, which also contributes to the industrial movement of shift-left security in DevSecOps (securing software adopting the DevOps paradigm) [24].

7.1.5 Insights of Developers' Real-World Concerns on Question and Answer Websites for Data-Driven SV Assessment

As shown in Chapter 2, there has been a deficiency in considering developers' real-world SV-related concerns for SV assessment. Most of the current studies have used expert-defined taxonomies, e.g., CVSS or CWE, as SV assessment outputs. However, these outputs usually do not put an emphasis on the challenges that developers commonly encounter when addressing SVs in practice, e.g., difficulty in implementing a solution proposed by experts. Such lacking considerations can lead to incomplete assessment of SVs, affecting the decision making in prioritizing SV remediation.

Given that developers usually seek solutions to these concerns on Question and Answer (Q&A) sites, in Chapter 6, we have collected 71,329 SV-related posts from two large Q&A sites, namely Stack Overflow (SO) and Security StackExchange (SSE), to identify developers' real-world SV concerns and analyze the support these concerns receive on these sites for SV assessment. We have used Latent Dirichlet Allocation (LDA) [30] to semi-automatically identify 13 commonly discussed SV topics/concerns on SO and SSE. We have discovered that these concerns are only a subset of all SVs/SV types reported by experts and do not follow the patterns of standard rankings like top-10 OWASP or top-25 CWE. Such differences ask for higher priorities and more specific techniques in assessing these commonly encountered SV types. We have also characterized these concerns in terms of their popularity, difficulty, expertise level and solutions received on these sites, which can be used to estimate the complexity/effort required to fix SVs. For instance, an identified SV whose similar SVs have received little support on Q&A sites would imply high remediation difficulty/effort. Overall, crowdsourcing Q&A sites like SO and SSE have much potential for providing supplementary SV assessment metrics from developers' perspectives.

7.2 Opportunities for Future Research

As shown in section 7.1, this thesis has made significant contributions to improve the practicality of data-driven SV assessment. However, there are still many opportunities for future research to further advance the field. Note that this section does not cover straightforward extensions of our work in Chapters 3, 4, 5, and 6 such as using more projects, features, models, and/or programming languages. Rather, we focus on the future research opportunities that have received little/no attention so far in this area.

7.2.1 Integration of SV Data on Issue Tracking Systems

Existing studies, including ours in Chapters 3, have mainly utilized NVD/CVE for collecting SV reports; whereas, *bug/issue tracking systems* like JIRA,[2] Bugzilla[3] or GitHub issues[4] also contain an abundance of SV reports, yet have been underexplored for data-driven SV assessment. Besides providing SV descriptions like CVE/NVD, these issue tracking systems also contain other artifacts such as steps to reproduce, stack traces and test cases that

[2] https://www.atlassian.com/software/jira
[3] https://www.bugzilla.org/
[4] https://docs.github.com/en/issues

give extra information about SVs [405]. However, it is not trivial to obtain and integrate these SV-related bug reports with the ones on SV databases.

One way to retrieve SVs on issue tracking systems is to use security bug reports [406]. Much research work has been put into developing effective models to automatically retrieve security bug reports (e.g., [367, 366, 407]). Among these studies, Wu et al. [407] manually verified and cleaned the security bug reports to provide a clean dataset for automated security bug report identification. However, more of such manual effort is still required to obtain up-to-date data because the original security bug reports in [407] were actually a part of the dataset collected back in 2014 [408].

It is worth noting that *not* all security bug reports are related to SVs such as issues/improvements in implementing security features.[5] Thus, future studies need to filter out these cases before using security bug reports for SV assessment. We also emphasize that some SV-related bug reports are overlapping with the ones on NVD (e.g., the SV report AMBARI-14780[6] on JIRA refers to CVE-2016-0731 on CVE/NVD). Such overlaps would require data cleaning during the integration of reports on issue tracking systems and SV databases to avoid data duplication (e.g., similar SV descriptions) when developing SV assessment models.

7.2.2 Improving Data Efficiency for Data-Driven SV Assessment

As shown in Chapter 2, many of the SV assessment tasks being automated by data-driven approaches, including the prediction of CVSS base metrics in Chapters 3, 4, and 5, suffer from the data imbalance and data scarcity issues. Moreover, existing work as well as our studies in Chapters 3, 4, and 5 have mainly used fully-supervised learning models for automating these tasks, but these models require sufficiently large and fully labeled data to perform well. To address the data-hungriness of these fully-supervised learning models, future studies can approach the SV assessment tasks with *low-shot learning* and/or *semi-supervised learning*.

Low-shot learning a.k.a. *few-shot learning* is designed to perform supervised learning using only a few examples per class, significantly reducing the labeling effort and increasing model robustness against imbalanced data [201]. According to Chapter 2, so far, only one study in this area utilized low-shot learning with a deep Siamese network [150] (i.e., a shared feature model with similarity learning) to effectively predict SV types (CWE) and even generalize to unseen classes (i.e., zero-shot learning). There are still many opportunities for investigating different few-shot learning techniques for other SV assessment tasks, e.g., predicting CVSS metrics. Note that the shared features in few-shot learning can also be enhanced with pre-trained models (e.g., BERT [80]) on another domain/task/project with more labeled data than the current task/project in the SV domain.

Semi-supervised learning enables training models with limited labeled data yet a large amount of unlabeled data [200], potentially leveraging hidden/unlabeled SVs in the wild. Recently, we have seen an increasing interest in using different techniques of this learning paradigm in the SV domain such as collecting SV patches using *multi-view co-training* [332], retrieving SV discussions on developer Q&A sites using *positive-unlabeled learning* [304], curating SVs from multiple sources in the wild using *self-training* [409]. However, it is still little known about the effectiveness of semi-supervised learning for SV assessment tasks.

[5] The security bug report AMBARI-1373 on JIRA (https://issues.apache.org/jira/browse/AMBARI-1373) was about improving the front-end of AMBARI Web by displaying the current logged in user.

[6] https://issues.apache.org/jira/browse/AMBARI-14780

7.2.3 Customized Data-Driven SV Assessment

Similar to the existing studies reviewed in Chapter 2, in Chapters 3, 4, and 5, we have used the standard CVSS metrics [29] for assessing the exploitability, impact and severity levels/score of SVs, but there are increasing concerns that these CVSS outputs are still generic. Specifically, Spring et al. [209] argued that CVSS tends to provide one-size-fits-all assessment metrics regardless of the context of SVs; i.e., the same SVs in different domains/environments are assigned the same metric values. For instance, banking systems may consider the confidentiality and integrity of databases more important than the availability of web/app interfaces.

In the future, alongside CVSS, prediction models should also incorporate the domain/business knowledge to customize the assessment of SVs to a system of interest (e.g., the impact of SVs on critical component(s) and/or the readiness of developers/solutions for mitigating such SVs in the current system). Development environments (e.g., programming language, tools or frameworks) being used are among the key factors that affect the fixing of a specific SV in an organization. Thus, SV-related issues that developers have encountered with these environments discussed on Q&A sites can be considered to enrich SV assessment information. Particularly, future work can leverage the taxonomy we identified in Chapter 6 to automatically match detected SVs with similar issues on Q&A sites, and then follow our framework to indicate the level of support (popularity, difficulty, expertise level, and solution type) these issues have received. The higher the support is, the more likely developers will find relevant information to fix the current SV, which in turn reduces the fixing complexity/effort. In the future, case studies with practitioners will also be fruitful to correlate the quantitative performance of models and their usability/usefulness in real-world systems (e.g., reducing more critical SVs yet using fewer resources).

7.2.4 Enhancing Interpretability of SV Assessment Models

As discussed in Chapter 6, the lack of interpretability is one of the key reasons impeding the adoption of data-driven approaches compared to static analysis tools for SV assessment in practice. Model interpretability is important to increase the transparency of the predictions made by a model, allowing practitioners to adjust the model/data to meet certain requirements [204]. According to Chapter 2, very few reviewed papers in this area (e.g., [97, 21]) have explicitly discussed important features and/or explained why/when their models worked/failed for a task.

SV assessment can draw inspiration from the related SV detection area where the interpretability of (DL-based) prediction models has been actively explored mainly by using (*i*) specific model architectures/parameters or (*ii*) external interpretation models/techniques [204]. In the first approach, prior studies successfully used the feature activation maps in a CNN model [410] or leveraged attention-based neural network [411] to highlight and visualize the important code tokens that contribute to SVs. The second approach uses separate interpretation models on top of trained SV detectors. The interpretation models are either domain/model-agnostic [412], domain-agnostic yet specific to a model type (graph neural network [276]) or SV-specific [413]. The aforementioned approaches produce local/sample-wise interpretation, which can be aggregated to obtain global/task-wise interpretation. The global interpretation is similar to the feature importance of traditional ML models [414] such as the weights of linear models (e.g., Logistic regression) or the (im)purity of nodes split by each feature in tree-based models (e.g., Random forest). However, it is still unclear about the applicability/effectiveness of these approaches for interpreting ML/DL-based SV assessment models, requiring further investigations.

7.2.5 Data-Driven SV Assessment in Data-Driven Systems

Like the current literature (see Chapter 2), this thesis has mainly focused on SV assessment for traditional software, but we envision there is an impending need for SV assessment in data-driven/Artificial Intelligence (AI)-based systems. Data-driven/AI-based systems (e.g., smart recommender systems, chatbots, robots, and autonomous cars) are an emerging breed of systems whose cores are powered by AI technologies, e.g., ML and DL models built on data, rather than human-defined instructions as in traditional systems [193]. However, it is challenging to adapt the current practices of SV assessment to data-driven/AI-based systems. Some of these challenges are presented hereafter.

CVSS [29] is currently the most popular SV assessment framework for traditional systems, but its compatibility with data-driven systems still requires more investigation. The current CVSS documentation lacks instructions on how to assign metrics/score for SVs in data-driven systems. For example, it is unclear how to assign static CVSS metrics to systems with automatically updated data-driven models [409] because adversarial examples for exploitation would likely change after the models are updated. Such ambiguities should be clarified/resolved in future CVSS versions as data-driven systems become more prevalent. The types of SVs in ML/DL models in data-driven systems are also mostly different from the ones provided by CWE [28]. The difference is mainly because these new SVs do not only emerge from configurations/code as in traditional systems, but also from training data and/or trained models [415]. Thus, we recommend that a new category of these SVs should be studied and potentially incorporated into CWE, similar to the newly added category for architectural SVs.[7]

Existing SV assessment models for traditional systems have not considered unique data/model-related characteristics/features of data-driven systems [416]. Specifically, data-driven systems also encompass information about data (e.g., format, type, size and distribution) and ML/DL model(s) (e.g., configurations, parameters and performance). It is worth noting that SVs of ML/DL models in data-driven systems can also come from the frameworks used to develop such models (e.g., Tensorflow[8] or Keras[9]). However, developers of data-driven systems may not be aware of the (security) issues in the used ML/DL frameworks [417]. Thus, besides currently used features, future work should also consider the information about underlying data/models and ML/DL development frameworks to improve the SV representation for building models to assess SVs in data-driven systems.

[7] https://cwe.mitre.org/data/definitions/1008.html
[8] https://github.com/tensorflow/tensorflow
[9] https://github.com/keras-team/keras

References

[1] M. Andreessen, "Why software is eating the world," *Wall Street Journal*, vol. 20, no. 2011, p. C2, 2011.

[2] Wired, "Google is 2 billion lines of code—and it's all in one place." [Online]. Available: https://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/

[3] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, pp. 1–36, 2017.

[4] NIST, "Vulnerability definition on nvd." [Online]. Available: https://nvd.nist.gov/vuln

[5] I. Synopsys, "Heartbleed bug." [Online]. Available: https://heartbleed.com/

[6] T. Conversation, "What is log4j?" [Online]. Available: https://bit.ly/log4j_the_conversation

[7] A. C. S. Centre, "Acsc annual cyber threat report 2020-21." [Online]. Available: https://www.cyber.gov.au/acsc/view-all-content/reports-and-statistics/acsc-annual-cyber-threat-report-2020-21

[8] K. Nayak, D. Marino, P. Efstathopoulos, and T. Dumitraş, "Some vulnerabilities are different than others," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 426–446.

[9] S. Khan and S. Parkinson, "Review into state of the art of vulnerability assessment using artificial intelligence," in *Guide to Vulnerability Analysis for Computer Networks and Systems*. Springer, 2018, pp. 3–32.

[10] V. Smyth, "Software vulnerability management: how intelligence helps reduce the risk," *Network Security*, vol. 2017, no. 3, pp. 10–12, 2017.

[11] T. H. M. Le, H. Chen, and M. A. Babar, "A survey on data-driven software vulnerability assessment and prioritization," *ACM Computing Surveys (CSUR)*, 2021.

[12] P. Foreman, *Vulnerability management*. CRC Press, 2019.

[13] K. Kritikos, K. Magoutis, M. Papoutsakis, and S. Ioannidis, "A survey on vulnerability assessment tools and databases for cloud-based web applications," *Array*, vol. 3, p. 100011, 2019.

[14] M. Z. Bell, "Why expert systems fail," *Journal of the Operational Research Society*, vol. 36, no. 7, pp. 613–619, 1985.

[15] J. Han, M. Kamber, and J. Pei, "Data mining concepts and techniques third edition," *The Morgan Kaufmann Series in Data Management Systems*, vol. 5, no. 4, pp. 83–124, 2011.

[16] NIST, "National vulnerability database." [Online]. Available: https://nvd.nist.gov

[17] ——, "Number of vulnerabilities reported on nvd in 2021." [Online]. Available: http://nvd.nist.gov/vuln/search/statistics?form_type=Basic&results_type=statistics&search_type=all&isCpeNameSearch=false

[18] D. S. Cruzes and T. Dybå, "Research synthesis in software engineering: A tertiary study," *Information and Software Technology*, vol. 53, no. 5, pp. 440–455, 2011.

[19] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM computing surveys (CSUR)*, vol. 46, no. 4, pp. 1–37, 2014.

[20] G. Spanos, L. Angelis, and D. Toloudis, "Assessment of vulnerability severity using text mining," in *the 21st Pan-Hellenic Conference on Informatics*, 2017, pp. 1–6.

[21] Z. Han, X. Li, Z. Xing, H. Liu, and Z. Feng, "Learning to predict severity of software vulnerability using only vulnerability description," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 125–136.

[22] G. Spanos and L. Angelis, "A multi-target approach to estimate software vulnerability characteristics and severity scores," *Journal of Systems and Software*, vol. 146, pp. 152–166, 2018.

[23] T. H. M. Le, B. Sabir, and M. A. Babar, "Automated software vulnerability assessment with concept drift," in *the 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 371–382.

[24] R. N. Rajapakse, M. Zahedi, M. A. Babar, and H. Shen, "Challenges and solutions when adopting devsecops: A systematic review," *Information and Software Technology*, vol. 141, p. 106700, 2022.

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

[26] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[27] Y. Zhang and Q. Yang, "A survey on multi-task learning," *arXiv preprint arXiv:1707.08114*, 2017.

[28] MITRE, "Common weakness enumeration." [Online]. Available: https://cwe.mitre.org

[29] FIRST, "Common vulnerability scoring system." [Online]. Available: https://www.first.org/cvss

[30] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[31] P. Kamongi, S. Kotikela, K. Kavi, M. Gomathisankaran, and A. Singhal, "Vulcan: Vulnerability assessment framework for cloud computing," in *2013 IEEE 7th International Conference on Software Security and Reliability*. IEEE, 2013, pp. 218–226.

[32] M. Bozorgi, L. K. Saul, S. Savage, and G. M. Voelker, "Beyond heuristics: learning to classify vulnerabilities and predict exploits," in *the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2010, pp. 105–114.

[33] C. Sabottke, O. Suciu, and T. Dumitraș, "Vulnerability disclosure in the age of social media: Exploiting twitter for predicting real-world exploits," in *24th {USENIX} Security Symposium*, 2015, pp. 1041–1056.

[34] B. L. Bullough, A. K. Yanchenko, C. L. Smith, and J. R. Zipkin, "Predicting exploitation of disclosed software vulnerabilities using open-source data," in *the 3rd ACM on International Workshop on Security And Privacy Analytics*, 2017, pp. 45–53.

[35] P. Zeng, G. Lin, L. Pan, Y. Tai, and J. Zhang, "Software vulnerability analysis and discovery using deep learning techniques: A survey," *IEEE Access*, 2020.

[36] S. K. Singh and A. Chaturvedi, "Applying deep learning for discovery and analysis of software vulnerabilities: A brief survey," *Soft Computing: Theories and Applications*, pp. 649–658, 2020.

[37] A. O. A. Semasaba, W. Zheng, X. Wu, and S. A. Agyemang, "Literature survey of deep learning-based vulnerability analysis on source code," *IET Software*, 2020.

[38] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: a survey," *the IEEE*, vol. 108, no. 10, pp. 1825–1848, 2020.

[39] J. Pastor-Galindo, P. Nespoli, F. G. Mármol, and G. M. Pérez, "The not yet exploited goldmine of osint: Opportunities, open challenges and future trends," *IEEE Access*, vol. 8, pp. 10 282–10 304, 2020.

[40] J. R. G. Evangelista, R. J. Sassi, M. Romero, and D. Napolitano, "Systematic literature review to investigate the application of open source intelligence (osint) with artificial intelligence," *Journal of Applied Security Research*, pp. 1–25, 2020.

[41] N. Sun, J. Zhang, P. Rimba, S. Gao, L. Y. Zhang, and Y. Xiang, "Data-driven cybersecurity incident prediction: A survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1744–1772, 2018.

[42] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[43] N. Dissanayake, A. Jayatilaka, M. Zahedi, and M. A. Babar, "Software security patch management–a systematic literature review of challenges, approaches, tools and practices," *arXiv preprint arXiv:2012.00544*, 2020.

[44] S. Keele, "Guidelines for performing systematic literature reviews in software engineering," Technical report, Ver. 2.3 EBSE Technical Report. EBSE, Tech. Rep., 2007.

[45] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *the 18th international conference on evaluation and assessment in software engineering*, 2014, pp. 1–10.

[46] M. Edkrantz, "Predicting exploit likelihood for cyber vulnerabilities with machine learning," Master's thesis, 2015.

[47] M. Edkrantz, S. Truvé, and A. Said, "Predicting vulnerability exploits in the wild," in *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*. IEEE, 2015, pp. 513–514.

[48] M. Almukaynizi, E. Nunes, K. Dharaiya, M. Senguttuvan, J. Shakarian, and P. Shakarian, "Proactive identification of exploits in the wild through vulnerability mentions online," in *2017 International Conference on Cyber Conflict (CyCon US)*. IEEE, 2017, pp. 82–88.

[49] ——, "Patch before exploited: An approach to identify targeted software vulnerabilities," in *AI in Cybersecurity*. Springer, 2019, pp. 81–113.

[50] C. Xiao, A. Sarabi, Y. Liu, B. Li, M. Liu, and T. Dumitras, "From patching delays to infection symptoms: Using risk profiles for an early discovery of vulnerabilities exploited in the wild," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 903–918.

[51] N. Tavabi, P. Goyal, M. Almukaynizi, P. Shakarian, and K. Lerman, "Darkembed: Exploit prediction with neural language models," in *the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[52] D. A. de Sousa, E. R. de Faria, and R. S. Miani, "Evaluating the performance of twitter-based exploit detectors," *arXiv preprint arXiv:2011.03113*, 2020.

[53] Y. Fang, Y. Liu, C. Huang, and L. Liu, "Fastembed: Predicting vulnerability exploitation possibility based on ensemble machine learning algorithm," *Plos one*, vol. 15, no. 2, p. e0228439, 2020.

[54] S.-Y. Huang and Y. Wu, "Dynamic software vulnerabilities threat prediction through social media contextual analysis," in *the 15th Asia Conference on Computer and Communications Security*, 2020, pp. 892–894.

[55] J. Jacobs, S. Romanosky, I. Adjerid, and W. Baker, "Improving vulnerability remediation through better exploit prediction," *Journal of Cybersecurity*, vol. 6, no. 1, p. tyaa015, 2020.

[56] J. Yin, M. Tang, J. Cao, and H. Wang, "Apply transfer learning to cybersecurity: Predicting exploitability of vulnerabilities by description," *Knowledge-Based Systems*, vol. 210, p. 106529, 2020.

[57] N. Bhatt, A. Anand, and V. Yadavalli, "Exploitability prediction of software vulnerabilities," *Quality and Reliability Engineering International*, vol. 37, no. 2, pp. 648–663, 2021.

[58] O. Suciu, C. Nelson, Z. Lyu, T. Bao, and T. Dumitras, "Expected exploitability: Predicting the development of functional vulnerability exploits," *arXiv preprint arXiv:2102.07869*, 2021.

[59] A. A. Younis and Y. K. Malaiya, "Using software structure to predict vulnerability exploitation potential," in *2014 IEEE Eighth International Conference on Software Security and Reliability-Companion*. IEEE, 2014, pp. 13–18.

[60] G. Yan, J. Lu, Z. Shu, and Y. Kucuk, "Exploitmeter: Combining fuzzing with machine learning for automated evaluation of software exploitability," in *2017 IEEE Symposium on Privacy-Aware Computing (PAC)*. IEEE, 2017, pp. 164–175.

[61] S. Tripathi, G. Grieco, and S. Rawat, "Exniffer: Learning to prioritize crashes by assessing the exploitability from memory dump," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2017, pp. 239–248.

[62] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 380–394.

[63] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *the Sixth ACM Conference on Data and Application Security and Privacy*, 2016, pp. 85–96.

[64] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 110–121.

[65] L. Zhang and V. L. Thing, "Assisting vulnerability detection by prioritizing crashes with incremental learning," in *TENCON 2018-2018 IEEE Region 10 Conference*. IEEE, 2018, pp. 2080–2085.

[66] MITRE, "Common vulnerabilities and exposures." [Online]. Available: https://cve.mitre.org/

[67] O. Security, "Exploit database." [Online]. Available: https://www.exploit-db.com

[68] Broadcom, "Symantec attack signatures." [Online]. Available: https://bit.ly/symantec_att_sign

[69] Microsoft, "Microsoft security advisories." [Online]. Available: https://bit.ly/ms_sec_advisories

[70] T. Micro, "Zeroday initiative security advisories." [Online]. Available: https://bit.ly/zeroday_sec

[71] Rapid7, "Metasploit security advisories." [Online]. Available: https://www.rapid7.com/db/modules

[72] S. Inc, "Bugtraq vulnerability database." [Online]. Available: http://www.securityfocus.com

[73] R. Future, "Recorded future security advisories." [Online]. Available: https://bit.ly/rf_sec

[74] I. Kenna Security, "Kenna security." [Online]. Available: http://www.kennasecurity.com

[75] ESET, "Eset security advisories." [Online]. Available: https://bit.ly/eset_virus

[76] T. Micro, "Trend micro security advisories." [Online]. Available: https://bit.ly/trend_micro_sec

[77] E. Nunes, A. Diab, A. Gunn, E. Marin, V. Mishra, V. Paliath, J. Robertson, J. Shakarian, A. Thart, and P. Shakarian, "Darknet and deepnet mining for proactive cybersecurity threat intelligence," in *2016 IEEE Conference on Intelligence and Security Informatics (ISI)*. IEEE, 2016, pp. 7–12.

[78] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *the 22nd International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 785–794.

[79] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, pp. 3146–3154, 2017.

[80] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[81] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *the IEEE international conference on computer vision*, 2015, pp. 19–27.

[82] W. Foundation, "Wikipedia pages." [Online]. Available: https://www.wikipedia.org

[83] FIRST, "Cvss version 3.1." [Online]. Available: https://www.first.org/cvss/v3.1/specification-document

[84] M. Bernaschi, E. Gabrielli, and L. V. Mancini, "Remus: A security-enhanced operating system," *ACM Transactions on Information and System Security (TISSEC)*, vol. 5, no. 1, pp. 36–61, 2002.

[85] P. K. Manadhata and J. M. Wing, "An attack surface metric," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 371–386, 2010.

[86] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 1, pp. 26–60, 1990.

[87] CERT, "Basic fuzzing framework." [Online]. Available: https://bit.ly/basic_fuzzing_framework

[88] S. K. Cha, "Ofuzz." [Online]. Available: https://github.com/sangkilc/ofuzz

[89] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer, "Online passive aggressive algorithms," 2006.

[90] J. Jacobs, S. Romanosky, B. Edwards, M. Roytman, and I. Adjerid, "Exploit prediction scoring system (epss)," *arXiv preprint arXiv:1908.04856*, 2019.

[91] J. Jacobs, S. Romanosky, B. Edwards, I. Adjerid, and M. Roytman, "Exploit prediction scoring system (epss)," *Digital Threats: Research and Practice*, vol. 2, no. 3, pp. 1–17, 2021.

[92] H. Chen, R. Liu, N. Park, and V. Subrahmanian, "Using twitter to predict when vulnerabilities will be exploited," in *the 25th International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 3143–3152.

[93] H. Chen, J. Liu, R. Liu, N. Park, and V. Subrahmanian, "Vest: A system for vulnerability exploit scoring & timing," in *IJCAI*, 2019, pp. 6503–6505.

[94] M. Kivelä, A. Arenas, M. Barthelemy, J. P. Gleeson, Y. Moreno, and M. A. Porter, "Multilayer networks," *Journal of complex networks*, vol. 2, no. 3, pp. 203–271, 2014.

[95] Y. Yamamoto, D. Miyamoto, and M. Nakayama, "Text-mining approach for estimating vulnerability score," in *2015 4th International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. IEEE, 2015, pp. 67–73.

[96] T. Wen, Y. Zhang, Y. Dong, and G. Yang, "A novel automatic severity vulnerability assessment framework," *Journal of Communications*, vol. 10, no. 5, pp. 320–329, 2015.

[97] D. Toloudis, G. Spanos, and L. Angelis, "Associating the severity of vulnerabilities with their description," in *International Conference on Advanced Information Systems Engineering*. Springer, 2016, pp. 231–242.

[98] S. Ognawala, R. N. Amato, A. Pretschner, and P. Kulkarni, "Automatically assessing vulnerabilities discovered by compositional analysis," in *the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis*, 2018, pp. 16–25.

[99] S. Ognawala, M. Ochoa, A. Pretschner, and T. Limmer, "Macke: Compositional analysis of low-level vulnerabilities with symbolic execution," in *the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 780–785.

[100] C. Elbaz, L. Rilling, and C. Morin, "Fighting n-day vulnerabilities with automated cvss vector prediction at disclosure," in *the 15th International Conference on Availability, Reliability and Security*, 2020, pp. 1–10.

[101] Y. Jiang and Y. Atif, "An approach to discover and assess vulnerability severity automatically in cyber-physical systems," in *13th International Conference on Security of Information and Networks*, 2020, pp. 1–8.

[102] M. Gawron, F. Cheng, and C. Meinel, "Automatic vulnerability classification using machine learning," in *International Conference on Risks and Security of Internet and Systems*. Springer, 2017, pp. 3–17.

[103] X. Gong, Z. Xing, X. Li, Z. Feng, and Z. Han, "Joint prediction of multiple vulnerability characteristics through multi-task learning," in *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2019, pp. 31–40.

[104] Z. Chen, Y. Zhang, and Z. Chen, "A categorization framework for common computer vulnerabilities and exposures," *The Computer Journal*, vol. 53, no. 5, pp. 551–580, 2010.

[105] J. Ruohonen, "Classifying web exploits with topic modeling," in *2017 28th International Workshop on Database and Expert Systems Applications (DEXA)*. IEEE, 2017, pp. 93–97.

[106] M. U. Aksu, K. Bicakci, M. H. Dilek, A. M. Ozbayoglu, and E. ı. Tatli, "Automated generation of attack graphs using nvd," in *the 8th Conference on Data and Application Security and Privacy*, 2018, pp. 135–142.

[107] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[108] H. Liu and B. Li, "Automated classification of attacker privileges based on deep neural network," in *International Conference on Smart Computing and Communication*. Springer, 2019, pp. 180–189.

[109] K. Kanakogi, H. Washizaki, Y. Fukazawa, S. Ogata, T. Okubo, T. Kato, H. Kanuka, A. Hazeyama, and N. Yoshioka, "Tracing capec attack patterns from cve vulnerability information using natural language processing technique," in *the 54th Hawaii International Conference on System Sciences*, 2021, p. 6996.

[110] FIRST, "Cvss version 2." [Online]. Available: https://www.first.org/cvss/v2/guide

[111] ——, "Cvss version 3." [Online]. Available: https://www.first.org/cvss/v3.0/specification-document

[112] D. M. Blei and J. D. McAuliffe, "Supervised topic models," in *the 20th International Conference on Neural Information Processing Systems*, 2007, p. 121–128.

[113] I. S. Services, "Online database x-force." [Online]. Available: http://www.iss.net/xforce

[114] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[115] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[116] Y. Zhang and Q. Yang, "A survey on multi-task learning," *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[117] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[118] S. Inc., "Secunia vulnerability advisories." [Online]. Available: http://secunia.com

[119] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media, 2009.

[120] Y. Kim, "Convolutional neural networks for sentence classification," in *the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2014, pp. 1746–1751.

[121] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*. PMLR, 2014, pp. 1188–1196.

[122] MITRE, "Common attack pattern enumeration and classification." [Online]. Available: https://capec.mitre.org

[123] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2009.

[124] P. K. Kudjo, J. Chen, M. Zhou, S. Mensah, and R. Huang, "Improving the accuracy of vulnerability report classification using term frequency-inverse gravity moment," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2019, pp. 248–259.

[125] J. Chen, P. K. Kudjo, S. Mensah, S. A. Brown, and G. Akorfu, "An automatic software vulnerability classification framework using term frequency-inverse gravity moment and feature selection," *Journal of Systems and Software*, vol. 167, p. 110616, 2020.

[126] P. K. Kudjo, J. Chen, S. Mensah, R. Amankwah, and C. Kudjo, "The effect of bellwether analysis on software vulnerability severity prediction models," *Software Quality Journal*, pp. 1–34, 2020.

[127] R. Malhotra, "Severity prediction of software vulnerabilities using textual data," in *International Conference on Recent Trends in Machine Learning, IoT, Smart Cities and Applications*. Springer, 2021, pp. 453–464.

[128] K. Chen, Z. Zhang, J. Long, and H. Zhang, "Turning from tf-idf to tf-igm for term weighting in text classification," *Expert Systems with Applications*, vol. 66, pp. 245–260, 2016.

[129] P. Wang, Y. Zhou, B. Sun, and W. Zhang, "Intelligent prediction of vulnerability severity level based on text mining and xgbboost," in *2019 Eleventh International Conference on Advanced Computational Intelligence (ICACI)*. IEEE, 2019, pp. 72–77.

[130] K. Liu, Y. Zhou, Q. Wang, and X. Zhu, "Vulnerability severity prediction with deep neural network," in *2019 5th International Conference on Big Data and Information Analytics (BigDIA)*. IEEE, 2019, pp. 114–119.

[131] R. Sharma, R. Sibal, and S. Sabharwal, "Software vulnerability prioritization using vulnerability description," *International Journal of System Assurance Engineering and Management*, vol. 12, no. 1, pp. 58–64, 2021.

[132] S. E. Sahin and A. Tosun, "A conceptual replication on predicting the severity of software vulnerabilities," in *the Evaluation and Assessment on Software Engineering*, 2019, pp. 244–250.

[133] S. Nakagawa, T. Nagai, H. Kanehara, K. Furumoto, M. Takita, Y. Shiraishi, T. Takahashi, M. Mohri, Y. Takano, and M. Morii, "Character-level convolutional neural network for predicting severity of software vulnerability from vulnerability description," *IEICE Transactions on Information and Systems*, vol. 102, no. 9, pp. 1679–1682, 2019.

[134] X. Zhang, H. Xie, H. Yang, H. Shao, and M. Zhu, "A general framework to understand vulnerabilities in information systems," *IEEE Access*, vol. 8, pp. 121 858–121 873, 2020.

[135] A. Khazaei, M. Ghasemzadeh, and V. Derhami, "An automatic method for cvss score prediction using vulnerabilities description," *Journal of Intelligent & Fuzzy Systems*, vol. 30, no. 1, pp. 89–96, 2016.

[136] G. Spanos, A. Sioziou, and L. Angelis, "Wivss: a new methodology for scoring information systems vulnerabilities," in *the 17th panhellenic conference on informatics*, 2013, pp. 83–90.

[137] S. Lai, L. Xu, K. Liu, and J. Zhao, "Recurrent convolutional neural networks for text classification," in *the AAAI Conference on Artificial Intelligence*, vol. 29, no. 1, 2015.

[138] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[139] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," *arXiv preprint arXiv:1509.01626*, 2015.

[140] H. Chen, J. Liu, R. Liu, N. Park, and V. Subrahmanian, "Vase: A twitter-based vulnerability analysis and score engine," in *2019 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2019, pp. 976–981.

[141] A. Anwar, A. Abusnaina, S. Chen, F. Li, and D. Mohaisen, "Cleaning the nvd: Comprehensive quality assessment, improvements, and analyses," *arXiv preprint arXiv:2006.15074*, 2020.

[142] ——, "Cleaning the nvd: Comprehensive quality assessment, improvements, and analyses," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[143] J. A. Wang and M. Guo, "Vulnerability categorization using bayesian networks," in *the sixth annual workshop on cyber security and information intelligence research*, 2010, pp. 1–4.

[144] B. Shuai, H. Li, M. Li, Q. Zhang, and C. Tang, "Automatic classification for vulnerability based on machine learning," in *2013 IEEE International Conference on Information and Automation (ICIA)*. IEEE, 2013, pp. 312–318.

[145] S. Na, T. Kim, and H. Kim, "A study on the classification of common vulnerabilities and exposures using naïve bayes," in *International Conference on Broadband and Wireless Computing, Communication and Applications*. Springer, 2016, pp. 657–662.

[146] J. Ruohonen and V. Leppänen, "Toward validation of textual information retrieval techniques for software weaknesses," in *International Conference on Database and Expert Systems Applications*. Springer, 2018, pp. 265–277.

[147] G. Huang, Y. Li, Q. Wang, J. Ren, Y. Cheng, and X. Zhao, "Automatic classification method for software vulnerability based on deep neural network," *IEEE Access*, vol. 7, pp. 28 291–28 298, 2019.

[148] M. Aota, H. Kanehara, M. Kubo, N. Murata, B. Sun, and T. Takahashi, "Automation of vulnerability classification from its description using machine learning," in *2020 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2020, pp. 1–7.

[149] E. Aghaei, W. Shadid, and E. Al-Shaer, "Threatzoom: Cve2cwe using hierarchical neural network," *arXiv preprint arXiv:2009.11501*, 2020.

[150] S. S. Das, E. Serra, M. Halappanavar, A. Pothen, and E. Al-Shaer, "V2w-bert: A framework for effective hierarchical multiclass classification of software vulnerabilities," in *2021 IEEE 8th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 2021, pp. 1–12.

[151] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μvuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[152] S. S. Murtaza, W. Khreich, A. Hamou-Lhadj, and A. B. Bener, "Mining trends and patterns of software vulnerabilities," *Journal of Systems and Software*, vol. 117, pp. 218–228, 2016.

[153] Z. Lin, X. Li, and X. Kuang, "Machine learning in vulnerability databases," in *2017 10th International Symposium on Computational Intelligence and Design (ISCID)*, vol. 1. IEEE, 2017, pp. 108–113.

[154] Z. Han, X. Li, H. Liu, Z. Xing, and Z. Feng, "Deepweak: Reasoning common software weaknesses via knowledge graph embedding," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 456–466.

[155] H. S. Venter, J. H. Eloff, and Y. Li, "Standardising vulnerability categories," *Computers & Security*, vol. 27, no. 3-4, pp. 71–83, 2008.

[156] S. Neuhaus and T. Zimmermann, "Security trend analysis with cve topic models," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE, 2010, pp. 111–120.

[157] V. Mounika, X. Yuan, and K. Bandaru, "Analyzing cve database using unsupervised topic modelling," in *2019 International Conference on Computational Science and Computational Intelligence*, 2019, pp. 72–77.

[158] M. Vanamala, X. Yuan, and K. Roy, "Topic modeling and classification of common vulnerabilities and exposures database," in *2020 International Conference on Artificial Intelligence, Big Data, Computing and Data Communication Systems (icABCD)*. IEEE, 2020, pp. 1–5.

[159] W. Aljedaani, Y. Javed, and M. Alenezi, "Lda categorization of security bug reports in chromium projects," in *the 2020 European Symposium on Software Engineering*, 2020, pp. 154–161.

[160] M. A. Williams, S. Dey, R. C. Barranco, S. M. Naim, M. S. Hossain, and M. Akbar, "Analyzing evolving trends of vulnerabilities in national vulnerability database," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 3011–3020.

[161] M. A. Williams, R. C. Barranco, S. M. Naim, S. Dey, M. S. Hossain, and M. Akbar, "A vulnerability analysis and prediction framework," *Computers & Security*, vol. 92, p. 101751, 2020.

[162] E. R. Russo, A. D Sorbo, C. A. Visaggio, and G. Canfora, "Summarizing vulnerabilities' descriptions to support experts during vulnerability assessment activities," *Journal of Systems and Software*, vol. 156, pp. 84–99, 2019.

[163] M. B. Kursa, A. Jankowski, and W. R. Rudnicki, "Boruta–a system for feature selection," *Fundamenta Informaticae*, vol. 101, no. 4, pp. 271–285, 2010.

[164] NIST, "Software assurance reference dataset (sard)." [Online]. Available: https://samate.nist.gov/SRD

[165] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *ACM sigmod record*, vol. 29, no. 2, pp. 1–12, 2000.

[166] T. Kohonen, "The self-organizing map," *the IEEE*, vol. 78, no. 9, pp. 1464–1480, 1990.

[167] OWASP, "Open web application security project." [Online]. Available: https://bit.ly/owasp_main

[168] S. M. Naim, A. P. Boedihardjo, and M. S. Hossain, "A scalable model for tracking topical evolution in large document collections," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 726–735.

[169] R. C. Barranco, A. P. Boedihardjo, and M. S. Hossain, "Analyzing evolving stories in news articles," *International Journal of Data Science and Analytics*, vol. 8, no. 3, pp. 241–256, 2019.

[170] S. Weerawardhana, S. Mukherjee, I. Ray, and A. Howe, "Automated extraction of vulnerability information for home computer security," in *International Symposium on Foundations and Practice of Security*. Springer, 2014, pp. 356–366.

[171] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang, "Towards the detection of inconsistencies in public security vulnerability reports," in *28th {USENIX} Security Symposium*, 2019, pp. 869–885.

[172] D. Gonzalez, H. Hastings, and M. Mirakhorli, "Automated characterization of software vulnerabilities," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 135–139.

[173] NIST, "Vulnerability description ontology." [Online]. Available: https://bit.ly/nist_vdo

[174] H. Binyamini, R. Bitton, M. Inokuchi, T. Yagyu, Y. Elovici, and A. Shabtai, "An automated, end-to-end framework for modeling attacks from vulnerability descriptions," *arXiv preprint arXiv:2008.04377*, 2020.

[175] ——, "A framework for modeling cyber attack techniques from security vulnerability descriptions," in *the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021, pp. 2574–2583.

[176] X. Ou, S. Govindavajhala, and A. W. Appel, "Mulval: A logic-based network security analyzer," in *USENIX security symposium*, vol. 8. Baltimore, MD, 2005, pp. 113–128.

[177] H. Guo, Z. Xing, and X. Li, "Predicting missing information of key aspects in vulnerability reports," *arXiv preprint arXiv:2008.02456*, 2020.

[178] H. Guo, S. Chen, Z. Xing, X. Li, Y. Bai, and J. Sun, "Detecting and augmenting missing key aspects in vulnerability descriptions," *ACM Transactions on Software Engineering and Methodology*, 2021.

[179] E. Wåreus and M. Hell, "Automated cpe labeling of cve summaries with machine learning," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2020, pp. 3–22.

[180] S. Yitagesu, X. Zhang, Z. Feng, X. Li, and Z. Xing, "Automatic part-of-speech tagging for security vulnerability descriptions," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 29–40.

[181] M. Marcus, B. Santorini, and M. A. Marcinkiewicz, "Building a large annotated corpus of english: The penn treebank," 1993.

[182] J. Sun, Z. Xing, H. Guo, D. Ye, X. Li, X. Xu, and L. Zhu, "Generating informative cve description from exploitdb posts by extractive summarization," *arXiv preprint arXiv:2101.01431*, 2021.

[183] S. Horawalavithana, A. Bhattacharjee, R. Liu, N. Choudhury, L. O. Hall, and A. Iamnitchi, "Mentions of security vulnerabilities on reddit, twitter and github," in *IEEE/WIC/ACM International Conference on Web Intelligence*, 2019, pp. 200–207.

[184] H. Xiao, Z. Xing, X. Li, and H. Guo, "Embedding and predicting software security entity relationships: A knowledge graph based approach," in *International Conference on Neural Information Processing*. Springer, 2019, pp. 50–63.

[185] L. B. Othmane, G. Chehrazi, E. Bodden, P. Tsalovski, and A. D. Brucker, "Time for addressing software security issues: Prediction models and impacting factors," *Data Science and Engineering*, vol. 2, no. 2, pp. 107–124, 2017.

[186] J. R. Finkel, T. Grenager, and C. D. Manning, "Incorporating non-local information into information extraction systems by gibbs sampling," in *the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, 2005, pp. 363–370.

[187] S. Özkan, "Cve details." [Online]. Available: https://www.cvedetails.com

[188] SecurityTracker, "Securitytracker vulnerability database." [Online]. Available: https://securitytracker.com

[189] O. Project, "Openwall security advisories." [Online]. Available: https://bit.ly/sec_openwall

[190] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," *arXiv preprint arXiv:1802.05365*, 2018.

[191] MITRE, "Common platform enumeration." [Online]. Available: https://cpe.mitre.org

[192] L. ben Othmane, G. Chehrazi, E. Bodden, P. Tsalovski, A. D. Brucker, and P. Miseldine, "Factors impacting the effort required to fix security vulnerabilities," in *International Conference on Information Security*. Springer, 2015, pp. 102–119.

[193] R. S. S. Kumar, J. Penney, B. Schneier, and K. Albert, "Legal risks of adversarial machine learning research," *arXiv preprint arXiv:2006.16179*, 2020.

[194] H. Zhang, L. Gong, and S. Versteeg, "Predicting bug-fixing time: an empirical study of commercial software projects," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1042–1051.

[195] S. Akbarinasaji, B. Caglayan, and A. Bener, "Predicting bug-fixing time: A replication study using an open source software project," *journal of Systems and Software*, vol. 136, pp. 173–186, 2018.

[196] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[197] B. Sabir, F. Ullah, M. A. Babar, and R. Gaire, "Machine learning for detecting data exfiltration: A review," *ACM Computing Surveys (CSUR)*, vol. 54, no. 3, pp. 1–47, 2021.

[198] D. Hommersom, A. Sabetta, B. Coppola, and D. A. Tamburri, "Automated mapping of vulnerability advisories onto their fix commits in open source repositories," *arXiv preprint arXiv:2103.13375*, 2021.

[199] Broadcom, "Symantec threat explorer." [Online]. Available: https://bit.ly/symantec_threats

[200] J. E. Van Engelen and H. H. Hoos, "A survey on semi-supervised learning," *Machine Learning*, vol. 109, no. 2, pp. 373–440, 2020.

[201] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni, "Generalizing from a few examples: A survey on few-shot learning," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–34, 2020.

[202] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *arXiv preprint arXiv:1310.4546*, 2013.

[203] T. H. M. Le, H. Chen, and M. A. Babar, "Deep learning for source code modeling and generation: Models, applications, and challenges," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–38, 2020.

[204] Y. Zhang, P. Tiňo, A. Leonardis, and K. Tang, "A survey on neural network interpretability," *arXiv preprint arXiv:2012.14261*, 2020.

[205] S. Dzeroski and B. Zenko, "Is combining classifiers better than selecting the best one?" in *ICML*, vol. 2002. Citeseer, 2002, p. 123e30.

[206] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[207] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," *arXiv preprint arXiv:1908.10084*, 2019.

[208] A. Mazuera-Rozo, A. Mojica-Hanke, M. Linares-Vasquez, and G. Bavota, "Shallow or deep? an empirical study on detecting vulnerabilities using deep learning," in *the 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, 2021, pp. 276–287.

[209] J. Spring, E. Hatleback, A. Householder, A. Manion, and D. Shick, "Time to change the cvss?" *IEEE Security & Privacy*, vol. 19, no. 2, pp. 74–78, 2021.

[210] T. Menzies, S. Majumder, N. Balaji, K. Brey, and W. Fu, "500+ times faster than deep learning:(a case study exploring faster methods for text mining stackoverflow)," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018, pp. 554–563.

[211] J.-O. Palacio-Niño and F. Berzal, "Evaluation metrics for unsupervised learning algorithms," *arXiv preprint arXiv:1905.05667*, 2019.

[212] S. Raschka, "Model evaluation, model selection, and algorithm selection in machine learning," *arXiv preprint arXiv:1811.12808*, 2018.

[213] F. G. de Oliveira Neto, R. Torkar, R. Feldt, L. Gren, C. A. Furia, and Z. Huang, "Evolution of statistical analysis in empirical software engineering research: Current state and steps forward," *Journal of Systems and Software*, vol. 156, pp. 246–267, 2019.

[214] Y. Jiao and P. Du, "Performance measures in evaluating machine learning based bioinformatics predictors for classifications," *Quantitative Biology*, vol. 4, no. 4, pp. 320–330, 2016.

[215] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2201–2215.

[216] V. Piantadosi, S. Scalabrino, and R. Oliveto, "Fixing of security vulnerabilities in open source projects: A case study of apache http server and apache tomcat," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 68–78.

[217] B. Ampel, S. Samtani, S. Ullman, and H. Chen, "Linking common vulnerabilities and exposures to the mitre att&ck framework: A self-distillation approach," *arXiv preprint arXiv:2108.01696*, 2021.

[218] P. Kuehn, M. Bayer, M. Wendelborn, and C. Reuter, "Ovana: An approach to analyze and improve the information quality of vulnerability databases," in *The 16th International Conference on Availability, Reliability and Security*, 2021, pp. 1–11.

[219] H. Kekül, B. Ergen, and H. Arslan, "A multiclass hybrid approach to estimating software vulnerability vectors and severity score," *Journal of Information Security and Applications*, vol. 63, p. 103028, 2021.

[220] M. R. Shahid and H. Debar, "Cvss-bert: Explainable natural language processing to determine the severity of a computer security vulnerability from its description," in *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2021, pp. 1600–1607.

[221] J. Lyu, Y. Bai, Z. Xing, X. Li, and W. Ge, "A character-level convolutional neural network for predicting exploitability of vulnerability," in *2021 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE, 2021, pp. 119–126.

[222] I. Babalau, D. Corlatescu, O. Grigorescu, C. Sandescu, and M. Dascalu, "Severity prediction of software vulnerabilities based on their text description," in *2021 23rd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 2021, pp. 171–177.

[223] K. Charmanas, N. Mittas, and L. Angelis, "Predicting the existence of exploitation concepts linked to software vulnerabilities using text mining," in *25th Pan-Hellenic Conference on Informatics*, 2021, pp. 352–356.

[224] J. Yin, M. Tang, J. Cao, H. Wang, M. You, and Y. Lin, "Vulnerability exploitation time prediction: an integrated framework for dynamic imbalanced learning," *World Wide Web*, vol. 25, no. 1, pp. 401–423, 2022.

[225] M. F. Bulut and J. Hwang, "Nl2vul: Natural language to standard vulnerability score for cloud security posture management," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 2021, pp. 566–571.

[226] V. Ramesh, S. Abraham, P. Vinod, I. Mohamed, C. A. Visaggio, and S. Laudanna, "Automatic classification of vulnerabilities using deep learning and machine learning algorithms," in *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2021, pp. 1–8.

[227] G. Aivatoglou, M. Anastasiadis, G. Spanos, A. Voulgaridis, K. Votis, and D. Tzovaras, "A tree-based machine learning methodology to automatically classify software vulnerabilities," in *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*. IEEE, 2021, pp. 312–317.

[228] P. Vishnu, P. Vinod, and S. Y. Yerima, "A deep learning approach for classifying vulnerability descriptions using self attention based neural network," *Journal of Network and Systems Management*, vol. 30, no. 1, pp. 1–27, 2022.

[229] Q. Wang, Y. Li, Y. Wang, and J. Ren, "An automatic algorithm for software vulnerability classification based on cnn and gru," *Multimedia Tools and Applications*, pp. 1–22, 2022.

[230] V. Yosifova, A. Tasheva, and R. Trifonov, "Predicting vulnerability type in common vulnerabilities and exposures (cve) database with machine learning classifiers," in *2021 12th National Conference with International Participation (ELECTRONICA)*. IEEE, 2021, pp. 1–6.

[231] G. Yang, S. Dineen, Z. Lin, and X. Liu, "Few-sample named entity recognition for security vulnerability reports by fine-tuning pre-trained language models," in *International Workshop on Deployable Machine Learning for Security Defense*. Springer, 2021, pp. 55–78.

[232] S. Yitagesu, Z. Xing, X. Zhang, Z. Feng, X. Li, and L. Han, "Unsupervised labeling and extraction of phrase-based concepts in vulnerability descriptions," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 943–954.

[233] L. Yuan, Y. Bai, Z. Xing, S. Chen, X. Li, and Z. Deng, "Predicting entity relations across different security databases by using graph attention network," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2021, pp. 834–843.

[234] H. Guo, Z. Xing, S. Chen, X. Li, Y. Bai, and H. Zhang, "Key aspects augmentation of vulnerability description based on multiple security databases," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2021, pp. 1020–1025.

[235] M.-T. Luong, I. Sutskever, Q. V. Le, O. Vinyals, and W. Zaremba, "Addressing the rare word problem in neural machine translation," *arXiv preprint arXiv:1410.8206*, 2014.

[236] C.-C. Huang, H.-C. Yen, P.-C. Yang, S.-T. Huang, and J. S. Chang, "Using sublexical translations to handle the oov problem in machine translation," *ACM Transactions on Asian Language Information Processing (TALIP)*, vol. 10, no. 3, pp. 1–20, 2011.

[237] A. Liu and K. Kirchhoff, "Context models for oov word translation in low-resource languages," *arXiv preprint arXiv:1801.08660*, 2018.

[238] M. Razmara, M. Siahbani, R. Haffari, and A. Sarkar, "Graph propagation for paraphrasing out-of-vocabulary words in statistical machine translation," in *the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2013, pp. 1105–1115.

[239] R. Future, "Exploiting old vulnerabilities." [Online]. Available: https://www.recordedfuture.com/exploiting-old-vulnerabilities/

[240] A. Kao and S. R. Poteet, *Natural language processing and text mining*. Springer Science & Business Media, 2007.

[241] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[242] E. Loper and S. Bird, "Nltk: the natural language toolkit," *arXiv preprint cs/0205028*, 2002.

[243] M. F. Porter, "An algorithm for suffix stripping." *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[244] C. Bergmeir and J. M. Benítez, "On the use of cross-validation for time series predictor evaluation," *Information Sciences*, vol. 191, pp. 192–213, 2012.

[245] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, pp. 391–407, 1990.

[246] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," *arXiv preprint arXiv:1607.01759*, 2016.

[247] Kaggle, "Kaggle." [Online]. Available: https://www.kaggle.com/

[248] S. Russell and P. Norvig, "Artificial intelligence: a modern approach," 2002.

[249] S. H. Walker and D. B. Duncan, "Estimation of the probability of an event as a function of several independent variables," *Biometrika*, vol. 54, no. 1-2, pp. 167–179, 1967.

[250] A. Basu, C. Walters, and M. Shepherd, "Support vector machines for text categorization," in *36th Annual Hawaii International Conference on System Sciences, 2003. the.* IEEE, 2003, pp. 7–pp.

[251] T. K. Ho, "Random decision forests," in *3rd international conference on document analysis and recognition*, vol. 1. IEEE, 1995, pp. 278–282.

[252] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth, "Occam's razor," *Information processing letters*, vol. 24, no. 6, pp. 377–380, 1987.

[253] T. H. M. Le, T. T. Tran, and L. K. Huynh, "Identification of hindered internal rotational mode for complex chemical species: A data mining approach with multivariate logistic regression model," *Chemometrics and Intelligent Laboratory Systems*, vol. 172, pp. 10–16, 2018.

[254] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in Statistics.* Springer, 1992, pp. 196–202.

[255] D. Marutho, S. H. Handaka, E. Wijaya *et al.*, "The determination of cluster number at k-mean using elbow method and purity evaluation on headline news," in *2018 international seminar on application for technology of information and communication.* IEEE, 2018, pp. 533–538.

[256] R. Rehurek and P. Sojka, "Software framework for topic modelling with large corpora," in *In the LREC 2010 Workshop on New Challenges for NLP Frameworks.* Citeseer, 2010.

[257] Facebook, "Pre-trained word vectors of fasttext." [Online]. Available: https://github.com/facebookresearch/fastText/blob/master/pretrained-vectors.md

[258] Y. Hou, X. Ren, Y. Hao, T. Mo, and W. Li, "A security vulnerability threat classification method," in *International Conference on Broadband and Wireless Computing, Communication and Applications.* Springer, 2017, pp. 414–426.

[259] Q. Liu and Y. Zhang, "Vrss: A new system for rating and scoring vulnerabilities," *Computer Communications*, vol. 34, no. 3, pp. 264–273, 2011.

[260] R. Sharma and R. Singh, "An improved scoring system for software vulnerability prioritization," in *Quality, IT and Business Operations.* Springer, 2018, pp. 33–43.

[261] P. Johnson, R. Lagerström, M. Ekstedt, and U. Franke, "Can the common vulnerability scoring system be trusted? a bayesian analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 6, pp. 1002–1015, 2016.

[262] C.-C. Huang, F.-Y. Lin, F. Y.-S. Lin, and Y. S. Sun, "A novel approach to evaluate software vulnerability prioritization," *Journal of Systems and Software*, vol. 86, no. 11, pp. 2822–2840, 2013.

[263] Y. Roumani, J. K. Nwankpa, and Y. F. Roumani, "Time series modeling of vulnerabilities," *Computers & Security*, vol. 51, pp. 32–40, 2015.

[264] M. Tang, M. Alazab, and Y. Luo, "Exploiting vulnerability disclosures: statistical framework and case study," in *2016 Cybersecurity and Cyberforensics Conference (CCC)*. IEEE, 2016, pp. 117–122.

[265] S. M. Rajasooriya, C. P. Tsokos, and P. K. Kaluarachchi, "Cyber security: Nonlinear stochastic models for predicting the exploitability," *Journal of information Security*, vol. 8, no. 2, pp. 125–140, 2017.

[266] P. K. Kaluarachchi, C. P. Tsokos, and S. M. Rajasooriya, "Non-homogeneous stochastic model for cyber security predictions," *Journal of Information Security*, vol. 9, no. 1, pp. 12–24, 2017.

[267] N. R. Pokhrel, H. Rodrigo, C. P. Tsokos *et al.*, "Cybersecurity: time series predictive modeling of vulnerabilities of desktop operating system using linear and non-linear approach," *Journal of Information Security*, vol. 8, no. 04, p. 362, 2017.

[268] T. H. M. Le and M. A. Babar, "On the use of fine-grained vulnerable code statements for software vulnerability assessment models," *arXiv preprint arXiv:2203.08417*, 2022.

[269] M. Foundation, "Bugzilla issue tracking system." [Online]. Available: https://www.bugzilla.org/

[270] R. Croft, A. Babar, and L. Li, "An investigation into inconsistency of software vulnerability severity across data sources," in *2022 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022.

[271] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[272] B. Aloraini, M. Nagappan, D. M. German, S. Hayashi, and Y. Higo, "An empirical study of security warnings from static application security testing tools," *Journal of Systems and Software*, vol. 158, p. 110427, 2019.

[273] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *arXiv preprint arXiv:1909.03496*, 2019.

[274] W. Zheng, J. Gao, X. Wu, F. Liu, Y. Xun, G. Liu, and X. Chen, "The impact factors on the performance of machine learning-based vulnerability detection: A comparative study," *Journal of Systems and Software*, vol. 168, p. 110659, 2020.

[275] G. Lin, W. Xiao, J. Zhang, and Y. Xiang, "Deep learning-based vulnerable function detection: A benchmark," in *International Conference on Information and Communications Security*, 2020, pp. 219–232.

[276] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2021.

[277] V. Nguyen, T. Le, O. De Vel, P. Montague, J. Grundy, and D. Phung, "Information-theoretic source code vulnerability highlighting," in *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2021, pp. 1–8.

[278] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "Vuldeelocator: a deep learning-based fine-grained vulnerability detector," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[279] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto, "Predicting defective lines using a model-agnostic technique," *IEEE Transactions on Software Engineering*, 2020.

[280] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *The ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 602–614.

[281] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, "Finding a needle in a haystack: Automated mining of silent vulnerability fixes," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[282] Atlassian, "Jira issue tracking system." [Online]. Available: https://www.atlassian.com/software/jira

[283] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[284] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[285] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 383–387.

[286] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 412–428, 2017.

[287] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "Deepjit: an end-to-end deep learning framework for just-in-time defect prediction," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 34–45.

[288] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[289] T. H. M. Le, D. Hin, R. Croft, and M. A. Babar, "Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 717–729.

[290] C. Rezk, Y. Kamei, and S. Mcintosh, "The ghost commit problem when identifying fix-inducing changes: An empirical study of apache projects," *IEEE Transactions on Software Engineering*, 2021.

[291] W. G. Cochran, *Sampling techniques*. John Wiley & Sons, 2007.

[292] A. J. Viera, J. M. Garrett *et al.*, "Understanding interobserver agreement: the kappa statistic," *Fam med*, vol. 37, no. 5, pp. 360–363, 2005.

[293] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, "Evaluating representation learning of code changes for predicting patch correctness in program repair," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 981–992.

[294] W. Zheng, Y. Jiang, and X. Su, "Vulspg: Vulnerability detection based on slice property graph representation learning," *arXiv preprint arXiv:2109.02527*, 2021.

[295] S. Salimi, M. Ebrahimzadeh, and M. Kharrazi, "Improving real-world vulnerability characterization with vulnerable slices," in *The 16th ACM International Conference on Predictive Models and Data Analytics in Software Engineering*, 2020, pp. 11–20.

[296] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[297] S. Dashevskyi, A. D. Brucker, and F. Massacci, "A screening test for disclosed vulnerabilities in foss components," *IEEE Transactions on Software Engineering*, vol. 45, no. 10, pp. 945–966, 2018.

[298] D. Shen, G. Wang, W. Wang, M. R. Min, Q. Su, Y. Zhang, C. Li, R. Henao, and L. Carin, "Baseline needs more love: On simple word-embedding-based models and associated pooling mechanisms," *arXiv preprint arXiv:1805.09843*, 2018.

[299] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[300] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[301] X. Zhou, D. Han, and D. Lo, "Assessing generalizability of codebert," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 425–436.

[302] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 518–529.

[303] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[304] T. H. M. Le, D. Hin, R. Croft, and M. A. Babar, "Puminer: Mining security posts from developer question and answer websites with pu learning," in *the 17th International Conference on Mining Software Repositories*, 2020, pp. 350–361.

[305] H. Hanif, M. H. N. M. Nasir, M. F. Ab Razak, A. Firdaus, and N. B. Anuar, "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches," *Journal of Network and Computer Applications*, p. 103009, 2021.

[306] T. Sonnekalb, T. S. Heinze, and P. Mäder, "Deep security analysis of program code," *Empirical Software Engineering*, vol. 27, no. 1, pp. 1–39, 2022.

[307] A. Luque, A. Carrasco, A. Martín, and A. de las Heras, "The impact of class imbalance in classification performance metrics based on the binary confusion matrix," *Pattern Recognition*, vol. 91, pp. 216–231, 2019.

[308] M. Tomczak and E. Tomczak, "The need to report effect size estimates revisited. an overview of some recommended measures of effect size," *Trends in Sport Sciences*, vol. 1, no. 21, pp. 19–25, 2014.

[309] A. Field, *Discovering statistics using IBM SPSS statistics*. sage, 2013.

[310] J. Cohen, *Statistical power analysis for the behavioral sciences*. Academic press, 2013.

[311] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma, "Cliff's delta calculator: A nonparametric effect size program for two groups of observations," *Universitas Psychologica*, vol. 10, no. 2, pp. 545–555, 2011.

[312] C. Treude and M. Wagner, "Predicting good configurations for github and stack overflow topic models," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 84–95.

[313] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, "The impact of class rebalancing techniques on the performance and interpretation of defect prediction models," *IEEE Transactions on Software Engineering*, vol. 46, no. 11, pp. 1200–1219, 2018.

[314] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[315] T. H. M. Le, R. Croft, D. Hin, and M. A. Babar, "A large-scale study of security vulnerability support on developer q&a websites," in *Evaluation and Assessment in Software Engineering*, 2021, pp. 109–118.

[316] J. Kim, T. Kim, and E. G. Im, "Survey of dynamic taint analysis," in *2014 4th IEEE International Conference on Network Infrastructure and Digital Content*. IEEE, 2014, pp. 269–272.

[317] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *The 14th ACM Conference on Computer and Communications Security*, 2007, pp. 529–540.

[318] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294–313, 2011.

[319] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Software Engineering*, vol. 18, no. 1, pp. 25–59, 2013.

[320] Y. Tang, F. Zhao, Y. Yang, H. Lu, Y. Zhou, and B. Xu, "Predicting vulnerable components via text mining or software metrics? an effort-aware perspective," in *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2015, pp. 27–36.

[321] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, and P. Montague, "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3289–3297, 2018.

[322] V. Nguyen, T. Le, T. Le, K. Nguyen, O. DeVel, P. Montague, L. Qu, and D. Phung, "Deep domain adaptation for vulnerable code function identification," in *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2019, pp. 1–8.

[323] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay, "Vulnerability prediction from source code using machine learning," *IEEE Access*, vol. 8, pp. 150 672–150 684, 2020.

[324] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang, "Combining graph-based learning with automated data collection for code vulnerability detection," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1943–1958, 2020.

[325] Y. Ding, S. Suneja, Y. Zheng, J. Laredo, A. Morari, G. Kaiser, and B. Ray, "Velvet: a novel ensemble learning approach to automatically locate vulnerable statements," in *2022 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022.

[326] X. Duan, M. Ge, T. H. M. Le, F. Ullah, S. Gao, X. Lu, and M. A. Babar, "Automated security assessment for the internet of things," in *2021 IEEE 26th Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, 2021, pp. 47–56.

[327] S. E. Ponta, H. Plate, and A. Sabetta, "Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 449–460.

[328] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010, pp. 1–10.

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

[330] L. Yang, X. Li, and Y. Yu, "Vuldigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes," in *GLOBECOM 2017-2017 IEEE Global Communications Conference*. IEEE, 2017, pp. 1–7.

[331] L. G. A. Rodriguez, J. S. Trazzi, V. Fossaluza, R. Campiolo, and D. M. Batista, "Analysis of vulnerability disclosure delays from the national vulnerability database," in *Anais do I Workshop de Segurança Cibernética em Dispositivos Conectados*. SBC, 2018.

[332] A. D. Sawadogo, T. F. Bissyandé, N. Moha, K. Allix, J. Klein, L. Li, and Y. L. Traon, "Learning to catch security patches," *arXiv preprint arXiv:2001.09148*, 2020.

[333] F. Thung, D. Lo, L. Jiang, F. Rahman, P. T. Devanbu *et al.*, "When would this bug get reported?" in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 420–429.

[334] A. Bosu and J. C. Carver, "Peer code review in open source communities using reviewboard," in *the ACM 4th Annual Workshop on Evaluation and Usability of Programming Languages and Tools*, 2012, pp. 17–24.

[335] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Investigating code review practices in defective files: An empirical study of the qt system," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 168–179.

[336] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk, "Auto-completing bug reports for android applications," in *the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 673–686.

[337] T. Hoang, J. Lawall, Y. Tian, R. J. Oentaryo, and D. Lo, "Patchnet: Hierarchical deep learning-based stable patch identification for the linux kernel," *IEEE Transactions on Software Engineering*, 2019.

[338] S. Chowdhuri, T. Pankaj, and K. Zipser, "Multinet: Multi-modal multi-task learning for autonomous driving," in *2019 IEEE Winter Conference on Applications of Computer Vision*. IEEE, 2019, pp. 1496–1504.

[339] A. Sabetta and M. Bezzi, "A practical approach to the automatic classification of security-relevant commits," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 579–582.

[340] E. Sahal and A. Tosun, "Identifying bug-inducing changes for code additions," in *the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2018, pp. 1–2.

[341] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Icml*, 2010.

[342] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[343] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[344] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[345] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.

[346] H. Hata, C. Treude, R. G. Kula, and T. Ishio, "9.6 million links in source code comments: Purpose, evolution, and decay," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1211–1221.

[347] Y. Fan, X. Xia, D. A. Da Costa, D. Lo, A. E. Hassan, and S. Li, "The impact of changes mislabeled by szz on just-in-time defect prediction," *IEEE Transactions on Software Engineering*, 2019.

[348] D. Falessi, J. Huang, L. Narayana, J. F. Thai, and B. Turhan, "On the need of preserving order of data when validating within-project defect classifiers," *Empirical Software Engineering*, vol. 25, no. 6, pp. 4805–4830, 2020.

[349] M. Jimenez, R. Rwemalika, M. Papadakis, F. Sarro, Y. Le Traon, and M. Harman, "The importance of accounting for real-world labelling when predicting software vulnerabilities," in *2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 695–705.

[350] J. Gorodkin, "Comparing two k-category assignments by a k-category correlation coefficient," *Computational Biology and Chemistry*, vol. 28, no. 5-6, pp. 367–374, 2004.

[351] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[352] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.

[353] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[354] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[355] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*. PMLR, 2015, pp. 448–456.

[356] S. Lloyd, "Least squares quantization in pcm," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[357] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 914–919.

[358] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin, "A comparative study of deep learning-based vulnerability detection system," *IEEE Access*, vol. 7, pp. 103 184–103 197, 2019.

[359] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[360] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *Journal of Systems and Software*, vol. 150, pp. 22–36, 2019.

[361] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.

[362] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 200–210.

[363] S. E. Ponta, H. Plate, and A. Sabetta, "Detection, assessment and mitigation of vulnerabilities in open source dependencies," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3175–3215, 2020.

[364] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2015, pp. 17–26.

[365] F. Camilo, A. Meneely, and M. Nagappan, "Do bugs foreshadow vulnerabilities? a study of the chromium project," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 269–279.

[366] F. Peters, T. T. Tun, Y. Yu, and B. Nuseibeh, "Text filtering and ranking for security bug report prediction," *IEEE Transactions on Software Engineering*, vol. 45, no. 6, pp. 615–631, 2017.

[367] M. Gegick, P. Rotella, and T. Xie, "Identifying security bug reports via text mining: An industrial case study," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010, pp. 11–20.

[368] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, "Identifying the characteristics of vulnerable code changes: An empirical study," in *the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 257–268.

[369] X. Chen, Y. Zhao, Z. Cui, G. Meng, Y. Liu, and Z. Wang, "Large-scale empirical studies on effort-aware security vulnerability prediction methods," *IEEE Transactions on Reliability*, vol. 69, no. 1, pp. 70–87, 2019.

[370] X.-L. Yang, D. Lo, X. Xia, Z.-Y. Wan, and J.-L. Sun, "What security questions do developers ask? a large-scale study of stack overflow posts," *Journal of Computer Science and Technology*, vol. 31, no. 5, pp. 910–924, 2016.

[371] S. Bayati and M. Heidary, "Information security in software engineering, analysis of developers communications about security in social q&a website," in *Pacific-Asia Workshop on Intelligence and Security Informatics*. Springer, 2016, pp. 193–202.

[372] T. Lopez, T. Tun, A. Bandara, L. Mark, B. Nuseibeh, and H. Sharp, "An anatomy of security conversations in stack overflow," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*. IEEE, 2019, pp. 31–40.

[373] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? an analysis of topics and trends in stack overflow," *Empirical Software Engineering*, vol. 19, no. 3, pp. 619–654, 2014.

[374] S. Ahmed and M. Bagherzadeh, "What do concurrency developers ask about? a large-scale study using stack overflow," in *the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2018, pp. 1–10.

[375] C. Rosen and E. Shihab, "What are mobile developers asking about? a large scale study using stack overflow," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1192–1223, 2016.

[376] M. Bagherzadeh and R. Khatchadourian, "Going big: a large-scale study on what big data developers ask," in *the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 432–442.

[377] A. A. Bangash, H. Sahar, S. Chowdhury, A. W. Wong, A. Hindle, and K. Ali, "What do developers know about machine learning: a study of ml discussions on stackoverflow," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 260–264.

[378] J. Han, E. Shihab, Z. Wan, S. Deng, and X. Xia, "What do programmers discuss about deep learning frameworks," *Empirical Software Engineering*, vol. 25, no. 4, pp. 2694–2747, 2020.

[379] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 771–781.

[380] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. A. Argoty, "Secure coding practices in java: Challenges and vulnerabilities," in *the 40th International Conference on Software Engineering*, 2018, pp. 372–383.

[381] A. Rahman, E. Farhana, and N. Imtiaz, "Snakes in paradise?: Insecure python-related coding practices in stack overflow," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 200–204.

[382] B. V. Hanrahan, G. Convertino, and L. Nelson, "Modeling problem difficulty and expertise in stackoverflow," in *the ACM 2012 conference on Computer Supported Cooperative Work Companion*, 2012, pp. 91–94.

[383] T. Dey, A. Karnauch, and A. Mockus, "Representation of developer expertise in open source software," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021.

[384] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on software engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[385] Z. Chen, Y. Cao, Y. Liu, H. Wang, T. Xie, and X. Liu, "A comprehensive study on challenges in deploying deep learning based software," in *the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 750–762.

[386] C. Treude, O. Barzilay, and M.-A. Storey, "How do programmers ask and answer questions on the web?(nier track)," in *the 33rd international conference on software engineering*, 2011, pp. 804–807.

[387] J. Bekker and J. Davis, "Learning from positive and unlabeled data: A survey," *arXiv preprint arXiv:1811.04820*, 2018.

[388] B. Schölkopf, R. C. Williamson, A. J. Smola, J. Shawe-Taylor, and J. C. Platt, "Support vector method for novelty detection," in *Advances in neural information processing systems*, 2000, pp. 582–588.

[389] L. M. Manevitz and M. Yousef, "One-class svms for document classification," *Journal of machine Learning research*, vol. 2, no. Dec, pp. 139–154, 2001.

[390] O. Mendsaikhan, H. Hasegawa, Y. Yamaguchi, and H. Shimada, "Identification of cybersecurity specific content using the doc2vec language model," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2019, pp. 396–401.

[391] M. U. Haque, L. H. Iwaya, and M. A. Babar, "Challenges in docker development: A large-scale study using stack overflow," in *the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–11.

[392] M. Röder, A. Both, and A. Hinneburg, "Exploring the space of topic coherence measures," in *the eighth ACM international conference on Web search and data mining*, 2015, pp. 399–408.

[393] A. Abdellatif, D. Costa, K. Badran, R. Abdalkareem, and E. Shihab, "Challenges in chatbot development: A study of stack overflow posts," in *the 17th International Conference on Mining Software Repositories*, 2020, pp. 174–185.

[394] M. Zahedi, R. N. Rajapakse, and M. A. Babar, "Mining questions asked about continuous software engineering: A case study of stack overflow," in *the Evaluation and Assessment in Software Engineering*, 2020, pp. 41–50.

[395] H. B. Mann, "Nonparametric tests against trend," *Econometrica: Journal of the Econometric Society*, pp. 245–259, 1945.

[396] W. R. Knight, "A computer method for calculating kendall's tau with ungrouped data," *Journal of the American Statistical Association*, vol. 61, no. 314, pp. 436–439, 1966.

[397] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[398] E. Database, "Exploit database." [Online]. Available: www.exploit-db.com

[399] M. Zahedi, M. Ali Babar, and C. Treude, "An empirical study of security issues posted in open source projects," in *the 51st Hawaii International Conference on System Sciences*, 2018.

[400] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 334–345.

[401] C. Elkan and K. Noto, "Learning classifiers from only positive and unlabeled data," in *the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2008, pp. 213–220.

[402] F. Mordelet and J.-P. Vert, "A bagging svm to learn from positive and unlabeled examples," *Pattern Recognition Letters*, vol. 37, pp. 201–209, 2014.

[403] X. Li and B. Liu, "Learning to classify texts using positive and unlabeled data," in *IJCAI*, vol. 3, no. 2003, 2003, pp. 587–592.

[404] D. H. Fusilier, M. Montes-y Gómez, P. Rosso, and R. G. Cabrera, "Detecting positive and negative deceptive opinions using pu-learning," *Information processing & management*, vol. 51, no. 4, pp. 433–443, 2015.

[405] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss, "What makes a good bug report?" *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 618–643, 2010.

[406] F. A. Bhuiyan, M. B. Sharif, and A. Rahman, "Security bug report usage for software vulnerability research: A systematic mapping study," *IEEE Access*, vol. 9, pp. 28 471–28 495, 2021.

[407] X. Wu, W. Zheng, X. Xia, and D. Lo, "Data quality matters: A case study on data label correctness for security bug report prediction," *IEEE Transactions on Software Engineering*, 2021.

[408] M. Ohira, Y. Kashiwa, Y. Yamatani, H. Yoshiyuki, Y. Maeda, N. Limsettho, K. Fujino, H. Hata, A. Ihara, and K. Matsumoto, "A dataset of high impact bugs: Manually-classified issue reports," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 518–521.

[409] Y. Chen, A. E. Santosa, A. M. Yi, A. Sharma, A. Sharma, and D. Lo, "A machine learning approach for vulnerability curation," in *the 17th International Conference on Mining Software Repositories*, 2020, pp. 32–42.

[410] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 757–762.

[411] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu, "Vulsniper: Focus your attention to shoot fine-grained vulnerabilities," in *IJCAI*, 2019, pp. 4665–4671.

[412] A. Warnecke, D. Arp, C. Wressnegger, and K. Rieck, "Evaluating explanation methods for deep learning in security," in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2020, pp. 158–174.

[413] D. Zou, Y. Zhu, S. Xu, Z. Li, H. Jin, and H. Ye, "Interpreting deep learning-based vulnerability detector predictions based on heuristic searching," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 2, pp. 1–31, 2021.

[414] G. Chandrashekar and F. Sahin, "A survey on feature selection methods," *Computers & Electrical Engineering*, vol. 40, no. 1, pp. 16–28, 2014.

[415] I. Rosenberg, A. Shabtai, Y. Elovici, and L. Rokach, "Adversarial machine learning attacks and defense methods in the cyber security domain," *ACM Computing Surveys (CSUR)*, vol. 54, no. 5, pp. 1–36, 2021.

[416] Z. Wan, X. Xia, D. Lo, and G. C. Murphy, "How does machine learning change software development practices?" *IEEE Transactions on Software Engineering*, 2019.

[417] J. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, "Is using deep learning frameworks free? characterizing technical debt in deep learning frameworks," in *the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society*, 2020, pp. 1–10.