A Security Requirements Management Framework for Open-Source Software Projects

A Dissertation submitted to the
Graduate School
of the University of Cincinnati
in partial fulfillment of the
requirements for the degree of
DOCTOR OF PHILOSOPHY
at the
UNIVERSITY OF CINCINNATI
COLLEGE OF ENGINEERING AND APPLIED SCIENCE
June, 2019
by
Wentao Wang

Thesis advisor and Committee chair:
Nan Niu, Ph.D
Abstract

Requirements in open-source software (OSS) projects are recorded in light-weight representation and are not fully discussed until they become part of an active sprint, as needed. We call this new fashion of requirements engineering (RE) as just-in-time RE. In contrast with up-front RE, stakeholders security concerns in just-in-time RE are rarely recognized when publishing requirements. In fact, most security concerns are expressed in requirements refinements (i.e., comments to requirements) or other software artifacts like code review. Therefore, existing security management frameworks that are designed for up-front RE are not suitable for OSS projects. This research proposes a new security management framework for OSS projects. The first part of our security management framework is a security requirements identification approach that based on machine learning method linear classifier. Six factors are used to build the binary classifier, and security requirements in three OSS projects are used to evaluate the performance of our new approach. After getting security requirements, a term-based relevance feedback (RF) approach is applied to find trace links between security requirements and their implementations (i.e., security requirements to source code trace links). Two datasets are used to conduct the experiment. The preliminary results indicate that, compared to standard RF and advanced RF, our term-based RF approach achieves higher accuracy while tracing security requirements. Finally, in order to thoroughly test OSS projects, we detect test paths by identifying requirements dependencies. The experiment results show that our semi-auto requirements dependency detection approach could help us find previously unknown vulnerabilities. The knowledge is then used for testing generation. Test cases generated in this step are applied to source code that is retrieved by term-based RF tracing approach to evaluate and improve OSS projects.
security. The success of this research is of significance in three areas. First, it provides tool support for stakeholders who are not security experts in OSS projects to make them aware of security requirements at the early stage. Second, it can increase the dependability of automated security requirements tracing tools as well as save manual effort. Last, it provides knowledge of requirements dependency that can be used not only in test paths generation but also in other tasks like security requirements change and reuse.
Acknowledgements

First and foremost, I would like to thank my advisor Dr. Niu. It has been my honor to be his student. The most important thing I have learned from him is “treat others how you want to be treated”. In fact, this is what he always does. He is always supportive of every goal we want to achieve. He is always passionate about everything we decide to do. He is always generous to provide guidance not only for my research but also about how to become a better person. I am truly grateful to him for all the help he provided along the way.

I would like to thank all professors in my committee for their support and guidance. I will never forget each time I met Dr. Hayes in Beijing, Slade, Lisbon, Banff, and Cincinnati. She helped me get involved in conference discussions, attended my paper presentations, and provided valuable feedback to my research. I will never forget each meeting I had with Dr. Dai when I worked as a teaching assistant in her Computer Networking course. She was patient on every question I had and provided useful suggestions and solutions. I will never forget approval and encouragement from Dr. Bhatnagar. His encouragement makes me confident in my research and other tasks. I will also never forget each conversation with Dr. Wang. He provided helpful suggestions on my work as well as my future career. I really appreciate all professors’ time and effort spent on my thesis in improving the quality of my research.

I would like to thank all members of our Software Engineering Research Lab. Friendships, collaborations, and encouragement from them supported me each time I encountered problems and difficulties. I will never forget the fun time with them. I am especially grateful for the help from Xiaoyu, Rue, Mounifah, and Xuanyi. I also want to thank my best friends Suyuan, Jiayang, Xingyu, Xiaobang, Yunke,
and Shisheng. Their friendships and supports are everywhere during my time in Cincinnati.

Lastly, I would like to thank my family for all their love. I want to express my special thanks to my parents. My Mom and Dad not only raise me up but also give me a big dream that makes me who I am now.

Chinese people say “a drop of water in need shall be returned with a spring indeed.” I will remember each help I received forever. Thank you all!
Contents

List of Figures ................................................................. iv
List of Tables ............................................................... vi

1 Introduction ................................................................. 2
  1.1 Security Requirements Detection ................................. 3
  1.2 Security Requirements Tracing .................................... 4
  1.3 Security Requirements Dependency Detection for Testing .... 6
  1.4 Thesis Organization and Contribution .......................... 8

2 Background and Related Work ........................................ 10
  2.1 Security Requirements Detection ................................. 10
  2.2 Security Requirements Tracing .................................... 13
  2.3 Vulnerability Detection in Web Applications .................. 17

3 Security Requirements Detection ..................................... 20
  3.1 Datasets and Metrics Selection ................................. 20
  3.2 Linear Classifier Based Approach .............................. 28
  3.3 Measurement and Results Analysis .............................. 32
  3.4 Summary ............................................................... 36
4 Automated Requirements Traceability 37
   4.1 Term-Based RF for Automated Tracing .......................... 37
   4.2 Experimental Evaluation ........................................ 41
   4.3 Discussion ..................................................... 53
   4.4 Summary ......................................................... 55

5 Test Paths Detection 57
   5.1 Dependency Detection Approach ................................. 57
   5.2 Experimental Evaluation ....................................... 71
   5.3 Vulnerability Testing and Validation ........................... 73
   5.4 Discussion ..................................................... 78
   5.5 Summary ......................................................... 82

6 Conclusions and Future Work 84
List of Figures

3.1 Security related URL examples ........................................... 28
3.2 Linear classifier based approach architecture: NFR-C stands for
NFR classifier ................................................................. 29
3.3 Linear classifier based security requirements identification approach
performance under different threshold ............................ 34
4.1 Tracing dependability requirements via term-based RF. ............ 38
4.2 iTrust dependability taxonomy: TS (transmission security), AC
(access control), INT (integrity), PA (person or entity authentication),
AUD (audit controls), IC (integrity control), AL (automatic logoff),
UUI (unique user identification), EAP (emergency access procedure),
EAD (encryption and decryption), MP (mechanism to authenticate
electronic protected health information), ER (emergency responder),
HCP (health care personnel), UAP (unlicensed authorized personnel),
PHA (public health agent), LT (lab technician), LHCP (licensed
HCP), DLHCP (designated LHCP), ULHCP (unlicensed LHCP). . 45
4.3 WDS dependability taxonomy: WIA (workforce investment act). . 46
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>Recall-precision analysis of iTrust: dependability requirements tracing (top); non-dependability requirements tracing (bottom).</td>
</tr>
<tr>
<td>4.5</td>
<td>Tracing effectiveness on WDS. White boxes: dependability and security requirements; gray boxes: non-dependability and non-security requirements.</td>
</tr>
<tr>
<td>4.6</td>
<td>DS specificity of WDS dependability requirements.</td>
</tr>
<tr>
<td>5.1</td>
<td>Dependency detection: HSR: high-level security requirement; LSRs: low-level security requirements.</td>
</tr>
<tr>
<td>5.2</td>
<td>HSRs’ dependencies in FIPS 200.</td>
</tr>
<tr>
<td>5.3</td>
<td>Feedforward neural network model for low-level security requirements (LSRs) dependency detection.</td>
</tr>
<tr>
<td>5.4</td>
<td>An XSS vulnerability example in Scholar@UC: (a) submit work; (b) export citation; and (c) unexpected pop-up window created by XSS code in work’s title.</td>
</tr>
<tr>
<td>5.5</td>
<td>Evidence of the vulnerability in medical records release requirement.</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Data collection for the subject projects ........................................ 21
3.2 Security indicator terms .............................................................. 22
3.3 Effectiveness of indicator terms: AS: answer set .......................... 23
3.4 Statistic analysis for complexity metrics: * indicates significant difference ................................................................. 25
3.5 Comparison between NFR-C, RM, and LC: $R$ refers to recall, $P$ refers to precision, bold values are highest values in their columns . 35

4.1 Traceability-related characteristics of dependability requirements (Dep.) and non-dependability requirements (Non-D.) in the experimental datasets ................................................................. 41
4.2 iTrust dependability requirements ...................................................... 43
4.3 WDS dependability requirements ....................................................... 43
4.4 RF performances in tracing iTrust’s dependability requirements (D) and non-dependability requirements (Non-D): $R$: recall; $P$: precision; $SR$: standard Rocchio; $AR$: adaptive Rocchio; $TB$: term-based RF . 48
5.1 Projects used to evaluate requirements dependency detection approach: LSRs: low-level security requirements; PID: project id; D: dependency; A: assessment (cost) ................................. 70

5.2 Results for requirements dependency detection: PID: project id in Table 5.1; LS: lexical similarity; ↓: reduction; R: recall; P: precision; *: student’s t-test p-value less than 0.05; V: vulnerability .......................... 70
Chapter 1

Introduction

Demand for highly secure software continues to escalate as reliance on software increases and software projects become increasingly popular in our daily life. Unfortunately, even iconic companies like Google and Facebook miss important security vulnerabilities in their software products before releasing them. More seriously, one undetected vulnerability can lead to a security accident that has an impact on tens of thousands of people. For instance, Facebook’s data breach reported in September 2018 caused nearly 50 million Facebook users’ personal information [1] to be exposed by attackers.

One of the resources that cause security vulnerabilities in commercial software products is the open-source software (OSS) used in them. For example, Libxml2\(^1\) is a widely used open-source XML file parser. A vulnerability reported in 2016 in Libxml2 negatively affects 10 companies including Microsoft and Apple and their 29 software products that use Libxml2 [2]. In 2018, there are 16,515 vulnerabilities are reported to Common Vulnerabilities and Exposures, an open-source vulnerability

\(^1\)http://www.xmlsoft.org
repository. Among them, over 3,000 are reported in open-source software projects. Suppose that one open-source software vulnerability could negatively impact 30 other software products, over 90,000 software products that are used in our daily life faces serious security issues.

Having an efficient way to manage security issues in OSS projects could help software developers detect as many vulnerabilities as possible before releasing their products. However, our observation on 345 OSS projects that are funded by Apache\(^2\) and JBoss\(^3\) suggests that few developers in OSS projects are well trained to manage security requirements. Therefore, in this thesis, we proposed a framework to help developers systematically manage security requirements. Our framework contains three main components, that is detecting, tracing, and testing security requirements. We will introduce the importance of these three components in terms of detecting vulnerabilities next.

### 1.1 Security Requirements Detection

Security refers to a class of non-functional requirements (NFRs) related to system confidentiality, integrity, and availability [3]. It is a fundamental requirement and research direction for the Internet of Things (IoT) [4, 5], a key component of Industry 4.0 (I4.0) [6, 7].

Experience indicates that thinking about security early in the software life cycle can help address security problems, such as reducing the defect density ratio (that is, number of bugs per thousand lines of code) [8]. Early detection of security requirements enables engineers to incorporate security into the initial

---

\(^2\)https://www.apache.org

\(^3\)https://www.jboss.org
architectural design [9]. However, differentiating security requirements from non-
security requirements can be nebulous even to the humans, much less the automated
detection methods. Additionally, the trend of agile software engineering and Just-
In-Time requirements engineering (JIT RE) [10, 11] increases the complexity and
difficulty of this problem.

A considerable number of studies have been done on detecting security require-
ments [12, 13]. However, they are labor intensive. Cleland-Huang et al. [9] proposed
NFR-classifier, an automated approach based on information retrieval methods for
detecting and classifying NFRs. Mahmoud and Williams [14] proposed another
automated approach that exploits the textual semantics of software functional
requirements (FRs) to infer potential quality constraints enforced in the system.
Their research demonstrated that those methods can achieve high accuracy in
traditional or *up-front* RE. As a precursor to our work, we applied these methods
to three open source software (OSS) projects, and the results show that none
of them achieve similar performance as in *up-front* RE projects. Furthermore,
the goal of those research is providing general methods for all NFRs, not specific
for security requirements. Therefore, security specific semantic information, such
as CVE (Common Vulnerabilities and Exposures)\textsuperscript{4} id are not included in their
approaches.

### 1.2 Security Requirements Tracing

Security is a critical quality attribute of many industrial systems such as
medical applications [15], controller area networks [16], and power platforms [17].

\textsuperscript{4}https://cve.mitre.org
Dependability refers to the system’s ability to deliver service that can justifiably be trusted, and for different systems, dependability concerns range from safety through reliability to availability [3]. Because hardware redesigning is expensive, software plays an increasingly important role in continuously securing dependability in industrial informatics.

In software and systems engineering, *traceability* refers to the potential to connect interrelated artifacts throughout the life cycle of a system. To be able to trace various dependability concerns, such concerns must exist in the first place. Identifying those concerns therefore receives much attention in dependability requirements engineering [18].

In contrast, little effort has been devoted to developing effective tracing methods that facilitate the verification and validation (V&V) of dependability requirements. In other words, even if all the important requirements are identified, system dependability cannot be assured without the proper implementation of those requirements. Verification, in this context, is to ensure that the implementation conforms to the dependability specifications, and validation is to ensure that the system meets the stakeholders’ expectations on dependability.

To reduce the manual effort, researchers have exploited information retrieval (IR) to automate the requirements tracing for V&V [19, 20, 21]. Each requirement’s textual description serves as a query, against that the source code elements are ranked in the order of estimated relevance. However, IR-based tracing methods return a large portion of false positives, partly due to the many yet superfluous terms appeared in the query requirement where a term is a content identifier typically encapsulated in a word [22]. The false positives not only negatively affect the effectiveness of the tool support, but also decrease the trust from the engineers.
To increase the automated tracing tool’s believability, Hayes et al. [19] suggested to solicit analyst feedback and incorporate it into the regeneration of candidate links such that the final traces could become as accurate as possible. The mechanism is known as relevance feedback (RF) and has received considerable attention in the IR literature [23].

1.3 Security Requirements Dependency Detection for Testing

In the past decade, cross-site scripting (XSS) vulnerability constituted the largest class of newly reported vulnerabilities making it the most prevalent type of attack for web applications today [24, 25]. Failing to identify a single XSS vulnerability can lead to catastrophic security issues. For instance, in 2014, an XSS vulnerability reported in eBay caused a data breach that compromised nearly 145 million customers’ usernames and passwords, and only in Britain, eBay could be fined £500,000 because of this data breach [26].

Static analysis [27, 28, 29] and security testing [30, 31] are two widely applied approaches to identify vulnerabilities. Previous research pointed out that static analysis not only has high false positive rates but also misses true vulnerabilities [30]. In contrast, security testing is highly precise [32]. Several testing approaches such as dynamic taint analysis [30] and penetration testing [33] are applied to validate individual security requirements. However, even if they are successfully satisfied in isolation, security requirements may be violated when they interact with one another, thus leading to undetected security vulnerabilities.

Requirements engineers have long recognized that requirements dependency
analysis is crucial to developing high-quality software systems. According to Robinson et al. [34], up to 70% of total software errors are caused by requirements dependencies, making the requirements dependency error a significant development problem. Not only is requirements dependency important for security testing, it also affects other tasks in software engineering, such as requirements change impact analysis [35], requirements prioritization [36], and source code refactoring [37].

Despite the importance of requirements dependency, the state-of-the-art suffers from the problem of an extremely high cost of interaction assessment. Carlshamre et al. [38] studied interdependencies within five distinct sets of requirements from industrial projects. They pointed out that pairwise assessment of only 40 requirements would take in the vicinity of 13 hours. However, in modern software projects, the size of requirements ranges from hundreds to thousands [39]. Therefore, it is necessary to address the issue of how the assessment effort could be reduced.

Based on their investigations, Carlshamre et al. [38] suggested a set of semi-automated approaches to reduce human effort. The results showed that their approaches could cover roughly 3/4 of the dependencies with 1/3 of the assessment cost of pairwise analysis. It is a reasonable trade-off for requirements dependency analysis in many industrial settings. However, security testing cannot afford to miss any dependencies, because, like the XSS vulnerability in eBay [26], a single vulnerability from the requirements dependency can affect millions of users. We need a new approach that can reduce the assessment cost without losing important requirements dependencies.

According to Carlshamre et al. [38], theoretically, there can be 190 pair-wise dependencies among 20 requirements. However, the number of identified dependencies varied between 19 and 42 in five projects. In other words, for a requirement, most
other requirements are not related to it. The assessment cost can be significantly reduced if we can exclude those obviously irrelevant requirements before we examine the dependencies for a requirement. In fact, security regulations such as Health Insurance Portability and Accountability Act (HIPAA)\textsuperscript{5} and Family Educational Rights and Privacy Act (FERPA)\textsuperscript{6} provide such guidance.

According to Cleland-Huang \textit{et al.} [40], to make claims that web applications comply with security regulations, developer teams have to make sure that low-level security requirements (LSRs) of web applications are Last accessed Dec.\textsuperscript{consist}ent with high-level security requirements (HSRs) defined in security regulations. It means that dependencies between LSRs have to be consistent with dependencies between HSRs. Therefore, if two HSRs are related to each other, there is a high possibility that LSRs that implement these two HSRs are related. Otherwise, we assume that there is no need to assess LSRs associated with these two HSRs.

\section{1.4 Thesis Organization and Contribution}

The goal of our work is to design a security requirements management framework for OSS projects with the hope that it can help find more security vulnerabilities in OSS products before releasing them. Our framework contains three main components: 1) detecting security requirements from OSS projects issues; 2) tracing security requirements to source code; and 3) finding security test paths by detecting requirements dependencies. We introduce the background and related work of each component in Chapter 2.

\textsuperscript{5}HIPAA requires health agencies in the United States to use technical safeguards to protect patient medical information. See \url{https://www.hhs.gov/hipaa/index.html}

\textsuperscript{6}FERPA forces all educational agencies like colleges in the United States to protect student education records. See \url{https://www2.ed.gov/policy/gen/guid/fpco/ferpa/index.html}
In Chapter 3, motivated by the above observations from OSS projects, we propose a novel and efficient approach for identifying security requirements in OSS development. In this chapter, we have the following contribution: 1) building models with metrics related to requirements complexity and eternal resources; and 2) finding an optimal way to integrate all models with NFR-classifier. Our results show that the enhanced approach can achieve average 92.31% recall and 62.94% precision in three OSS projects.

In Chapter 4, the goal is to explore ways that best leverage RF to trace dependability requirements. To that end, this paper makes four main contributions: developing a novel term-based RF algorithm to augment IR-based tracing, devising domain-specific dependability taxonomies to inform V&V, define a new metric to measure tracing method’s specificity, and performing experimental evaluations of our proposed method.

In Chapter 5, we propose a semi-automated requirements dependency identification approach that uses dependencies between HSRs to reduce LSRs’ dependency assessment cost. Other contributions include conducting an experiment on an OSS project Scholar@UC to evaluate whether our approach can detect validated vulnerabilities and suggesting best practices and improvements for vulnerability detection.

The outcome and contribution of our work lie in two aspects: 1) we propose automated or semi-automated approaches to assist developers on their security-related tasks including detecting, tracing, and testing security requirements; and 2) we apply our framework on an OSS projects Scholar@UC. Original developer from Scholar@UC developer confirms that four vulnerabilities found by our framework are valid. This demonstrates the effectiveness of our framework.
Chapter 2

Background and Related Work

Our security requirements management framework asks developers to complete three main tasks: security requirements identification, verification, and testing. Automated or semi-automated tools are designed to help developers accomplish those tasks. Detailed information about those tools will be provided in Section 3, 4, and 5. To give readers background knowledge, in this section, we summarize related work of each tool design.

2.1 Security Requirements Detection

Existing researches applied different machine learning algorithms to identify and classify NFRs. In this section, we introduce previous researches based on different machine learning algorithms and their performance on security requirements identification.

Cleland-Huang et al. [9] described NFR classifier, an automated method based on the assumption that certain keywords can be used to distinguish different types of NFRs. A set of pre-defined requirements was used to derive different sets of
weighted keywords called indicator terms for different NFRs. Indicator terms were then used as queries to retrieve NFRs from various software artifacts. An industrial case study was conducted to evaluate the proposed approach. The result showed that NFR classifier can recall 82.8% security requirements. At the same time, the result also showed a very high ratio of false positive (7.1% precision). The proposed approach benefit from the textual features of requirements, and ignore other requirements features, e.g., stakeholders information and lexical information. We conjecture that contextual and lexical features are complementary with the textual feature on indicating security requirements and thus classifiers build with contextual information may be possible to integrate with NFR classifier and improve the performance of the proposed approach.

Unlike NFR classifier that derives indicator terms from a pre-labeled set of requirements, Mahmoud and Williams [14] proposed a method that discovers and assesses the query terms by using un-labeled datasets. The proposed approach first grouped terms of requirements into different clusters based on their semantic similarity. Then average pairwise similarity between clusters and NFR categories (e.g., accessibility, interoperability, and security) were calculated. A cluster was only assigned to the NFR category with highest average pairwise similarity. Requirements then were automatically classified into different NFR categories based on their semantic similarity to the clusters of terms representing individual NFRs. This approach adopts crisp clustering to ensure clearcut results, i.e. one term can belong to one NFR only. However, the same term can indicate different things. For instance, “denial-of-service” is related to both security and availability. Thus, crisp clustering may cause the in-completed set of representative terms. Furthermore, it may harm the performance of NFR identification.
Kurtanović and Maalej [41] applied Support Vector Machine (SVM) with 11 different lexical features, such as text length, fractions of nouns and verbs, and sentence syntax tree height to build two classifiers. The binary classifier was used to distinguish functional requirements (FRs) and NFRs. The multi-class NFRs classifier is worked on the result of the binary classifier to identify usability, security, operational and performance NFRs. The proposed approach was applied to two datasets, and results showed that this method can find 90.1% security requirements and filter out most non-security requirements (precision 90.0%). However, unlike datasets used in this research, requirements specifications in OSS projects tend to be organized by functionality, with NFR scattered widely across multiple documents [9]. Therefore, there is no clear boundary between FRs and NFRs. Moreover, the requirements stored in issue tracking systems are unstructured and seldom obey grammar and punctuation rules [42]. Thus, this approach is not suitable for OSS projects.

Munaiah et al. [43] proposed a un-supervised approach for detecting security requirements. They trained their One-Class Support Vector Machine with the Common Weakness Enumeration (CWE)\(^1\), a formal list of software weakness types intended to serve as the common language for describing software security weaknesses in architecture, design, or code. The assumption authors conjectured is that the language used to describe security requirements and that used to describe weaknesses overlap thus it could be a good candidate to train the classifier. Like Gibiec et al. [44] pointed out, term mismatch problem is a common occurrence in software artifacts. Additionally, specific terms contained in requirements that indicate concerns like security and dependability are domain/application dependent.

\(^1\)https://cwe.mitre.org
Therefore, the performance of this approach is lower than other approaches we discussed in the earlier paragraphs of this section (i.e., 70.48% recall and precision level is 67.35%).

To the best of our understanding, there is no approach using features related to requirements complexity and external resources to identify security requirements. However, those features are widely applied in other research domains. Such as, software quality prediction [45], software complexity evaluation [46], and vulnerability detection [28]. Intuitively, these features could be used as metrics to train a binary security requirements classifier.

2.2 Security Requirements Tracing

Requirements traceability is widely recognized as an effective means to verify whether requirements are correctly implemented and tested. In this section, we first introduce security requirements and then discuss the-state-of-the-art approaches of automated tracing.

2.2.1 Engineering Security and Dependability

As industrial information and control systems become pervasive in our daily lives, we have a natural dependence on the proper construction and operation of these systems. Dependable systems are those with the capability of avoiding severe and/or frequent service failures [3]. The past decade has seen considerable advances in technology for building dependable industrial applications [15, 16, 17].

An emerging trend in dependable industrial engineering is the increasing exploitation of the fast evolution pace of software in the entire development life
cycle [47], and preferably at the requirements level [12, 13]. So far, the research focus of security requirements engineering has been on elicitation [18]. While this is important, ensuring the requirements are implemented in the code base is equally important. In fact, certain dependability requirements are readily available, well documented, and therefore must be rigorously satisfied. Examples include the Health Insurance Portability and Accountability Act (HIPAA)\(^2\) that all healthcare-related products in the USA must comply with, as well as the IEC 61508 functional safety standard\(^3\) that critical environments like automotive or energy production applications should follow.

In sum, dependability is a concern that must be taken into consideration starting from the early stages of industrial system development. Dependability requirements engineering has resulted in many approaches to eliciting specific concerns, most notably security and privacy. Less effort is made to ensure the realization of dependability requirements across the development life cycle. Next, we review the literature on automated traceability as a way to achieve requirements V&V.

\subsection*{2.2.2 Requirements Traceability and Relevance Feedback}

The traceability information is instrumental in V&V to ensure that the right processes have been used to build the right system [19]. Tracing based on information retrieval (IR) aims at automatically identifying candidate links between different types of software artifacts by relying on the artifacts’ textual descriptions. Researchers have applied many IR methods in automated tracing. In most cases, a recall of 90\% is achievable at precision levels of 5-30\%. In traceability, recall mea-

\(^2\)http://www.hhs.gov/hipaa/
\(^3\)http://www.iec.ch/functionalsafety/
sures the percentage of true links found by IR algorithms, and precision measures the accuracy of the returned candidate link list [19]:

\[
\text{Recall} = \frac{\sum_{q \in Q} r_q}{\sum_{q \in Q} R_q}, \quad \text{Precision} = \frac{\sum_{q \in Q} r_q}{\sum_{q \in Q} n_q},
\]

(2.1)

where each \( q \in Q \) is a to-be-traced requirement, \( R_q \) is the set of true links of \( q \), and \( n_q \) is the set of candidate links that the IR method returns, out of which \( r_q \) are true. Because achieving high precision and high recall is a balancing act, the weighted harmonic mean of \( F_2 \) is commonly used as a single metric to measure tracing effectiveness [19]:

\[
F_2 = \frac{5 \cdot \text{Precision} \cdot \text{Recall}}{4 \cdot \text{Precision} + \text{Recall}}.
\]

(2.2)

To improve the retrieval effectiveness and hence the tracing tool’s believability, Hayes et al. [19] proposed to integrate relevance feedback (RF) into the tracing process. In traditional IR, the basic idea of RF is to present an initial set of retrieved documents to the user and ask her to judge which documents are relevant to her information needs. The relevance judgments are used to produce a modified version of the query by weighting more on the terms appeared in the relevant documents and less on the terms from the irrelevant ones. The modified query is then used to retrieve a new set of documents [23].
Developed using the vector space model, the standard Rocchio method remains an effective and robust RF mechanism. Specifically, Rocchio’s formula for modifying the query vector \( \overrightarrow{Q_m} \) from the original query vector \( \overrightarrow{Q_o} \) is:

\[
\overrightarrow{Q_m} = (\alpha \cdot \overrightarrow{Q_o}) + (\beta \cdot \frac{1}{|D_r|} \sum_{\vec{d}_j \in D_r} \vec{d}_j) - (\gamma \cdot \frac{1}{|D_i|} \sum_{\vec{d}_k \in D_i} \vec{d}_k)
\]  \( (2.3) \)

where \( D_r \) is the set of relevant documents and \( D_i \) is the set of irrelevant documents. Intuitively, the modified query takes the initial query vector while adding the weighted vectors \( \vec{d}_j \in D_r \) and subtracting the weighted vectors \( \vec{d}_k \in D_i \). Weighting parameters \( \alpha \), \( \beta \), and \( \gamma \) are used to assign different emphases to \( \overrightarrow{Q_o} \), positive feedback, and negative feedback, respectively.

Hayes et al. [19] provided evidence for standard Rocchio’s accuracy improvement in IR-based tracing. Using simulated RF, other researchers were also able to show Rocchio’s positive effect on tracing performances. In particular, the recent work by Panichella et al. [21] presented an adaptive version of RF. Rather than applying the standard Rocchio to every pair of tracing source and target, the adaptive Rocchio algorithm checks certain conditions before applying the RF. The conditions are defined based on the software artifacts’ verbosity and the links already classified.

In sum, IR-based tracing algorithms reduce much manual effort in checking the implementation of requirements in the software development life cycle. When RF is integrated, the tracing effectiveness can be improved. A gap is to perform RF on a subset of the terms used to express specific dependability concerns, rather than adjusting weights for all the terms appeared in the query requirement. Next, we
present a term-based RF algorithm for dependability requirements tracing.

2.3 Vulnerability Detection in Web Applications

Development teams usually apply multiple approaches to help prevent vulnerabilities. These approaches can be classified into two categories: static analysis and dynamic analysis. In this section, we discuss approaches in both categories that can be applied to detect vulnerabilities in web applications.

**Static analysis** looks into the source code of the system under test (SUT) without actually executing it and reports potential vulnerabilities. It can be done by the manual code review. A couple of studies from academic and industry [50, 51] indicate that code review can find functional errors that can be exploited by attackers as well as issues that affect the system’s maintainability. Code review requires expertise in the application architecture, the implementation techniques, as well as security. However, such expertise is not always available. In addition, code review asks experts to read the source code “line-by-line” [52]. Thus, instead of applying it to the system level, it is more reasonable to apply code review on subsets of the source code (e.g., the modified part).

Automated approaches are proposed to improve the efficiency of static analysis. In these approaches, different software metrics related to source code complexity, like lines of code, coupling, and cohesion, are used to predict software vulnerabilities [28, 53]. Most automated approaches predict all vulnerabilities in general. With appropriate adaptions or configurations, they can also be applied to predict specific vulnerabilities that are unique to the SUT (e.g., XSS vulnerability in web applications). However, the previous study [53] showed that these approaches
suffer from two limitations: low precision rate and low effectiveness on cross-project vulnerability prediction. The first limitation increases the human effort spent on results evaluation, while the second one leads to that a small team of security experts is required to build new prediction model for each new project which is neither practical nor realistic.

**Dynamic analysis** is complementary to static analysis. Dynamic analysis approaches are performed by executing SUT on the real or virtual running environment. These include fuzzing, taint analysis, and vulnerability scanning. Fuzzing or fuzz testing is based on the idea of feeding random data to the SUT “until it crashes” [54]. However, vulnerabilities in web applications like XSS do not necessarily crash systems, making fuzzing less suitable for these types of vulnerabilities.

A more suitable approach for vulnerability detection in web applications is taint analysis [30]. Untrusted user data is labeled as “tainted” at runtime, which is cleared only if the data passes a dedicated sanitization function. If the data that still carries the taint information reaches a security sensitive sink (e.g., a webpage that displays the tainted data), the system is considered as vulnerable. Taint analysis requires the source code that is not always available in security testing. In addition, extra engineering effort (e.g., adding new database columns for tracking user input [30]) is required to implement taint analysis.

Another approach which requires less engineering effort is vulnerability scanning. Automated scanners like SecuBat [55] and ZAP [56] are widely used to detect vulnerabilities in web applications. These scanners query the system’s interface with a set of predefined attack payloads (e.g., attacks in XSS Filter Evasion Cheat Sheet [57]) and analyze immediate responses of the system for indicators of if the attack was successful. However, hints for the successful attacks are not always in
the immediate responses. For instance, in the eBay case, it is hard to tell whether
attackers successfully attack the system when they try to save the malicious XSS
code into the system. The judgment can be provided until the malicious links are
displayed to victims. This limitation can be addressed by systematic requirements
dependency detection.
Chapter 3

Security Requirements Detection

Our observation from 345 OSS projects funded by Apache and JBoss suggests that developers of OSS projects rely on security experts to help distinguish security requirements from others. However, security experts are not always available in those OSS projects. Therefore, having an automatic approach like binary classifier to help developers identify security requirements shall be the first step of our security requirements management framework. However, there are several challenges in building classifier for detecting security requirements.

3.1 Datasets and Metrics Selection

The first challenge in building security requirements identification classifier is to find candidate metrics. We manually analyzed three web-based OSS projects. We found that two types of metrics are suitable to distinguish security requirements from non-security requirements. In this section, we first introduce the three datasets we used to evaluate our approach and then discuss metrics selection.
3.1.1 Datasets and Answer Sets

In our study, we analyzed the requirements of three OSS projects\(^1\): Apache Axis2/Java (Axis2)\(^2\), Drools\(^3\), and GeoServer\(^4\). We selected these projects as the subject systems due to three reasons. First, all of them are successful and long-lived projects. Second, all resources including requirements and source code are available. At last, all three projects are web-based systems and security is one of the core aspects of these projects, so identifying and realizing security requirements are important tasks for developers of these three projects.

These projects come from different application domains, and all of them are written in Java. Axis2 is a web services engine funded by Apache Software Foundation since August 2004. The newest version of Axis2 (1.7.4) was released in December 2016. Drools is a business rule management system developed by Red Hat. The latest stable release of Drools is 6.5.0, which was published in December 2016. GeoServer is a geographic system that allows users to edit and share geospatial information. The current stable version 2.11.0 was published in March 2017. Table 3.1 shows some basic information about these three projects.

\(^1\)We share our collected data and analysis results in the online document [58] for validation and replication purpose.
\(^2\)http://axis.apache.org/axis2/java/core/
\(^3\)https://www.drools.org
\(^4\)http://geoserver.org

Table 3.1: Data collection for the subject projects

<table>
<thead>
<tr>
<th>Project</th>
<th>Domain</th>
<th># of issues</th>
<th># of security issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axis2</td>
<td>Web server</td>
<td>5751</td>
<td>109</td>
</tr>
<tr>
<td>Drools</td>
<td>Business</td>
<td>326</td>
<td>33</td>
</tr>
<tr>
<td>GeoServer</td>
<td>Geo information</td>
<td>8172</td>
<td>651</td>
</tr>
</tbody>
</table>
Table 3.2: Security indicator terms

<table>
<thead>
<tr>
<th>Project</th>
<th>Project dependent indicator terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axis2</td>
<td>Apache rampart, credentials, MEP (message exchange pattern), stack overflow exception, socket, SOAP (simple object access protocol), threadsafe, WS-Security, WS-Addressing</td>
</tr>
<tr>
<td>Drools</td>
<td>ACL (access control list), security-module-admin.properties, security-policy.properties</td>
</tr>
<tr>
<td>GeoServer</td>
<td>GeoServerSecurityFilterChain, GeoServerSecurityFilterChainProxy, KeyStoreProvider, OAuth2, ResourceAccessManager, SQL</td>
</tr>
</tbody>
</table>

We manually created the answer sets\(^5\) via a two-step analysis. In the first step, two pre-defined sets of security indicator terms were used to retrieve security requirements candidates in each project. One is called the “general indicator terms” \(^5\) that is extracted from the previous research by Avizienis \textit{et al.} \cite{Avizienis} and Cleland-Huang \textit{et al.} \cite{Cleland-Huang}. For another one called “project dependent indicator terms”, we manually went through the official websites of three projects, collected all security related artifacts, and then extracted project dependent indicator terms from those security related artifacts. For instance, in Axis2, developers describe their transport framework as “We have a clean and simple abstraction for integrating and using transports (i.e., senders and listeners for SOAP over various protocols such as SMTP, FTP, message-oriented middleware, etc), and the core of the engine is completely transport-independent.” According to SOAP (simple object access protocol) definition from W3C\(^6\), it is security related. Therefore, we added it into Axis2’s project dependent indicator terms set. Table 3.2 shows “project dependent

\(^5\)Our answer sets can also be found in the online document \cite{Avizienis}.

\(^6\)\url{https://www.w3.org/TR/soap/}
Table 3.3: Effectiveness of indicator terms: AS: answer set

<table>
<thead>
<tr>
<th>Project</th>
<th>Effectiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AS</td>
</tr>
<tr>
<td>Axis2</td>
<td>13</td>
</tr>
<tr>
<td>Drools</td>
<td>4</td>
</tr>
<tr>
<td>GeoServer</td>
<td>54</td>
</tr>
</tbody>
</table>

indicator terms” for three projects.

To evaluate the effectiveness of our indicator terms, we design a pilot study. The research question in this pilot study is “can indicator terms displayed in Table 3.2 retrieve all security requirements (i.e., achieve 100% recall)?” First, from each dataset, we randomly choose 10% requirements and ask an analyst who has one year research experience on both these three projects and security to manually go through all of selected requirements and classify them into security or non-security requirements. After this step A keywords searching is then applied to the selected requirements. That is, if a requirement contains at least one indicator term, we mark it as security requirements. The searching results are showed in Table 3.3. Indicator terms can find 100%, 100%, and 96.30% security requirements. Two GeoServer security requirements not retrieved by the keywords searching shared the word “SQL”. Therefore we add it to Table 3.2 as a new indicator for GeoServer project (red part). Results of pilot study demonstrated that keywords searching can retrieve all security requirements, we therefore apply it to help us to reduce manual effort.

In the second step, two analysts classified all candidates as security or non-security requirements independently. If two analysts have the same decision on one issue, we mark it as security/non-security requirement directly. Otherwise,
two analysts have a discussion and make the final decision together. According to Cohen’s Kappa analysis, two analysts achieved almost perfect agreement [59] (i.e. 0.89, 0.73, and 0.97 on Axis2, Drools, and GeoServer, respectively).

### 3.1.2 Metrics Selection

In this section, we classify our metrics into two categories: complexity and external resources. For each metric, we not only provide the hypotheses for the discriminative power and the predictability, but also provide the rationale for why these metrics can work for security requirements identification.

**Complexity**: Security experts claim that complexity is the enemy of security [60, 61]. Complexity in requirements can lead to complex code that contains subtle vulnerabilities that are difficult to test and diagnose, providing more chances for attackers to exploit [Anonymous, 2016]. From this reason, we set up the following hypothesis on complexity in requirements.

\[ H_{CommentsComplexity} \]: Security requirements in OSS projects involve more discussions than non-security requirements.

A requirement containing more comments has a higher chance of having interaction with other issues, leading to more code having input from the external source to modules implemented by other developers without properly understanding the security concerns or assumptions of the modules [28]. We first tested four metrics, including number of comments/requirement (\# C), number of nouns in comments/requirement (\# N), number of verbs in comments/requirement (\# V), and average length of comments/requirements (ALC), by comparing their distribution difference between security and non-security requirements. For each dataset, we chose subsets of security and non-security requirements via simple random sampling.
Table 3.4: Statistic analysis for complexity metrics: * indicates significant difference

<table>
<thead>
<tr>
<th>Project</th>
<th>Metric</th>
<th>Sec. Req.s (mean ± s.d.)</th>
<th>Non-Sec. Req.s (mean ± s.d.)</th>
<th>p value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># C</td>
<td>4.24 ± 4.67</td>
<td>3.97 ± 4.93</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td># N</td>
<td>11.23 ± 8.23</td>
<td>12.65 ± 9.72</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td># V</td>
<td>7.52 ± 6.26</td>
<td>6.71 ± 7.64</td>
<td>0.11</td>
</tr>
<tr>
<td></td>
<td>ALC</td>
<td>33.16 ± 21.37</td>
<td>15.26 ± 22.69</td>
<td>0.02 *</td>
</tr>
<tr>
<td></td>
<td># S</td>
<td>7.26 ± 3.21</td>
<td>4.31 ± 4.59</td>
<td>0.04 *</td>
</tr>
<tr>
<td>Axis2</td>
<td># C</td>
<td>6.23 ±3.76</td>
<td>5.44 ± 5.24</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td># N</td>
<td>10.32 ± 11.73</td>
<td>9.72 ± 9.27</td>
<td>0.09</td>
</tr>
<tr>
<td></td>
<td># V</td>
<td>7.43 ± 7.53</td>
<td>6.99 ± 6.17</td>
<td>0.07</td>
</tr>
<tr>
<td></td>
<td>ALC</td>
<td>29.73 ± 32.61</td>
<td>17.29 ± 29.96</td>
<td>0.03 *</td>
</tr>
<tr>
<td></td>
<td># S</td>
<td>6.33 ± 5.31</td>
<td>4.11 ± 4.23</td>
<td>0.04 *</td>
</tr>
<tr>
<td>Drools</td>
<td># C</td>
<td>5.49 ± 7.62</td>
<td>4.24 ± 6.52</td>
<td>0.11</td>
</tr>
<tr>
<td></td>
<td># N</td>
<td>11.33 ± 7.54</td>
<td>10.58 ± 3.72</td>
<td>0.07</td>
</tr>
<tr>
<td></td>
<td># V</td>
<td>5.43 ± 5.29</td>
<td>6.34 ± 4.23</td>
<td>0.08</td>
</tr>
<tr>
<td></td>
<td>ALC</td>
<td>24.52 ± 28.31</td>
<td>16.33 ± 23.42</td>
<td>0.03 *</td>
</tr>
<tr>
<td></td>
<td># S</td>
<td>8.21 ± 4.27</td>
<td>3.62 ± 3.69</td>
<td>0.02 *</td>
</tr>
</tbody>
</table>

(SRS) [62]. Then we applied student t-test to measure statistic difference between two sets for each metrics. Table 3.4 shows the testing results. Only ALC has the significant difference between security and non-security requirements. Therefore, only ALC is used to measure comments complexity.

\[ H_{Stakeholders\text{Complexity}}: \text{Security requirements in OSS projects involve more stakeholders than non-security requirements.} \]

We used the number of stakeholders/requirement (# S) to measure the stakeholders’ complexity. Issue tracking system used by three projects (i.e., JIRA) predefined several stakeholders: creators, assignees, comment authors, watchers, and voters. Creators, assignees, and comment authors provide their opinions and
refinement of requirements via commenting on requirements. Different people have different opinion and preferences. Therefore, the more stakeholders provide their opinion, the more complicated the requirements are. In addition, human errors are the most frequent type of errors in the requirements [63]. The errors also indicate the high possibility of vulnerability. For watchers and voters, JIRA only shows how many of them, not who are they. That means they may or may not overlap with creators, assignees, and comment authors. If they do not belong to previous three types of stakeholders, that means they do not provide any opinions or refinements. Therefore, we ignore them when we calculate stakeholders complexity. Shapiro-Wilk test accepts distribution normality for size of stakeholders in both security and non-security requirements. Thus, the result of student $t$-test that shows significant difference between security and non-security requirements is valid.

**External Resources**: Stakeholders tend to provide external resources, e.g., URLs to documents on other websites, to provide the rationale for their refinement and explain their solutions. After analyzed three OSS projects, we found that three external resources have potential to help to distinguish security and non-security requirements.

From our observation, stakeholders add URLs to the requirements while they want to use information provided by the URLs. For instance, solutions for the current requirements, project dependabilities, and detailed description of requirements. However, not every URL can indicate the security issue. After analyzed three OSS project, we found that only two types of URLs indicates the security issues. In the Type I example, the URL points to the document that is mainly about security-related issues (Figure 3.1(a)). In this case, the last part of the URL, which is the name of the security document (e.g. “selfadmin.html”), contains security
indicator terms. In the Type II URL (Figure 3.1(b)) points to a document that contains both security and non-security issues. However, it uses “#” to refers to the title of the specific section/chapter in this document. If the title of this specific section/chapter (e.g. “Character_Encodings”) contains security indicator terms, we classify it as a security related-URL. Therefore, we provide following hypothesis:

\[ H_{\text{SecurityURLs}}: \text{Security requirements in OSS projects involve more security related URLs than non-security requirements.} \]

Another external resource provided by stakeholders is the commits from version control system, e.g. CVS, GitHub. These systems record the diff for each commit. Security experts claim that even software use permission systems to attempt to limit the privileges of the softwares. However, existing permission systems are not sufficient to prevent sensitive data from being leaked [64]. Any outside sources can contaminate the sensitive data [65]. Therefore, we only interested in the commits that contain changes related to data exchange. From our observation, there are two types of code change strongly related to data exchange: 1) the change contains “=”; and 2) the change contains “get” or “set” methods. Thus, only the commits that contain data exchange related changes are count as security-related commits. We provide following hypothesis for security-related commits:

\[ H_{\text{SecurityCommits}}: \text{Security requirements in OSS projects involve more security related commits than non-security requirements.} \]

Unlike previous external resources, CVE identifiers (CVE IDs) assigned by CVE Numbering Authorities (CNAs) and used to discuss and share information about a unique software vulnerability is a strong indicator of security requirements. That means if a requirement contains a CVE ID, it is a security-related requirement. Therefore, we provide following hypothesis for CVE IDs:
http://docs.geoserver.org/latest/en/user/rest/api/selfadmin.html

(a) Type I

http://www.ws-i.org/.../BindingProfile.html#Character_Encodings

(b) Type II

Figure 3.1: Security related URL examples

\[ H_{\text{SecurityCommits}}: \text{A requirements in OSS projects involve any CVE IDs shall be consider as security requirements.} \]

We would like to point out that, intuitively, features used to build our classifier are selected based on our investigation of 345 OSS projects. Statistic method student’s \( t \)-test is applied to help feature selection. We also recognize that advanced feature selection approach like Minimum-redundancy-maximum-relevance [66] could be applied to improve the feature selection process in the future.

### 3.2 Linear Classifier Based Approach

In this section, we describe the general architecture of our linear classifier-based approach. Our approach contains five parts: Information Preprocessing Component (IPC), NFR Classifier (NFR-C), Regression Models (RMs), CVE ID Detector (CID), and Linear Classifier (LC). Figure 3.2 presents the framework of our linear classifier based approach.

Input requirements are processed by IPC to extract information of each metric described in Section 3.1.2. Then we feed these information to four regression models
Figure 3.2: Linear classifier based approach architecture: NFR-C stands for NFR classifier (Comment Complexity Regression Model (CRM), Stakeholder Complexity Regression Model (SRM), Security URLs Regression Model (URM), Security Commits Regression Model (CiRM)), NFR-C and CID. For each requirement, each regression model will generate a weight between 0 to 1 that indicates the likelihood of this requirement is security requirement. At the same time, CID will detect whether
this requirement contains a CVE ID. If there is a CVE ID in this requirement, the weight of it is 1, otherwise, the weight of it is 0. For NFR-C, it directly reuses the approach from previous study by Cleland-Huang et al. [9]. Finally, a linear classifier is used to generate a linear discriminant function that is an optimal way to summarize weights from NFR-C, CID, and all RMs. The result of the LC then is used to classifier whether this requirement is a security or non-security requirement. We briefly describe the regression model and the linear classifier in the following subsections.

### 3.2.1 Logistic Regression Model

Lots of classification techniques, such as J48 decision tree, Random forest, Naive Bayes, Bayes Network, and Logistic Regression, have been widely applied to build security identification models [28, 67]. Lessmann et al. [67] investigated 17 classification techniques, and reported that there is no significant difference in performance between those techniques. Same as previous research by Shin et al. [28], we used binary logistic regression to identify security requirements.

Binary logistic regression computes the probability of occurrence of an outcome event from given independent variables by mapping the linear combination of independent variables to the probability of outcome using the log of odds ratio (logit). An open-source cluster-computing framework Apache Spark\(^7\) was used to build our regression models.

\(^7\)https://spark.apache.org
3.2.2 Linear Classifier

A linear classifier is a statistical classification technique. The goal of it is to use an object’s feature values to identify which class it belongs to. It achieves this goal by making the classification decision based on the value of the linear combination of feature values [68].

The discriminant function of linear classification shows in Equation (3.1):

\[ f(\vec{\omega} \cdot \vec{x}) = \Sigma(\omega_i x_i) \]  

(3.1)

where \( \vec{x} \) is a real vector of feature values, and \( \vec{\omega} \) is a real vector of weights for each feature values. The weights vector \( \vec{\omega} \) is learned from a set of labeled training samples by applying linear discriminant analysis (LDA).

Linear classifier is widely used in software engineering field, such as NFR identification [41] and CVS issue status prediction [69]. The major advantages of the linear classifier are its simplicity and time efficiency. Therefore, we thought linear classifier could be a good candidate for our approach. To our best understanding, this research is the first one that applies linear classifier to solve the security identification problem. Therefore, we tried to answer following research questions:

- **RQ1**: Compared to non-function requirements classifier, can the linear combination of feature values achieve better performance?

In addition, the function \( f(\vec{\omega} \cdot \vec{x}) \) maps all values above a certain threshold to one class and all other values to the second class. Therefore, the following question is:

- **RQ2**: What is the optimal threshold for linear classifier when identifying security requirements?
3.3 Measurement and Results Analysis

Previous research proposed different measurements to help evaluate the performance of machine learning approaches. In Section 3.3.1, we discuss three measurements used to evaluate our approach. Then, we report the evaluation results in Section 3.3.2.

3.3.1 Measurement

For evaluating binary identify modes, we used *recall*, *precision*, and *F-measure*.

*Recall* is defined as the ratio of correctly identified security requirements to actual security requirements:

\[
\text{Recall} = \frac{TP}{TP + FN}
\]

(3.2)

where TP refers to true positives and FN refers to false negatives. *Precision* is defined as the ratio of correctly identified security requirements to all detected issues:

\[
\text{Precision} = \frac{TP}{TP + FP}
\]

(3.3)

where FP refers to false positives. *F-measure* is a harmonic mean of *recall* and *precision*.

\[
F_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{\beta^2 \cdot \text{precision} + \text{recall}}
\]

(3.4)

Failing to identify and realize security requirements will lead to a number of serious problems in software systems. So, we are more concerned about *recall* rather
than precision. $F_2$ is a widely used measurement which fits our purpose.

### 3.3.2 Results Analysis

To answer research questions, we applied a specific 10-fold strategy to test our approach. That is, for each project, we randomly divided security requirements into 10 subsets $S = \{s_1, s_2, ..., s_{10}\}$. We used $S - \{s_i\}$ and same size of randomly selected non-security requirements as input to train the classifier. Then the classifier is used to identify security requirements from the rest of requirements (e.g., $s_i$ and non-security requirements that are not used in training phase). All the results reported in this section are the average values of 10 rounds of testing. The purpose of using the specific 10-fold strategy is to having a balanced training set, by doing that we can avoid that the classifier just learns to predict the more frequent class.

To answer RQ2, we tested the performance of our approach via increasing threshold $\theta$ by 0.1 from 0.0 to 1.0. The performance of our approach under different thresholds is shown in Figure 3.3.

The recall curve (Figure 3.3(a)) shows that our approach can achieve 100% recall while the threshold less than 0.3. While the threshold greater than or equal to 0.5, the recall values of three projects decrease as the threshold increases. The significant decrease occurs while threshold increase from 0.9 to 1.0. This phenomenon indicates that lots of false positive occur at the top of the rank list. We conjecture that there are two possible reasons. First, it may because of that some non-security requirements have similar characters with security requirements, we need other strong security indicators to distinguish them from security requirements. Second, it could be also caused by incompleteness of answer sets. We need to improve the quality of the answer sets for the future research.
(a) Recall curve  
(b) Precision curve  
(c) $F_2$ curve

Figure 3.3: Linear classifier based security requirements identification approach performance under different threshold

For the precision curve (Figure 3.3(b)), the precision values of three projects are asymptotically stable at an equilibrium point (i.e., threshold equals to 0.5). Additionally, In the $F_2$ curve (Figure 3.3(c)), our approach achieved highest $F_2$ values while threshold equals to 0.5 in three projects. Based on these observations, we believe that 0.5 is an optimal threshold can be used in our approach to classify security and non-security requirements.

To answer RQ1, we compare the performance of linear classifier with NFR
Table 3.5: Comparison between NFR-C, RM, and LC: R refers to recall, P refers to precision, bold values are highest values in their columns

<table>
<thead>
<tr>
<th>Approach</th>
<th>Axis2</th>
<th></th>
<th></th>
<th>Drools</th>
<th></th>
<th></th>
<th>GeoServer</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$R$</td>
<td>$P$</td>
<td>$F_2$</td>
<td>$R$</td>
<td>$P$</td>
<td>$F_2$</td>
<td>$R$</td>
<td>$P$</td>
<td>$F_2$</td>
</tr>
<tr>
<td>NFR-C</td>
<td>0.62</td>
<td>0.26</td>
<td>0.48</td>
<td>0.55</td>
<td>0.30</td>
<td>0.47</td>
<td>0.55</td>
<td>0.19</td>
<td>0.40</td>
</tr>
<tr>
<td>CRM</td>
<td>0.73</td>
<td>0.21</td>
<td>0.49</td>
<td>0.77</td>
<td>0.23</td>
<td>0.53</td>
<td>0.64</td>
<td>0.15</td>
<td>0.39</td>
</tr>
<tr>
<td>SRM</td>
<td>0.68</td>
<td>0.16</td>
<td>0.41</td>
<td>0.73</td>
<td>0.26</td>
<td>0.54</td>
<td>0.44</td>
<td>0.17</td>
<td>0.33</td>
</tr>
<tr>
<td>URM</td>
<td>0.53</td>
<td>0.20</td>
<td>0.45</td>
<td>0.64</td>
<td>0.37</td>
<td>0.56</td>
<td>0.47</td>
<td>0.23</td>
<td>0.39</td>
</tr>
<tr>
<td>CiRM</td>
<td>0.46</td>
<td>0.69</td>
<td>0.49</td>
<td>0.51</td>
<td>0.65</td>
<td>0.53</td>
<td>0.44</td>
<td>0.53</td>
<td>0.46</td>
</tr>
<tr>
<td>LC</td>
<td>0.88</td>
<td>0.67</td>
<td>0.83</td>
<td>0.95</td>
<td>0.67</td>
<td>0.88</td>
<td>0.91</td>
<td>0.57</td>
<td>0.81</td>
</tr>
<tr>
<td>Pilot</td>
<td>0.84</td>
<td>0.71</td>
<td>0.81</td>
<td>0.93</td>
<td>0.66</td>
<td>0.86</td>
<td>0.87</td>
<td>0.61</td>
<td>0.80</td>
</tr>
</tbody>
</table>

classifier and four regression models. Like previous study by Shin et al. [28], in four regression models, a requirement is classified as a security requirement when the outcome probability is greater than 0.5. We do not include the performance of the CID. The main reason is, even it can achieve 100% precision, however, its recall value is very low, around (3% to 10%). This mainly because a few of security requirements are related to CVE IDs. Table 3.5 shows the comparison results.

Among five individual metrics model, NFR-C, CRM, SRM achieved relatively higher recall but relatively lower precision. For NFR-C, the main reason could be that, in OSS projects, stakeholders are more likely to describe the requirements’ function. Therefore, very little security indicator terms are used in their descriptions. The lower precision of CRM and SRM may indicate that the complexity is not only related to the security but also other non-functional requirements, such as performance.

On the contrary, URM and CiRM achieved relatively higher precision. This again demonstrated that URLs and commits are strong indicators for security requirements. However, the lower recall values also indicate that not all security requirements contains these two features. Therefore, other complementary features are needed to improve the performance of these two features.
The results also show that, linear combination of all features (LC) achieved the best performance in all three projects. It is noteworthy that LC does not only improve the recall, but also improve the precision. This again demonstrated that LC is an optimal way to combine all features.

Another risk may have negative impact on our experimental results is that answer sets generated in Section 3.1.1 are not completed. That is not all true security requirements are identified. However, we have fully confident that randomly selected datasets in the pilot study in Section 3.1.1 are completed (Table 3.3). Therefore, we apply LC to these datasets, results are showed in the latest row of Table 3.5. Statistic analysis (Wilcoxon signed-rank test) shows that there is no significant difference between results from different datasets.

3.4 Summary

In this chapter, we investigate six metrics that can help automated retrieval of security requirements from OSS projects and find an optimal way (i.e., linear classifier) to build security requirements classifier. We perform empirical studies on three long-lived, widely-used OSS projects. The results indicate that compared to the single metric, our approach could achieve higher recall and precision.

After getting all security requirements, a natural next step is to validate and verify implementations of those requirements. Traceability is recognized as a standard mean to support V&V task. In the next chapter, we propose an approach that bases on relevant feedback to help developers automatically build trace links between security requirements and source code.
Chapter 4

Automated Requirements Traceability

Security requirements V&V is one of the important development tasks that can ensure that security requirements are correctly implemented. However, manually creating and maintaining trace links from security requirements to other software documents like source code in large-size OSS projects is a mission impossible. Therefore, having an automatic approach to build trace links is essential for security requirements management.

4.1 Term-Based RF for Automated Tracing

It is important to point out here that, as far as the V&V of security requirements is concerned, tracing is a means, but not the means. Software testing represents another way to achieve V&V [47]; however, testing requires the software to be fully compilable, and at least partially, executable. Tracing therefore complements
Figure 4.1: Tracing dependability requirements via term-based RF.

methods like testing in that only textual information of the software artifacts is exploited.

Our analysis of security requirements in the experimental datasets shows that
these requirements contain specific terms indicating concerns like security and privacy. This observation is consistent with the dependability regulations such as HIPAA and IEC 61508 mentioned above. When RF is applied, it is this specific set of dependability terms, rather than all the terms in a candidate traceability link, whose weights should be adjusted. Note that need for finer-grained control over the terms was recognized by Shin and Cleland-Huang [70], but their approach was manual. We therefore contribute in Figure 4.1 an automated algorithm that integrates term-based RF in dependability requirements tracing.

We adopt the n-gram models [71] to capture the statistical regularity in the code (Lines 25–29 of Figure 4.1). We consider 2-grams to 6-grams to balance term usage context with noisy information [71]. Unlike previous research by Hindle et al. [71], if two n-grams have exactly the same terms but different orderings, we consider them as one n-gram and increase its frequency of occurrence accordingly. The frequency of occurrence of the n-grams of a given n is either flat or follows a power-law-like distribution [72]. In the latter case, we take the most appeared n-grams (i.e., head of long tail) as regularities.

The n-gram analysis is performed only on the five most important terms for each dependability requirement (Line 10 of Figure 4.1). Our main rationale is that RF is effective in the vector space IR model if the weights of a few dimensions (terms), rather than all the dimensions, are adjusted [49]. On average, 2.3 out of five top terms represent dependability concerns. For example, in a security-critical requirement of iTrust¹ (namely, Use Case 2), the top-5 terms are ‘admin’, ‘password’, ‘secret’, ‘database’, and ‘agent’. The first three terms are strongly dependability-oriented while the latter two are relatively general.

¹http://agile.csc.ncsu.edu/iTrust
In our algorithm, if a resulting term’s regular usage is empty, no RF will be further defined on top of it. This case shows that the term has high inverse document frequency value and therefore its appearance is relatively concentrated. An example is ‘electronic’ which appears in only one requirement of iTrust (Use Case 3). Adjusting the weight of such terms has little effect on improving retrieved results. For the same reason, if more than half of a requirement’s terms have no regular uses in the source code, we do not think RF should be applied to that requirement (Lines 14–15 of Figure 4.1).

For the remaining regular n-grams, we categorize them in Lines 17–23 of Figure 4.1 with three groups: $\alpha_{\text{RF}}$, $\beta_{\text{RF}}$, and $\gamma_{\text{RF}}$. If any term of the regular n-gram and the key term co-occur in the same requirements sentence, we think both the sentence and the source code where the n-gram occurs describe the same concept. As a result, such an n-gram’s terms’ weights should be increased. For example, one regular n-gram ‘change session time out’ is deemed positive feedback, since ‘session’ is one of the top-5 terms from iTrust’s Use Case 3 and the term ‘out’ co-occurs with ‘session’ in one of the sentences of Use Case 3: “An authenticated session ends when the user logs out or closes the iTrust application.”

In contrast, although “view” is a top-5 term of iTrust’s Use Case 21 and “view patient office visit history” is regular in the code, none of the terms from this n-gram except for “view” appear in the requirement at all. This poses a strong signal that “view” should be weighted less when tracing this iTrust requirement, because the regular code usage pattern of “view” bears little relationship to the requirement. Finally, if a term belongs to neither positive ($\beta_{\text{RF}}$) nor negative ($\gamma_{\text{RF}}$), then it falls into $\alpha_{\text{RF}}$ and its weight is kept intact.
Table 4.1: Traceability-related characteristics of dependability requirements (Dep.) and non-dependability requirements (Non-D.) in the experimental datasets

<table>
<thead>
<tr>
<th>Traceability-related characteristics</th>
<th>iTrust Dep.</th>
<th>iTrust Non-D.</th>
<th>WDS Dep.</th>
<th>WDS Non-D.</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace granularity</td>
<td>use cases →</td>
<td>Java methods</td>
<td>feature requests → Java classes</td>
<td></td>
</tr>
<tr>
<td># of req.s</td>
<td>13</td>
<td>22</td>
<td>11</td>
<td>171</td>
</tr>
<tr>
<td>average # of true links</td>
<td>8.69</td>
<td>8.77</td>
<td>9.45</td>
<td>10.99</td>
</tr>
<tr>
<td>range of true links</td>
<td>2 – 39</td>
<td>3 – 44</td>
<td>3 – 19</td>
<td>2 – 23</td>
</tr>
</tbody>
</table>

4.2 Experimental Evaluation

In this section, we evaluate our approach by applying it to two datasets. We first provide detailed information of two datasets in Section 4.2.1. Then, we discuss evaluation results and threads to validity in Section 4.2.2 and 4.2.3.

4.2.1 Datasets and Measures

Two datasets are used to conduct the experiments in this paper. Table 4.1 shows the traceability-related characteristics of the datasets. Both projects are developed in Java and the correct trace links are defined by projects’ original developers. Tables 4.2 and 4.3 list the dependability requirements.

The classification of dependability requirements in both projects is currently performed manually. We use the concepts presented in the previous study by Avizienis et al. [3] as guidelines. Take iTrust’s Use Case 1 (“Create and Disable Patients”) as an example, the requirement explicitly states that: “The HCP (health care professional) does not have the ability to enter/edit/view the patient’s security question/password.” We therefore mark Use Case 1 as an AC (access control) requirement. As shown in Tables 4.2 and 4.3, each dependability requirement is
classified into one or more categories. Figures 4.2 and 4.3 show the taxonomies where the dependability categories are fully defined.

We compare our term-based RF mechanism with three other requirements tracing methods: the TF-IDF vector space model [19] serving as the baseline, the standard Rocchio algorithm [48], and the adaptive Rocchio variant [21]. We evaluate the tracing methods along three dimensions: effectiveness, browsability, and dependability and security (DS) specificity. The standard metrics of IR-based tracing effectiveness are: recall, precision, and \( F_2 \), as defined in Equations (2.1)–(2.2). Browsability of the resulting ranked list of the traceability links complements the effectiveness measures because recall, precision, and \( F_2 \) are all set-based metrics. Following previous study by Hayes et al. [19], we adopt two browsability metrics: mean average precision (MAP) and Lag. We next describe a new metric to quantify the DS specificity of RF mechanism in the context of dependability requirements tracing.

The central idea of our DS specificity metric is to assess the extent to which a requirement expresses dependability needs. To that end, we first manually build a dependability taxonomy for each domain by following the grounded-theory approach presented in the previous study by Scott et al. [73]: Figure 4.2 for iTrust and Figure 4.3 for WDS. From left to right, the degree of DS specificity increases. For example, ‘close application’ and ‘terminate’ are more specific than ‘log out’ in Figure 4.2. Note that each taxonomy is constructed independent of tracing methods, especially the RF mechanism. We then leverage the taxonomy to formulate the ideal dependability representation of a particular requirement, which we denote as \( R_{dep} \). Finally, we compare \( R_{dep} \) with the requirement’s representation resulted from RF in order to calculate DS specificity.
Because RF essentially adjusts the weight of a requirement’s terms and/or adds new terms to the query requirement, our $R_{dep}$ construction has two phases: manipulating existing term’s weight and appending the requirement with new query terms. We illustrate the two phases with iTrust’s Use Case 9 (“View Records”). From Table 4.2, this requirement is an AUD (audit controls) requirement. Therefore,
the subtree of AUD in Figure 4.2 plays a key role in defining this requirement’s $R_{dep}$.

In the first stage, all the terms of Use Case 9 are checked, and if the term belongs to the AUD subtree, then we use the log-scale of the term’s depth [74] to adjust the term’s weight. The terms “visit” and “view” appear in both the original description of Use Case 9 and the AUD subtree. They are subject to weight increase. The TF-IDF value of “visit” and “view” are 0.47 and 0.79 respectively. The log-scale term depth in the AUD subtree is log(4)=0.60 for “visit” and log(5)=0.69 for “view”. Thus, after the first stage, the adjusted weight of “visit” and “view” is 1.07 and 1.48 respectively. The second phase takes the terms belonging to the AUD subtree and then adds them with their log-scale depth weights to the $R_{dep}$ of Use Case 9. For instance, “edit” is appended with the weight of log(3)=0.48. Note that the weight of “visit” and “view” is adjusted again in stage two, making 1.67 and 2.17 their final weights in $R_{dep}$.

The resulting $R_{dep}$ is a new vector $(w_{t_1}, w_{t_2}, \ldots, w_{t_m})$ where $m$ is the number of total terms in the domain vocabulary. According to the dependability type of the requirement and the taxonomy related to the type, we consider $R_{dep}$ best reflects the dependability concerns of the requirement and thus refer it to the ideal query requirement representation. Using $R_{dep}$, we define $DS$ specificity as follows:

$$DS\text{Specificity} = \sqrt{\sum_{i}^{m} (w_{t_i} - w'_{t_i})^2}.$$  \hspace{1cm} (4.1)

where $w_{t_i}$ is the term weight in $R_{dep}$ and $w'_{t_i}$ is the term weight in the dependability requirement adjusted by a given RF mechanism. Thus, DS specificity computes
Figure 4.2: iTrust dependability taxonomy: TS (transmission security), AC (access control), INT (integrity), PA (person or entity authentication), AUD (audit controls), IC (integrity control), AL (automatic logoff), UUI (unique user identification), EAP (emergency access procedure), EAD (encryption and decryption), MP (mechanism to authenticate electronic protected health information), ER (emergency responder), HCP (health care personnel), UAP (unlicensed authorized personnel), PHA (public health agent), LT (lab technician), LHCP (licensed HCP), DLHCP (designated LHCP), ULHCP (unlicensed LHCP).
Figure 4.3: WDS dependability taxonomy: WIA (workforce investment act).
the distance between a requirement’s representation undergone RF and its ideal representation reflecting the particular dependability concerns. The less the DS specificity value according to Equation (4.1), the better the RF algorithm in transforming the dependability requirement to its ideal representation.

4.2.2 Results

We analyze the experimental results based on our three evaluation goals: effectiveness, browsability, and DS specificity. Our analyses are divided into dependability requirements (i.e., those in Tables 4.2 and 4.3) and non-dependability requirements, following the dependability requirements classification described in Section 4.2.1. We pre-processed the software artifacts in a uniform way. In particular, an expanded indexer that handles both natural-language requirements and Java source code was deployed [22]. For RF, we instantiated the parameters as $\alpha=1.0$, $\beta=0.75$, and $\gamma=0.25$ because these values exhibit consistency and robustness in both traditional IR [49] and automated traceability [19, 21].

Arguably, effectiveness is one of the most important criteria used to evaluate IR-based tracing methods. Metrics like recall, precision, and $F_2$ provide measures toward the tracing algorithm’s accuracy as well as the automated tool’s believability [19]. Table 4.4-a presents the descriptive statistical results on the iTrust dataset when the 70% threshold is applied to the tracing methods (i.e., only evaluating the top 70% of retrieved candidate links). To compare the performance of the four different tracing methods, a pairwise Bonferroni-Holm correction is conducted and the effect sizes are computed using $\hat{A}_{12}$ non-parametric statistics [75]. The inferential results are reported in Table 4.4-b, where statistically insignificant $p$ values are given without $\hat{A}_{12}$ values. The $p$ values indicate significant difference between
Table 4.4: RF performances in tracing iTrust’s dependability requirements (D) and non-dependability requirements (Non-D): R: recall; P: precision; SR: standard Rocchio; AR: adaptive Rocchio; TB: term-based RF

(a) Descriptive statistics

<table>
<thead>
<tr>
<th></th>
<th>TF-IDF</th>
<th>SR</th>
<th>AR</th>
<th>TB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D</td>
<td>Non-D</td>
<td>D</td>
<td>Non-D</td>
</tr>
<tr>
<td>Effectiveness</td>
<td>R</td>
<td>0.79</td>
<td>0.83</td>
<td>0.84</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>0.07</td>
<td>0.09</td>
<td>0.10</td>
</tr>
<tr>
<td></td>
<td>$F_2$</td>
<td>0.25</td>
<td>0.30</td>
<td>0.34</td>
</tr>
<tr>
<td>Browsability</td>
<td>MAP</td>
<td>0.29</td>
<td>0.35</td>
<td>0.38</td>
</tr>
<tr>
<td></td>
<td>Lag</td>
<td>215</td>
<td>156</td>
<td>98</td>
</tr>
</tbody>
</table>

(b-1) Inferential statistics (Due to space constraint, only dependability requirements’ results are listed. The reported $p$ value is in the $10^{-3}$ scale. SR: standard Rocchio, AR: adaptive Rocchio, TB: term-based RF.)

<table>
<thead>
<tr>
<th></th>
<th>Effectiveness</th>
<th></th>
<th></th>
<th>F_2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Recall</td>
<td>Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SR vs TF-IDF</td>
<td>33.24</td>
<td>16.3 ($\hat{A}_{12}=0.60$)</td>
<td>50.39</td>
<td></td>
</tr>
<tr>
<td>AR vs TF-IDF</td>
<td>10.3 ($\hat{A}_{12}=0.76$)</td>
<td>9.21 ($\hat{A}_{12}=0.84$)</td>
<td>24.97</td>
<td></td>
</tr>
<tr>
<td>AR vs SR</td>
<td>146.3</td>
<td>62.73</td>
<td>63.31</td>
<td></td>
</tr>
<tr>
<td>TB vs TF-IDF</td>
<td>6.34 ($\hat{A}_{12}=0.80$)</td>
<td>4.73 ($\hat{A}_{12}=0.88$)</td>
<td>6.32 ($\hat{A}_{12}=0.84$)</td>
<td></td>
</tr>
<tr>
<td>TB vs SR</td>
<td>11.4 ($\hat{A}_{12}=0.66$)</td>
<td>6.38 ($\hat{A}_{12}=0.84$)</td>
<td>17.98</td>
<td></td>
</tr>
<tr>
<td>TB vs AR</td>
<td>12.9 ($\hat{A}_{12}=0.58$)</td>
<td>7.32 ($\hat{A}_{12}=0.89$)</td>
<td>19.36</td>
<td></td>
</tr>
</tbody>
</table>

(b-2) Inferential statistics (Due to space constraint, only dependability requirements’ results are listed. The reported $p$ value is in the $10^{-3}$ scale. SR: standard Rocchio, AR: adaptive Rocchio, TB: term-based RF.)

<table>
<thead>
<tr>
<th></th>
<th>Browsability</th>
<th></th>
<th></th>
<th></th>
<th>DS Specificity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MAP</td>
<td>Lag</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SR vs TF-IDF</td>
<td>30.97</td>
<td>58.86</td>
<td></td>
<td>127.5</td>
<td></td>
</tr>
<tr>
<td>AR vs TF-IDF</td>
<td>1.95 ($\hat{A}_{12}=0.86$)</td>
<td>0.48 ($\hat{A}_{12}=0.71$)</td>
<td>9.48 ($\hat{A}_{12}=0.72$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AR vs SR</td>
<td>100.3</td>
<td>19.6</td>
<td></td>
<td>224.7</td>
<td></td>
</tr>
<tr>
<td>TB vs TF-IDF</td>
<td>0.23 ($\hat{A}_{12}=0.88$)</td>
<td>0.14 ($\hat{A}_{12}=0.78$)</td>
<td>2.61x$10^{-5}$ ($\hat{A}_{12}=0.88$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB vs SR</td>
<td>12.98</td>
<td>2.68</td>
<td>($\hat{A}_{12}=0.65$)</td>
<td>0.056 ($\hat{A}_{12}=0.84$)</td>
<td></td>
</tr>
<tr>
<td>TB vs AR</td>
<td>20.35</td>
<td>392.2</td>
<td></td>
<td>0.061 ($\hat{A}_{12}=0.86$)</td>
<td></td>
</tr>
</tbody>
</table>
Figure 4.4: Recall-precision analysis of iTrust: dependability requirements tracing (top); non-dependability requirements tracing (bottom).
two datasets (e.g., SR and TF-IDF). For those with significant differences, the $\hat{A}_{12}$ values further tell us the effective sizes of the differences. Following previous study by Vargha et al. [75], we use the intervals defined by 0.56, 0.64, and 0.71 to distinguish small, medium, and large effect sizes.

Table 4.4-b shows that, compared to the baseline TF-IDF method, our term-based RF algorithm achieves significantly better performances in all the three areas of effectiveness, browsability, and DS specificity. Furthermore, the effect sizes are large. Our method also outperforms the standard and adaptive Rocchio in a significant manner. This trend can be readily visualized in Figure 4.4 where 10 cutoff points (0.1, 0.2, \ldots, 1.0) are applied on recall.

When tracing non-dependability requirements, our term-based RF algorithm has mixed performances when compared with the adaptive Rocchio method. As shown in the right of Figure 4.4, term-based RF has a plateau at the low recall values. We conjecture the main reason is because, for non-dependability requirements, their top-5 most important terms are often too general. For example, iTrust’s UC 17 (“Proactive Determine Needed Patient”) returns ‘patient’, ‘month’, ‘week’, ‘immunization’, and ‘alphabetical’. Fine-tuning the weights of these terms, as opposed to performing link-based RF like standard or adaptive Rocchio, leads to more false links to be ranked higher in the tracing results, e.g., the links containing ‘patient name’ and ‘alphabetical sort’. In this sense, dependability requirements are more suited for the term-based RF treatment (cf. left of Figure 4.4). The tracing effectiveness of WDS exhibits the same trend as iTrust. Due to the space constraint, only the $F_2$ statistics are summarized in Figure 4.5.

Similarly to effectiveness, the browsability measures show the superior performance of our term-based RF method over the other three methods. For DS
Figure 4.5: Tracing effectiveness on WDS. White boxes: dependability and security requirements; gray boxes: non-dependability and non-security requirements specificity, only the dependability requirements are compared among the four tracing methods. Recall that DS specificity captures the distance between a dependability requirement’s ideal representation ($R_{dep}$) and its representation adjusted by RF. For TF-IDF, no RF is applied and therefore DS specificity is the lowest. Surprisingly, standard and adaptive Rocchio do not improve DS specificity significantly, as shown by the results of Table 4.4 and Figure 4.6. In contrast, DS specificity is enhanced by term-based RF, indicating that the trace query resulted from our algorithm is the closest to the dependability concerns that the requirement intends to express.

4.2.3 Threats to Validity

We mitigate the threats to construct validity [76] by considering three performance facets: effectiveness, browsability, and DS specificity. For the first two, we
adopt standard IR metrics [23]. For DS specificity, one limitation is our manual construction of the domain-specific dependability taxonomy. While ontology engineering has dramatically advanced in the past several decades, fully automated ways to build the knowledge base for dependability hardly exist. Even though researchers may be eager to advocate their exciting ontology building techniques, we argue that certain level of manual intervention, like quality control or consistency management, is unavoidable. In this sense, the taxonomies of Figures 4.2 and 4.3 should be regarded only as starting points and are subject to subsequent refinement and maintenance. While this limitation should have little effect on the comparisons, caution must be taken in interpreting the absolute values of DS specificity.

We believe the main strength of our experimental design is its high *internal*
validity [76]: soundness of the relationship between independent and dependent variables. Because all the factors potentially affecting the responses (effectiveness, browsability, and DS specificity) are under our direct control, any significant difference must be caused by the different requirements tracing methods employed.

The results of our analysis may not generalize to other dependability requirements tracing datasets — a threat to the external validity [76]. Our chosen systems cover both safety-critical and mission-critical applications. However, these are not necessarily representative of all dependable industrial systems and, in particular, embedded software products are likely to exhibit different characteristics.

4.3 Discussion

An analytical comparison pinpointing the theoretical improvements of RF mechanisms over the baseline TF-IDF method is as follows: standard Rocchio [48] adjusts the weights for the candidate traceability links, adaptive Rocchio [21] adjusts those for only a subset of the links, and our approach performs weight adjustment for only the selected terms within each link.

One of the key results is that our algorithm outperforms the standard and adaptive Rocchio in tracing dependability requirements. An explanation is that the two Rocchio methods instrument the RF at the link level, that is, if a candidate traceability link is marked as a positive (or negative) feedback, then all the terms of that link will be treated as positive (or negative) to modify the query requirement.

In contrast, our method works at the term level, which allows for different treatments of different terms. In our algorithm presented in Figure 4.1, the requirements term is checked against its usage in the code base. When the gram
containing the term is used in a regular and repetitive way in code, we then analyze the term’s context in the requirement and use this information to determine the RF type: positive, negative, or uncertain (unchanged). The main benefit of the term-based RF algorithm, in our opinion, is the finer-grained control over the term weighting, especially for the terms that appear in the same requirement.

It turns out that dependability requirements are commonly expressed with specific terms, and these terms are not only domain-specific but also concern specific [77] and task specific [78]. For these reasons, we constructed two taxonomies for the systems that we studied, and further used the taxonomies to define a new metric to measure DS specificity. The experimental results show that our term-based RF method was able to transform the original query requirement into a form that is the closest to the requirement’s dependability concerns. In this sense, the superior performances of our algorithm in effectiveness and DS specificity are not independent but correlated. Finally, it is worth mentioning that our term-based RF method can potentially improve the tracing of other nonfunctional requirements (such as portability and interoperability) than dependability, as well as those concerns related to dependability (such as safety and reliability). Testing the effect of term-based RF on other concerns requires further experiments, and if DS specificity is to be assessed using our method, new taxonomies or other knowledge representations of concern-dependent terms.

The implications of our work are two-fold. For researchers working in requirements traceability, our algorithm represents a significant departure from applying traditional methods in IR. In fact, one of the latest developments, namely the adaptive Rocchio method [21], critically revisits the underlying assumptions of RF and modifies how Rocchio should be operated in requirements tracing. Sim-
ilarly, the method proposed in this paper, to the best of our knowledge, is the first algorithm that automates RF at the term level. Such a novelty combines the naturalness of software [71] and the DS specificity of dependability requirements. More importantly, we anticipate our work to illuminate the practitioners with further automation to reduce the manual effort in engineering dependable industrial systems. As is commonly believed: One cannot patch security—for the same reason, dependability—late in the development life cycle. The most cost-effective stage of building dependability is therefore in requirements engineering. Our work illustrates that engineering dependability requirements is not only about elicitation. One also needs V&V and our algorithm represents a new way to achieve both effectiveness and automation. We therefore encourage researchers and practitioners to go beyond traditional domains like IR so that transformative innovations can be made to best fit the critical tasks of requirements analysts, software developers, system assurers, and other industrial engineers.

4.4 Summary

The main contribution of this chapter is to propose an automated trace link building approach based on term-based relevant feedback. The experiment on two OSS projects demonstrates that our new approach, compared to baselines, improves the accuracy of automated tracing. With traceability, follow-up tasks like V&V and testing can be applied to security requirements. However, such tasks are well defined but implemented within single requirements. However, the requirements dependency plays an important role in causing security vulnerability. Therefore, finding requirements dependencies becomes the key to the success of vulnerability
detecting.

In the next chapter, we propose a semi-automated approach to detect security requirements dependencies. Those dependencies are actually test paths for security testing. With those test paths, different testing approaches like fuzzing and penetration testing could be applied to detect vulnerabilities.
Chapter 5

Test Paths Detection

Testing is recognized as an effective and efficient approach to finding security vulnerabilities. However, the state-of-the-art security testing approaches are applied within single security requirements. Vulnerability found in Facebook demonstrated that dependencies between security requirements cause security issues. Therefore, evaluating security requirements dependencies shall be considered as an important and necessary part of vulnerabilities detection. In fact, such dependencies are test paths for security testing. With those dependencies, security test cases could be generated to detect vulnerabilities in OSS projects. In this chapter, we propose a new semi-automated security requirements dependency detection approach and apply our approach to five OSS projects to evaluate its effectiveness and efficiency.

5.1 Dependency Detection Approach

Figure 5.1 presents our three-step requirements dependency detection approach. In the first step (Figure 5.1 ❶), security experts are asked to manually detect dependencies between HSRs. In the second step (Figure 5.1 ❷), a machine learning
Figure 5.1: Dependency detection: HSR: high-level security requirement; LSRs: low-level security requirements

approach is applied to retrieve associated LSRs for each HSR. In the final step (Figure 5.1 3), a semantic approach is designed to identify relations between two LRSs according to the type of the relation between HSRs associated with them. One of the limitations of our approach is in the first step, i.e., manually detecting HSRs’ dependencies costs much time. However, experts who have strong security background involved in this step provides two benefits: ① it increases our confidence that all validated HSRs’ dependencies are detected; and ② it also increases our confidence that all detected HSRs’ dependencies are validated. The following steps that automatically detect LSRs’ dependencies are highly dependent on the results of the first step. Therefore, benefit ① will increase the possibility of finding all LSRs’ dependencies, thus increasing the possibility of finding more vulnerabilities. At the same time, benefit ② will increase the accuracy of detected LSRs’ dependencies, thus reducing the time wasted on testing invalidated dependencies. We provide a detailed description of each step as follows.
Regulations like HIPAA provide guidance on protecting information security. Detecting dependencies between HSRs defined in regulations could benefit not only security testing but also other software engineering tasks. The completeness of regulations ensures that dependencies between HSRs cover all scenarios in the real world. The relatively small size of HSRs in regulations reduces the effort on dependencies assessment. The stability of regulations reduces the effort spent on dependencies maintenance. In addition, since one security regulation is enacted to address security problems in a particular area, the results of dependency assessment can be reused to many other systems under the same area.

In this research, we applied our approach to security requirements defined in HIPAA and FERPA. For HIPAA, a set of few security requirements is summarized in the previous study by Cleland-Huang et al. [40]. A couple of studies were conducted based on it [40, 79]. For FERPA, the United State Department of Education lists the standard “Minimum Security Requirements for Federal Information and Information Systems” (FIPS 200) [80] as the best practice for addressing security requirements and recommends that online education services shall use FIPS 200 to help address legal requirements in FERPA [81]. FIPS 200 is a standard that was summarized from success cases of Federal Government Agencies information system. According to the Federal Information Security Management Act (FISMA) [82], all government agencies’ information systems shall meet the minimum set of security requirements specified in FIPS 200.

Two students who have at least one year research experience on security requirements and related topics are asked to detect HSRs dependencies individually. Then their analysis results are merged in such a way: if both of them have the same
judgment on one HSRs pair (either there is dependence or there is no dependence), we record the judgment. Otherwise, a joint meeting is conducted to make the final decision. The average Cohen’s kappa coefficient [83] analysis on two analysts’ results before the joint meeting ($\kappa = 0.63$) shows that they highly agree with each other. This value also demonstrates the high-quality of the results of step 1.

A systematic analysis of the results shows that detected HSRs’ dependencies in HIPPA and FIPS 200 fall into five scenarios. We next discuss all scenarios with the intention that the knowledge can be transferred to HSRs’ dependencies detection in other regulations. These hints also impact the design of the semantic approach in step 3.

**Scenario I:** If requirement $HSR_1$ creates, modifies, or deletes a resource or attributes of a resource that is used in another requirement $HSR_2$, we would like to label that there is a dependency $HSR_1 \rightarrow HSR_2$ (i.e., $HSR_1$ depends on $HSR_2$). An example is AU-I (audit and accountability I) $\rightarrow$ SC-I (system communication protection) in FIPS 200, where AU-I creates information system audit records to the extent needed to enable the monitoring and SC-I monitors information transmitted or received by the information system. If an LSR $LSR_1$ satisfies AU-I by recording audit information in a log file but does not follow the format used in another requirement $LSR_2$ which satisfies SC-I, then a potential security risk is that $LSR_2$ will miss illegal communications/attacks which may steal or destroy information saved in the system because $LSR_2$ would ignore log items that do not match the specific format.

**Scenario II:** If there is a clear chronological relation between $HSR_1$ and $HSR_2$, we mark that there is a dependency between them. An example is the dependency between AC (access control) and AL (automatic logoff) in HIPAA. Only after a user
logs in the system by passing the access control, it is necessary for the system to automatically log out the user after a predetermined time of inactivity. Therefore, we label that AL depends on AC. A security issue may happen when the daytime saving time ends. Suppose AC records that a user logs into the system at 1:59 am on March 11. This user shall be automatically logged out if she inactive for one hour. However, due to time is automatically set back one hour (i.e., the time is automatically changed to 2 am), the system will not log her out. This error directly violate the security requirements in HIPAA.

**Scenario III:** If requirement $HSR_1$ describes the general goal, and another one $HSR_2$ specifies a detailed approach to achieve this goal, we say that $HSR_1$ depends on $HSR_2$. For instance, in FIPS 200, MP-I (organizations must protect information system media, both paper and digital) sets the general goal of protecting system media. MP-II (organizations must limit access to information on information system media to the authorized user) and MP-III (organizations must sanitize or destroy information system media before disposal or release for reuse) specify “how to” protect system media under certain circumstances. Thus, we recognize MP-I depends on MP-II and MP-III. Imaging that a system adds a new type of media (e.g., pdf files) and an LSR $LSR_1$ to ensure confidentiality, integrity, and availability. However, developers of $LSR_2$ that satisfies MP-III may not be aware of this new type of media and forget to sanitize it (e.g., via automatically scanning and removing sensitive information in it), the sensitive data saved in this type of media could then be accidentally disclosed to malicious attackers in the next release.

**Scenario IV:** If requirement $HSR_2$ has to react to the change while requirement $HSR_1$ is took place, the dependency $HSR_2 \rightarrow HSR_1$ is recorded. For example, in
FIPS 200, after AC (access control) rejects login request from the same IP address several times in a predetermined time since the username and password pairs are not matched, PS-III (organizations must employ formal sanctions for personnel failing to comply with security policies and procedures) shall automatically punish this IP address by putting it in the blacklist. Therefore PS-III→AC is a valid dependency. A potential security risk will happen when AC handles login request in parallel. Attackers may send two requests from one IP address at the same time, AC may record all rejections for that IP address in one log item, however PS-III may treat it as one rejection. This mismatch allows attackers to try more combinations before being blocked, thus increasing the possibility that attackers figure out the correct combination. Sometimes, the requirements dependency also exists in the opposite direction. For instance, in the relationship AC→PS-III, AC shall automatically disable the login function to the IP addresses in the blacklist after PS-III applies the punishment.

**Scenario V:** If two HSRs do not fall into the above scenarios but share important keywords, this indicates that they will be implemented by using the same or similar technique. For instance, SED (encryption and decryption) and TED (encryption) in HIPAA share keyword “encryption.” The former one refers to storage encryption and decryption and the latter one refers to transaction encryption and decryption. If two LSRs \( LSR_1 \) and \( LSR_2 \) satisfy them by using different encryption approaches, developers will be confused and may mistakenly use encryption approach for storage security in transaction security. Thus, the data integrity may be harmed. We shall point out that this type of dependencies is a two-way relationship. That means we shall test both \( LSR_1 \rightarrow LSR_2 \) and \( LSR_2 \rightarrow LSR_1 \). In other words, test cases shall be carefully designed to make sure that encryption method for transaction reports
error when the data to be stored in the database is used, and vice versa.

We present the assessment results of the FIPS 200 in Figure 5.2. Each node in Figure 5.2 represents a HSR in FIPS 200. It must be noted that we do not include HSRs that are not related to software development in our analysis. For instance, physical and environment protection-IV (PE-IV) asks that organization must protect information systems against environmental hazards. It can be implemented by physical protections like using fireproof materials to build the facility house, not software. Therefore, we do not consider it in our approach. If there exists a dependency between two HSRs, we link their nodes in Figure 5.2. In our first step, 49 dependencies are detected from FIPS 200 and 14 dependencies are found in HIPAA.

5.1.2 Automated Requirements Traceability

Manually and painstakingly creating requirements traceability matrix (RTM) is a mission impossible in industrial projects. The approaches based on information
retrieval (IR) provide cost-efficient traceability. The term mismatch problem dramatically harms the performance of the IR-based algorithms while tracing regulations. There are many different approaches proposed to solve the term mismatch problem including probabilistic network (PN) models [40], latent semantic indexing (LSI), and relevant feedback (RF) [19]. Our previous research [84] shows that term-based RF can significantly improve the performance of dependency and security requirements tracing. We therefore adapt the term-based RF by using indicator terms detection method [40] to help find feedback terms, and apply the term-based RF to detect relationships between HSRs and LSRs.

RF has the power to address the term mismatch problem because it adjusts the searching queries for IR methods by adding terms used in true trace links. Our previous research demonstrated that, compared to standard RF [19] and adaptive RF [21], term-based RF significantly improves the effectiveness of IR-based approaches while tracing security and dependability requirements to the source code [84]. For traceability from HSRs to LSRs, experimental results of machine learning approaches [40] showed that LSRs contains specific terms expressing different security concerns like integrity and access control. When applying RF, it is this specific set of indicator terms, rather than all terms in a true trace link that shall be added to the searching query. Term mining approach proposed as part of web mining approach [40] has been demonstrated to be effective for identifying those indicator terms. Therefore, we combine these two approaches together in our automated HSRs to LSRs traceability. The steps are described in detail as follows:

**Retrieving Relevant Documents:** Unlike the web mining approach [40] which collects HSRs’ related documents by web searching, we retrieve related documents from LSRs. Compared to the online materials, LSRs contain more
project specific terms that are more powerful for retrieving true trace links. The IR-based traceability returns a ranked list of LSRs, then top-\(n\) LSRs are used as the collection of domain-relevant documents \(D_r\) for term mining.

**Mining Indicator Terms:** For a term \(t\) in \(D_r\), two metrics are calculated [40]: domain specificity (DS) and concept generality (CG). DS measures the extent to which a term is specific to the domain document, as opposed to occurring frequently across a broad spectrum of topics. It can be calculated as follows:

\[
DS(t) = \ln \left( \frac{\sum_{t_i \in D_r} freq(t, D_r)}{\sum_{t_i \in D} freq(t_i, D)} \right) \tag{5.1}
\]

where the first component \(\sum_{t_i \in D_r} freq(t, D_r)\) is the normalized term frequency of \(t\) in top-\(n\) LSRs. Following the best practice of RF [19], we use top-2 LSRs as domain-relevant documents. The second component is the normalized term frequency of \(t\) in all LSRs \(D\).

CG computes the fraction of domain specific documents in which a specific term occurs:

\[
CG(t) = \frac{|D_t|}{|D|} \tag{5.2}
\]

where \(|D_t|\) indicates the number of LSRs contain \(t\) and \(|D|\) is the size of LSRs. Following the heuristics proposed in the previous study by Cleland-Huang et al. [40], \(t\) is marked as an indicator term if \(DS(t) \geq 5 \& CG(t) \geq 0.3\).

**Applying Relevance Feedback:** After collecting indicator terms \(T\), we apply RF to modify \(HSR\)'s initial query vector \(Q_o\):

\[
\overrightarrow{Q_m} = (\alpha \cdot \overrightarrow{Q_o}) + (\beta \cdot \frac{1}{|D_r|} \cdot \sum_{t_i \in T \& t_i \in D_r} \overrightarrow{d_t}) \tag{5.3}
\]
where \( \frac{1}{|D_r|} \cdot \sum_{t_i \in D_r} \overrightarrow{d_t} \) presents average term frequency-inverse document frequency (TFIDF) value of \( t \) in \( D_r \). Then new query \( \overrightarrow{Q_m} \) is used to retrieve LSRs. According to the previous studies by Hayes et al. [19] and Wang et al. [84], best setting for the size of \( D_r \), feedback weights \( \alpha \) and \( \beta \) are 1.0, and 0.75 respectively. Following the practice of our previous study et al. [84], we use top 75% retrieved LSRs as candidate links for \( R \).

### 5.1.3 Semantic Analysis

As previously stated, a practical strategy for reducing assessment cost is that: if in Section 5.1.1, there is no dependency between two HSRs, there is no dependency between their associated LSRs that are retrieved in Section 5.1.2. Otherwise, the additional assessment shall be applied to 1) filter out false positives introduced in automated tracing (Section 5.1.2) and 2) validate dependencies between LSRs retrieved for these two HSRs pair-wisely. The reason we need the second step is that each LSR implements one or several aspects of an HSR. For instance, an LSR [85], which implements MP-I protects the integrity of data during database migration via setting features that related database writing actions invisible. Another LSR [86], which implements MP-II, prevents users without permissions from downloading files like video and pdf. There are no clear relationships between these two LSRs even their associated HSRs interact with each other.

Even analyzing LSRs associated with interacting HSRs can reduce assessment cost, manually validating LSRs dependencies is a time-consuming task because of the large size of requirements in industrial projects. From five scenarios described in Section 5.1.1 we know that there are semantic relations between interacting HSRs. These semantic relations between HSRs are automatically inherited by dependencies
Figure 5.3: Feedforward neural network model for low-level security requirements (LSRs) dependency detection

between their LSRs. Previous studies showed that artificial neural network is a powerful approach to detect those semantic relationships [87, 88]. Therefore, we train a feedforward neural network (FNN) by using five features related to five HSRs dependency scenarios to help us detect semantic relationships between LSRs. We discuss these features as well as the structure of FNN as follows.

5.1.3.1 Features Selection

Five features derived from five scenarios (Section 5.1.1) are used in our FNN-based semantic relationship detection approach. We discussed how these features relate to previous scenarios as follows:

- According to scenario I, resources are bridges of two requirements. Those resources are described in two LSRs by using the same noun or nouns with the similar meaning. Therefore, the first feature of our FNN is a set of nouns used in two LSRs.

- According to scenario II, there is a chronological order between two actions in two LSRs. For instance, “login” should obviously happen earlier than “log
off”. To learn this type of semantic relation, all verbs in two LSRs are used to train our FNN model.

- In scenario III, sub-task indicates that the operation in one LSR is part of another operation in the parent LSR. For instance, “limit access” is one approach of “protect media”. To learn this type of dependencies, the feature a set of verbs and their objects is used in our FNN-based approach.

- In scenario IV, one LSR triggers another LSR. Our observation from two pilot projects iTrust\(^1\) and Scholar@UC\(^2\) indicates that one LSR recording another LSR’s ID is clear evidence that two LSRs fall into this scenario. For example, one LSR UC11 (document office visit\(^3\)) in iTrust states “the health care personnel (HCP) documents the following information related to an office visit and all events are logged (UC5)”. It means that events (documenting office visit related information) in UC11 will trigger log action in UC5 (log transaction\(^4\)). Therefore, a set of LRSs’ IDs (including to be detected LSRs’ IDs and LSRs’ IDs recorded in them) is another feature of our FNN-based approach.

- The dependency in scenario V is based on shared keywords. It is easy to understand that a set of shared terms in two LSRs could be used as a strong feature for our FNN model.

---

\(^1\)iTrust is a medical application that provides patients with a means to keep up with their medical history and records as well as communicate with their doctors. Website: [https://152.46.18.254/doku.php](https://152.46.18.254/doku.php)

\(^2\)Scholar@UC is a system that enables students and faculties from the University of Cincinnati to share research and scholarly works with worldwide audience. Website: [https://scholar.uc.edu](https://scholar.uc.edu)

\(^3\)[https://152.46.18.254/doku.php?id=requirements:uc11](https://152.46.18.254/doku.php?id=requirements:uc11)

\(^4\)[https://152.46.18.254/doku.php?id=requirements:uc5](https://152.46.18.254/doku.php?id=requirements:uc5)
5.1.3.2 Dependency Detection Model

Figure 5.3 presents our FNN model. The input layer of the FNN model is the combination of word embeddings of five features we discussed in Section 5.1.3.1. Instead of representing terms as single weights, the word embedding presents each word as a continuous high dimensional vector. The vector is computed from not only the term itself but also the distribution of terms around it. In other words, semantically similar or related words are close to each other in the vector space [89].

Three hidden layers are conducted to learn by using rectified linear unit (ReLU) neurons [90]. Each neuron in a hidden layer is fully connected to the previous layer. The activation function of neurons in \( i \)th hidden layer \( h_i(LSR_1, LSR_2) \) is based on all neurons in the previous layer \( h_{i-1} \):

\[
h_i(LSR_1, LSR_2) = \max(0, W_i h_{i-1}(LSR_1, LSR_2)) + b_i
\]

where \( W_i \) is a weights vector and \( b_i \) is a bias vector. Compared to other activation functions, such as sigmoid, ReLU allows efficient and effective training of FNN on large and complex data. Recent research was done by Glorot \textit{et al.} [91] showed that ReLU is remarkably suitable for naturally sparse data. According to the previous study by Carlshamre \textit{et al.} [38], 75% of dependencies are detected in 20% of total requirements. In other words, the dependency space of remaining 80% of requirements is sparse. Therefore, we believe that our FNN model based on ReLU is suitable for detecting dependencies between LSRs.

The output layer is similar to hidden layers. The only difference is that there is
Table 5.1: Projects used to evaluate requirements dependency detection approach: LSRs: low-level security requirements; PID: project id; D: dependency; A: assessment (cost)

<table>
<thead>
<tr>
<th>Regulation</th>
<th>Project</th>
<th>PID</th>
<th># LSRs</th>
<th># D</th>
<th># A</th>
</tr>
</thead>
<tbody>
<tr>
<td>HIPAA</td>
<td>CARE2X</td>
<td>P1</td>
<td>24</td>
<td>22</td>
<td>276</td>
</tr>
<tr>
<td></td>
<td>iTrust</td>
<td>P2</td>
<td>36</td>
<td>43</td>
<td>630</td>
</tr>
<tr>
<td></td>
<td>WorldVistA</td>
<td>P3</td>
<td>116</td>
<td>193</td>
<td>6,670</td>
</tr>
<tr>
<td>FIPS 200</td>
<td>P4</td>
<td>Moodle</td>
<td>144</td>
<td>267</td>
<td>10,296</td>
</tr>
<tr>
<td></td>
<td>Scholar@UC</td>
<td>P5</td>
<td>1182</td>
<td>1396</td>
<td>973,710</td>
</tr>
</tbody>
</table>

Table 5.2: Results for requirements dependency detection: PID: project id in Table 5.1; LS: lexical similarity; ↓: reduction; R: recall; P: precision; ∗: student’s t-test p-value less than 0.05; V: vulnerability

<table>
<thead>
<tr>
<th>PID</th>
<th>LS</th>
<th>Our Three-Step Approach</th>
<th># V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
<td>Performance</td>
<td>Cost</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>101</td>
<td>.63</td>
<td>.68</td>
</tr>
<tr>
<td>P2</td>
<td>153</td>
<td>.76</td>
<td>.51</td>
</tr>
<tr>
<td>P3</td>
<td>1,098</td>
<td>.83</td>
<td>.72</td>
</tr>
<tr>
<td>P4</td>
<td>2,732</td>
<td>.73</td>
<td>.64</td>
</tr>
<tr>
<td>P5</td>
<td>148,766</td>
<td>.87</td>
<td>.59</td>
</tr>
</tbody>
</table>

only one neuron in it:

\[
s_o(LSR_1, LSR_2) = \max(0, W_o h_3(LSR_1, LSR_2)) + b_o \quad (5.5)
\]

Instead of having a common threshold for all pairs of LSRs, we detect dependency between LSR\(_1\) and LSR\(_2\) if and only if \(s_o(LSR_1, LSR_2) \geq s_o(NA, LSR_1)\) and \(s_o(LSR_1, LSR_2) \geq s_o(NA, LSR_2)\), where \(s_o(NA, LSR_i)(i = 1, 2)\) presents the probability that LSR\(_i\) has no dependency.
5.2 Experimental Evaluation

We applied our three-step approach to two regulations: HIPAA and FIPS 200. Twelve HSRs are elicited from HIPAA (See the previous study by Cleland-Huang et al. [40]) and 29 HSRs are suggested by FIPS 200. For HIPAA, we searched all 10 healthcare projects used in the previous research by Cleland-Huang et al. [40]. However, some of them are no longer available (e.g., ClearHealth). For others, since we need to dynamically test the project, we only choose those projects that provide either running demos or the source code. For FIPS 200, we test it by using two open source education software Moodle\(^5\) and Scholar@UC. The basic information of these projects related to our study is reported in Table 5.1.

The second last column of Table 5.1 displays the number of dependencies detected via manually pairwise assessment and the very last column shows the cost of manual analysis. For each project, two researchers who have at least one year security research experience built answer sets of LSRs’ dependencies individually and reached a substantial degree of agreement: average Cohen’s kappa=0.69 on these projects. Then, if a dependency is identified by both of them, it will be added to the answer set. Otherwise, a joint meeting is held to solve the discrepancy.

We compared our approach with the lexical similarity (LS) approach proposed by Carlshamre et al. [38]. Following strategies in the previous research by Carlshamre et al. [38] while applying LS, we first remove singular requirements and retrieve requirements that fall into the four categories. Then the pairwise assessment of lexical similarity is applied to retrieved requirements. According to the previous studies by Carlshamre et al. [38] and Natt och Dag et al. [92], if the lexical similarity score is greater than 0.125, two requirements are interdependent. For our approach,

\(^5\)https://moodle.org
we first manually build dependencies between HSRs. Then the automated RF-based approach is applied to retrieve LSRs for each HSR. Finally, the 10-fold approach is applied to evaluate our three-step approach.

There are two goals of semi-automated requirements dependencies detection: 1) finding all security-related dependencies, because missing any dependency will cause incomplete/insufficient security testing; 2) reducing assessment costs. Therefore, two types of measures are used to evaluate experimental results: cost and performance. The experimental results are summarized in TABLE 5.2. The number of assessments \(|A|\) is used to evaluate approaches’ costs. The reduction means the percentage of assessments are saved. Three measures (recall \(R\), precision \(P\), and \(F_2\)) are applied to evaluate the performance of different approaches:

\[
R = \frac{|\text{identified dependencies} \cap \text{true dependencies}|}{|\text{true dependencies}|} \quad (5.6)
\]

\[
P = \frac{|\text{identified dependencies} \cap \text{true dependencies}|}{|\text{identified dependencies}|} \quad (5.7)
\]

\[
F_2 = \frac{5 \cdot R \cdot P}{R + 4 \cdot P} \quad (5.8)
\]

TABLE 5.2 shows average results of 10-round evaluations. Compared to LS approach, student’s t-test results (i.e., average \(p\)-value of student’s t-test on “Recall” and “Precision” are 0.016 and 0.044 respectively) suggested that our three-step approach significantly improves both recall and precision of the requirements dependency identification without increasing the assessment cost (i.e., average \(p\)-value of student’s t-test on “Reduction” is 0.18). The reason we apply student’s t-test in this analysis is that Shapiro-Wilk test suggests that both “Recall” and
“Precision” results of 10-round evaluations follow the normal distribution.

5.3 Vulnerability Testing and Validation

For each dependency detected in Section 5.2, we use all XSS attacks designed in a well defined XSS attack cheat sheet [57] as test inputs to detect XSS vulnerabilities in all projects. The number of vulnerability candidates we detected in each project is reported in the last column of Table 5.2. Figure 5.4 shows a sample vulnerability detected in Scholar@UC in which a user submits a new work and names it with a string of XSS code (the requirement “submit work”\(^6\), Figure 5.4 a). Then a window (Figure 5.4 c) is unexpectedly popped out when another user tries to view citations of this work by clicking the “Citations” button (the requirement “export citation”\(^7\).

\(^6\)https://trello.com/c/LHaTnSnF/
\(^7\)https://github.com/uclibs/scholar_use_cases/blob/master/display_download/display_download_use_cases.md
To validate vulnerability candidates, we contacted the Scholar@UC developer team. A core developer met with us. After one hour discussion, he confirmed that all eight candidates were new vulnerabilities for them. These vulnerabilities were reported in an issue [93]. In this section, we reported insights we learned during the process that is instructive not only for the vulnerability mitigation but also for improving the requirements dependency detection.

5.3.1 Improving Dependency Detection

The question the developer keeps asking during the meeting is that under which scenario this vulnerability will harm other users. The question tries to find the victim of the vulnerability. Attackers can inject attacks in any features of the web application. Predicting who is the victim can potentially narrow down the search space of requirements dependency. In addition, adding a new feature “user role” to our FNN model allows it to learn semantic relations between requirements like patient “schedule an appointment (UC22)” has impact on the HCP when he tries to “view scheduled calendar (UC40)” in iTrust. By doing this, the accuracy of the dependency detection approach can be potentially improved. In fact, a new vulnerability is detected based on this dependency.

A surprising response from the developer happened when we report vulnerability in the requirements dependency “work’s representing image” → “preview image in JEPG2000 viewer” [10]. In this vulnerability, the JEPG2000 viewer reads and

---

8 https://152.46.18.254/doku.php?id=requirements:uc22
9 https://152.46.18.254/doku.php?id=requirements:uc40
10 https://trello.com/c/tbndltJM/
11 https://github.com/uclibs/scholar_use_cases/blob/master/display_download/display_download_use_cases.md
executes the malicious code saved in the title of the image. After confirming this vulnerability, the developer predicted another vulnerable requirements dependency: “submit work”→“preview image in JEPG2000 viewer”. In this dependency, attacks saved in work’s title also be read and executed by the JEPG200 viewer. This “hidden” dependency can be easily detected by a human analyst who has contextual information of these two requirements. A potential solution to allow machine to have this type of intelligence is putting the context (the requirement “work’s representing image”) into consideration when detecting the dependency.

Putting more requirements into consideration not only helps detect more requirements dependency, but also increases the probability of finding vulnerabilities. For instance, one dependency in Scholar@UC (i.e., submit work→add work’s editor\textsuperscript{12}) was detected by our FNN model but no vulnerability was detected. In fact, the attack was successfully injected into the work’s title when the attacker submits the work and we did not observe it in both “submit work” and “add work’s editor” requirements. However, if we push a step forward (i.e., test another requirement “edit the work”\textsuperscript{13}), the vulnerability (vulnerability 2 in the GitHub issue [93]) will be observed when the newly added editor deletes the work. Requirements dependencies that involve more than two requirements (e.g., “submit work”→“add work’s editor”→“edit the work”) increase the success rate of detecting vulnerabilities. Extending our FNN model to detect such dependency is valuable future work.

\textsuperscript{12}https://trello.com/c/5LHWk5d1/
\textsuperscript{13}https://trello.com/c/53pFg90e
5.3.2 Vulnerability Mitigation

According to the GitHub issue [93], all vulnerabilities that we detected in Scholar@UC have been mitigated except vulnerability 4 (i.e., the vulnerability in “work’s representing image” → “preview image in JPEG2000 viewer” that was discussed in the second paragraph of Section 5.3.1). Unlike other vulnerabilities, vulnerability 4 is not caused by Scholar@UC but the vulnerable third-party software Universal Viewer\(^\text{14}\). A separate issue [94] was created to address this vulnerability.

Two alternative mitigations were proposed in the GitHub issue [94]: (1) Address it locally by sanitizing the information sent to Universal Viewer; and (2) updating Universal Viewer to an invulnerable version. Obviously, the second method can mitigate this vulnerability more thoroughly because mitigating the vulnerability in the Universal Viewer benefits not only Scholar@UC, but also other systems that depend on the Universal Viewer. In fact, developers tried the second mitigation first.

However, according to a related issue in the Universal Viewer [95], developers tried to mitigate the vulnerability by using js-xss\(^\text{15}\). A long-term disadvantage here is that, just like the Universal Viewer, any vulnerabilities detected in js-xss in the future will negatively impact other systems that depend on it (e.g., the Universal Viewer) as well as other products on the downstream of the project dependency chain (e.g., Scholar@UC). Alqahtani et al. [96] tried to build the dependency chain between different projects and track the vulnerabilities detected in all nodes of the chain. Since each node in the dependency chain represents a project, results are not friendly to security testers because the information of how those software projects

---

\(^{14}\)https://github.com/UniversalViewer/universalviewer

\(^{15}\)js-xss is an open source software used to sanitize untrusted HTML. Website: https://jsxss.com/en/index.html.
interacted to each other is missing. Extending our study to detecting dependency between requirements in different projects is a promising solution to address this difficulty.

For other vulnerabilities, among different mitigation solutions (e.g., encoding and encryption), developers choose sanitizing untrusted user inputs. Instead of sanitizing user inputs after the system received it, developers label all user inputs as “sanitize” and modify them when they are displayed to the user. One advantage of this solution is that it can prevent executing malicious code when the user inputs do not reach the sanitization function in the back-end. For instance, in iTrust, an attacker tries to inject malicious code “</TITLE><SCRIPT>alert(‘XSS’)</SCRIPT>” into the textbox “First name” of the medical records release request form16. The system returns the error information (Figure 5.5) because the user input (i.e., first name) does not pass the formatting checking. It also means that the user input does not reach the sanitization function. Therefore, an unexpected pop-up window shows the vulnerability here when the system reloads the page and fill the “First name” textbox with the original user inputs. The reason of this vulnerability is that the web browser treats the first “>” in the attack as the ending symbol of the textbox and executes the malicious code saved in the later part of the attack input. However, this vulnerability can be prevented by labeling the user input as “sanitize” before passing it to format checking and replacing character “>” with the HTML code “&gt;” when the web browser reloads it.

16https://152.46.18.254/doku.php?id=requirements:uc56
5.4 Discussion

In this section, we first discuss the advantages and limitations of our approach in Section 5.4.1, and then discuss threats to validity in the second subsection (Section 5.4.2).

5.4.1 Advantages and Limitations

Felderer et al. [52] suggest that six aspects shall be taken into consideration when evaluating and selecting a specific security testing method. In this section, we follow these six aspects to discuss the advantages and disadvantages of our security testing approach.

5.4.1.1 Attack Surface

Different security testing methods find different attack and vulnerability types. In this paper, we focus on the XSS vulnerability. However, the mechanism of other user input sensitive vulnerabilities like structured query language injection (SQLI) is the same as XSS [27, 30, 32]. Therefore, our approach can be applied to
detect those vulnerabilities. In addition, one important results of our approach is a set of requirements dependencies. As previously stated, requirements dependency provides directed benefit for many software engineering tasks. Different domains focus on different non-functional requirements. For example, elevator systems prefer safety requirements. Currently, our experiments are built on security regulations. More experiments shall be conducted to test the generalizability of our approach.

5.4.1.2 Application Type

Different security testing methods perform differently when applied to different types of applications. Subject projects used in our experiments are web applications. Since our approach depends on textual requirements only, it does not matter whether the targeted application is a mobile app or a system of systems. In other words, our approach is application independent. However, we shall point out that our approach focuses on vulnerabilities caused by requirements dependency. Combining our approach with other vulnerability detecting technologies such as static analysis [28] and dynamic testing [33] which focus on individual requirements or source code level can increase the confidence in system security.

5.4.1.3 Performance and Resource Utilization

Different methods require different computing power and different manual efforts. We address this issue by providing automated support for two time-consuming sub-tasks in our approach: the requirements tracing and low-level security requirements dependency validation. The experiments results showed that our approach achieves high performance in terms of detecting more requirements dependencies without increasing assessment costs. We shall point out that our automated low-level
security requirements dependency validation took about two hours to finish all 199,528 assessments in Scholar@UC. Comparing to applying the validation of all 973,710 requirements pairs, it saved almost 10 hours.

5.4.1.4 Maintainability

Design method for the large enterprise is usually not a one-time effort [52]. In order to improve the efficiency of the method, results generated by testing methods are likely to be stored and reused in continuous testing. Two results of our approach will be stored and require maintenance. The first one is high-level security requirements dependencies. Stability of the regulations indicates that little maintenance is needed for this part. Another one is low-level security requirements dependencies. The maintenance is required when new requirements or changes on existing requirements are proposed. A light-weighted and agile way to maintain low-level security requirements dependencies is that tracing the new or changed requirement to its HSR \( R \), and only assessing dependencies between this new requirement and other LSRs under interacting HSRs of \( R \).

5.4.1.5 Quality of Results

The first result of our approach is a set of requirements dependencies. Average \( p \)-value of student’s t-test on recall values in Table 5.2 demonstrated that more requirements dependencies are detected by our approach. However, the requirements dependency detection is the means. Our ultimate goal is finding more security vulnerabilities before the software is released. Experiments on five projects showed that our approach can find more vulnerable requirements dependencies. The survey with the original developer of Scholar@UC demonstrated that all candidates detected
by our approach are validated vulnerabilities. In other words, the results of our approach are highly accurate.

5.4.1.6 Supported Technologies

The same method usually only supports a limited number of technologies such as specific programming languages [52]. If one method supports multiple technologies, it does not necessarily support all of them with the same quality. The previous discussion shows that our approach depends on requirements description only and is independent with source code. Therefore, combining with different test inputs generation approaches proposed in the previous studies [30, 97], our approach can be extended to any software developed with different technologies, such as Android applications developed in Java and Apple apps developed in Swift 4.

5.4.2 Threats to Validity

Several factors can affect the validity of our study. Construct validity is the degree to which the variables accurately measure the concepts they purport to measure [98]. In our experiment, there were minimal internal threats to construct validity as standard measures (recall, precision, and $F_2$), which have been widely used in machine learning approach studies [99], were employed to assess the different approaches. These measures were also complemented by another measure (assessment cost reduction) that is used to provide more insights into the results, in particular, the efficiency of different approaches. Therefore, we believe that measures we used sufficiently capture and quantify the different aspects of requirements dependency identification approaches evaluated in this paper.

Threats to external validity impact the generalizability of results. In particular,
the results of this study might not generalize beyond the underlying experimental settings [98]. One major threat to the external validity comes from the datasets used in this experiment. We mitigate this threat by testing our approach with five projects from two different application domains. Additionally, the sizes of requirements in different projects used in the experimental are varied from hundreds to thousands. Therefore scalability questions are addressed. Another threat comes from training datasets (i.e., HSRs to LSRs trace links for automated requirements traceability in Section 5.1.2 and dependencies between LSRs for semantic analysis in Section 5.1.3). We acknowledge that manually creating training datasets is an error-prone task. However, we believe having two analysts worked jointly helps mitigate this threat. In fact, inter-rater agreement analysis (Fleiss’ kappa [83]) also demonstrated that two analysts provided coherent judgments.

In addition, specific design decisions and heuristics used during the implementation can also limit the results applicability. Such decisions include the thresholds of indicator terms, feedback weights, the size of input and hidden layers, and using ReLU as activation functions.

5.5 Summary

In this chapter, we not only discuss the importance of requirements dependency to security test paths generation but also propose a semi-automated approach to detect security requirements dependencies. We applied our approach to Scholar@UC, an OSS project developed by the University of Cincinnati. The original developers confirm that all four vulnerabilities detected our requirements dependency detection approach are valid. In the next chapter, I will conclude this thesis and discuss
future work.
Chapter 6

Conclusions and Future Work

In this thesis, we proposed a security requirements management framework that aims to help OSS projects developers detect, trace, and test security requirements. We also designed automated tools for three main tasks in our framework (i.e., security requirements identification, security requirements tracing, and security requirements dependency detection).

For the first part (i.e., security requirements identification) of our framework, we analyze six metrics that can help distinguish security requirements from others. Six models are built to extract these six metrics in requirements. Then, we construct a linear classifier to combine the six models together. We also apply our linear classifier on three long-lived OSS projects. The results demonstrated that, compare to single models, our linear classifier achieves highest recall and precision.

With security requirements that are identified by the automated approach proposed in the first part, the natural next step is to validate and verify security requirements. However, the-state-of-the-art approach for automated security requirements tracing suffer the low-accuracy problem. Low recall rate indicates that
important security requirements trace links are missing. However, we cannot afford to miss any trace links since it could lead to incomplete validation and verification, thus leading to missing security vulnerabilities. At the same time, the low precision rate requires spending more manual effort on judging security requirements, thus making the result of automated tracing useless. From our analyzing on two OSS projects, we find that certain terms in security requirements and related software documents play an important role in automated tracing.

Therefore, for the second part of our framework, we propose a term-based relevant feedback approach with the hope it can significantly improve the accuracy of the automated tracing. We apply it to two OSS projects, the results show that, compare to other the-state-of-the-art approaches, the term-based relevant feedback approach is more suitable for security requirements tracing. Another contribution of this part is that we designed two new metrics to evaluated the effectiveness and efficiency of the automated tracing approach.

With security requirements and the trace links to the source code, V&V and testing approaches can be applied to ensure that there is no security vulnerabilities in each requirement. There are many well-defined methods and sophisticated tools to support those tasks. However, recently reported vulnerabilities demonstrate that there are many vulnerabilities are caused by requirements interactions or dependencies. Developers of OSS projects demand automated tools to help them detect such critical dependencies between security requirements. In fact, with those dependencies, better test cases can be designed to detect vulnerabilities.

In order to increase efficiency, we propose a semi-automated approach to detect security requirements dependencies. We also conduct an experiment on five OSS projects to evaluate our approach. The results show that our approach can detect
more dependencies with less cost. In addition, we systematically test dependencies in an OSS project Scholar@UC that are detected by our approach. Four vulnerabilities are found and reported to original developers of Scholar@UC. They confirm that all vulnerabilities are valid and create an issue in their issue track system to mitigate those vulnerabilities. Insights from the vulnerability validation and mitigation are reported in Chapter 5.

For each component of our framework, we also identify key limitations. We discuss those limitations and related future work in the following paragraphs.

For the security requirements identification, there are two main limitations: First, as we discussed in Chapter 3, metrics used to measure requirements complexity may also relate to other non-functional requirements. In future research, we need to find other complexity metrics with the hope that they can improve the precision without decrease the recall. Second, feature selection is conducted base on the student’s $t$-test only. Advanced feature selection and analysis approach like minimum-redundancy-maximum-relevance and quadratic programming could help avoid the curse of dimensionality as well as enhance generalization of our models. Finally, in this study, we only tested the linear combination of feature values. The advantages including simplicity and time efficient. However, compared to the non-linear classifier, such as KNN, its disadvantage is also obvious. That is the accuracy of it relatively lower. Based on our previous discussion, we propose following future work: 1) systematically select features (including adding new features to build our models;) 2) compare linear classifier with non-linear classifiers to find the most optimal way to combine all feature values; and 3) integrating our models to industry common tools, such as JIRA, to provide real-time support.

In the security requirements tracing, the main limitation is our manual con-
struction of the domain-specific dependability taxonomy for specificity evaluation. Even ontology engineering has dramatically advanced in the past several decades, fully automated ways to build the knowledge base for dependability hardly exist. Researchers may be eager to advocate their exciting ontology building techniques, we argue that certain level of manual intervention, like quality control or consistency management, is unavoidable. In this sense, the taxonomies of built by our analysts should be regarded only as starting points and are subject to subsequent refinement and maintenance. Future work for this part includes assessing the usability of our method, conducting more empirical evaluations with different types of industrial systems, covering other types of dependability concerns like fault tolerance and autonomic healing, expanding the tracing to other nonfunctional requirements such as maintainability and reliability, and incorporating ontology engineering to refine and reuse the knowledge representation and reasoning of dependability.

There are two limitations in the security requirements dependency detection component: First, manually analyzing dependencies between HSRs. This not only requires human effort when applying our approach to a new domain but also leads to manually maintaining HSR dependencies while there is any requirements change in HSRs. Second, only five features used to build FNN model. This may harm the accuracy of our approach. More dependency specific features shall be tested and used to build new FNN model. Finally, our research focuses on dependencies between two requirements. However, there are many dependencies that involve more than two requirements. Thus, the future work for this part includes: 1) applying our approach to other industrial projects from different domains or focusing on different non-functional requirements; 2) conducting new experiments to test the generalizability of our approach to detecting other types of vulnerabilities like SQLI;
3) applying systematic feature selection for FNN model; and 4) extending our work to detecting vulnerabilities in requirements dependencies that involve more than two requirements and dependencies between requirements from different projects.
Bibliography


[34] W. N. Robinson, S. D. Pawlowski, and V. Volkov, “Requirements interaction


[40] J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Emenecker, “A machine learning approach for tracing regulatory codes to product specific requirements,”


[95] Esmé Cowles, “XSS vulnerability: executing code in HTML tags


