Detecting Software Security Vulnerabilities Via Requirements Dependency Analysis

Wentao Wang, *Member, IEEE*, Faryn Dumont, Nan Niu, *Senior Member, IEEE*, and Glen Horton

Abstract—Cyber attacks targeting software applications have a tremendous impact on our daily life. For example, attackers have utilized vulnerabilities of web applications to steal and gain unauthorized use of sensitive data stored in these systems. Previous studies indicate that security testing is highly precise, and therefore is widely applied to validate individual security requirements. However, dependencies between security requirements may cause additional vulnerabilities. Manual dependency detection faces scalability challenges, e.g., a previous study shows that the pairwise dependency analysis of 40 requirements would take around 12 hours. In this article, we present a novel approach which integrates the interdependency among high-level security requirements, such as those documented in policies, regulations, and standards. We then use automated requirements tracing methods to identify product-level security requirements and their dependencies. Our manual analysis of HIPAA and FIPS 200 leads to the identification of five types of high-level security requirements dependencies, which further inform the automated tracing methods and guide the designs of system-level security tests. Experimental results on five projects in healthcare and education domains show the significant recall improvements at 81 percent. Our case study on a deployed production system uncovers four previously unknown vulnerabilities by using the detected requirements dependencies as test paths, demonstrating our approach's value in connecting requirements engineering with security testing.

Index Terms—Security requirements, requirements dependency management, requirements traceability, vulnerability discovery

1 Introduction

The number of vulnerabilities reported to the Common Vulnerabilities and Exposures (CVE)[1] continues to grow every year [1]. This number reached a new record of 16,511 and 17,307 in 2018 and 2019 respectively. A single vulnerability could negatively impact tens of thousands of end users. For instance, in 2014, a cross-site scripting (XSS) vulnerability reported in eBay caused a data breach that compromised nearly 145 million customers' usernames and passwords [2]. Another painful instance happened in 2018: a data breach caused by security vulnerability exposed more than 50 million Facebook user accounts to malicious attackers [3].

Static analysis [4], [5], [6] and security testing [7], [8] 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 [7]. In contrast, security testing is highly precise [9]. Several testing approaches such as dynamic taint analysis [7] and penetration testing [10] 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 the importance of *requirements dependency analysis* aimed at the discovery and management of critical relationships among sets of requirements [11]. According to Robinson *et al.* [11], up to 70 percent of total software errors are caused by interacting requirements, making the requirements dependency error a significant software development and quality challenge. In fact, the Facebook vulnerability mentioned earlier was caused by two requirements, namely "view as" and "upload birthday video" [3]. Although "upload birthday video" was introduced in 2017, its interdependency with other requirements (especially with "view as") was not thoroughly tested, resulting in the serious security breach.

Manually analyzing requirements dependency suffers from a prohibitively high cost. Carlshamre *et al.* [12] studied interdependencies within five distinct sets of requirements from industrial projects. They showed that pairwise dependency analysis of only 40 requirements would take in the vicinity of 12 hours. In modern software projects, the size of requirements ranges from hundreds to thousands [13], making manual pairwise dependency analysis a mission impossible.

Carlshamre *et al.* [12] pointed out that empirically the 20 percent "most dependent" requirements were responsible for 75 percent of all the dependencies. Therefore, pairwise assessments could be done only for those 20 percent requirements in order to reduce the effort needed for identifying the dependencies. This is a reasonable trade-off in tasks like release planning [12]. However, missing any critical interdependency may lead to security vulnerabilities in

1. http://cve.mitre.org/

- *Wentao Wang, Faryn Dumont, and Nan Niu are with the Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati, OH 45221 USA. E-mail: {wang2wt, dumontfn}@mail.uc.edu, nan.niu@uc.edu.*
- *Glen Horton is with University of Cincinnati Libraries, University of Cincinnati, Cincinnati, OH 45221 USA. E-mail: glen.horton@uc.edu.*

Manuscript received 13 July 2019; revised 9 Aug. 2020; accepted 6 Oct. 2020. Date of publication 13 Oct. 2020; date of current version 16 May 2022. (Corresponding author: Nan Niu.)
Recommended for acceptance by Z. Jin.
Digital Object Identifier no. 10.1109/TSE.2020.3030745

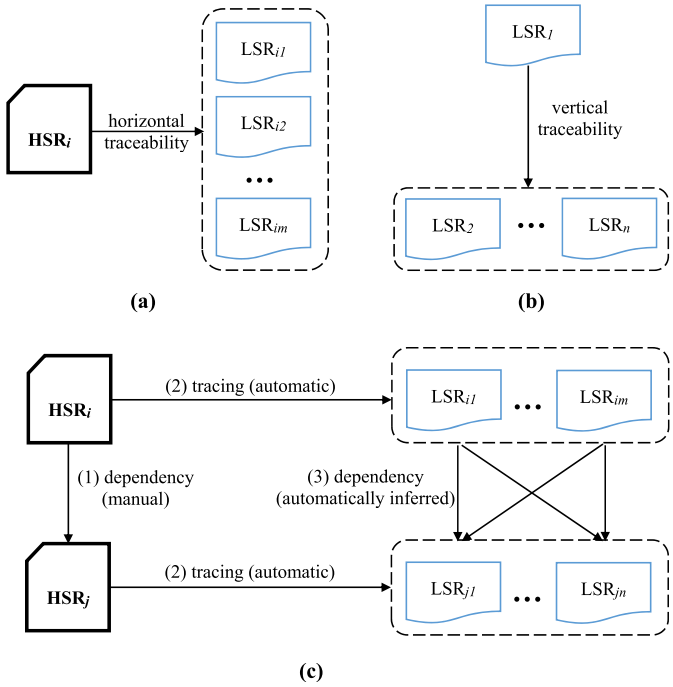


Fig. 1. **(a)** Horizontal traceability concerns individual HSRs (high-level security requirements found in regulations or policies), **(b)** Vertical traceability concerns dependencies among LSRs (low-level security requirements found in concrete software), and **(c)** Our semi-automated approach, guided by dependencies among HSRs, incorporates both horizontal and vertical traceability.

the software product. Given that a single vulnerability, like the XSS vulnerability in eBay [2], could negatively effect millions of users, having an automated tool is key to achieving a high recall especially when security requirements are involved.

A class of solutions striving for high recalls is automated requirements traceability based on information retrieval algorithms [14]. Hayes and her colleagues [15], [16] were among the first to automatically link high-level and low-level requirements, and their experiment on the CM-1 dataset showed that the TF-IDF algorithm achieved an acceptable recall with a good precision [16]. Of particular relevance to security is the work by Cleland-Huang *et al.* [17] where high-level security requirements (HSRs) like the regulatory codes of the Health Insurance Portability and Accountability Act (HIPAA)[2] are traced to low-level security requirements (LSRs) such as the features of specific software systems.

Linking HSRs and LSRs, shown in Fig. 1a, is a kind of *horizontal traceability* where relationships between artifacts that are part of different work products are established [18]. In contrast, Fig. 1b illustrates *vertical traceability* that captures dependencies among artifacts that are part of a single work product within the software development process [18]. The state-of-the-art of horizontal traceability [17] concerns individual requirements without their interdependencies, whereas vertical traceability [18] focuses on product-level requirements dependencies without considering the high-level guidelines regulated by the security policies.

In this paper, we propose a novel approach to integrate horizontal and vertical traceability in analyzing security requirements dependencies. We distinguish two levels of security requirements: HSRs are those specified in regulations and policies whereas LSRs are those implemented in concrete software systems and products. As shown in Fig. 1c, our hybrid approach first identifies the dependencies among the HSRs manually, and then traces HSRs to LSRs automatically. The combination of horizontal and vertical traceability gives rise to requirements dependencies at the product level, and our key insight here is that dependencies between LSRs shall be consistent with dependencies between their corresponding HSRs.

Our research methodology involves three steps shown in Fig. 1c. We first perform manual analysis to identify the dependencies of HSRs found in regulations and policies. An important result of the HSR dependency analysis is the set of indicator terms which we use in the second step to automatically trace HSRs and LSRs. As the third step, the dependencies of LSRs are automatically derived, i.e., if HSR$_i$ depends on HSR$_j$, then all LSR_{im} traced to HSR$_i$ depend on all LSR_{jn} traced to HSR$_j$. This last step establishes requirements dependencies at the concrete software system's level. We carry out two evaluations to assess our approach: a quantitative experiment on five open-source projects to evaluate the requirements dependency detection, and a qualitative case study on a deployed, production system to use the detected requirements dependencies to derive system-level security tests in order to uncover software vulnerabilities.

Our approach presented in this paper makes three main contributions:

1) We present five HSR-dependency types—namely, input modification, temporal relation, task refinement, triggering condition, and realization similarity—while manually analyzing the regulatory codes in HIPAA and Family Educational Rights and Privacy Act (FERPA);[3]
2) We evaluate the performance of our requirements dependency identification in five long-lived software projects, demonstrating the significant recall improvements at 81 percent as well as the reusability of the manual analysis; and
3) We conduct a case study with a deployed system to derive system-level tests based on the interdependent LSRs, and illustrate the effectiveness of our approach in detecting four previously unknown security vulnerabilities.

The remainder of this paper is organized as follows. We review related work in Section 2. In Section 3, we introduce our semi-automated requirements dependency analysis approach. In Section 4, we evaluate the requirements dependencies quantitatively on five projects. Section 5 describes our case study on software vulnerability detection. We discuss limitations and threats to validity in Section 6, and conclude with the summary and future work in Section 7.

2. HIPAA requires health agencies in the United States to use technical safeguards to protect patient medical information. See https://www.hhs.gov/hipaa/index.html

3. FERPA forces all educational agencies like colleges in the United States to protect student education records. See https://www2.ed.gov/policy/gen/guid/fpco/ferpa/index.html

2 Background and Related Work

2.1 Software Vulnerability Detection

Vulnerabilities in our work refer to the implementation bugs hurting a software system's security goals, such as protecting the confidentiality, preserving the integrity, and ensuring the availability of the information assets. Many types of vulnerability are documented in the CVE, including XSS (CWE-79)[4], SQL injection (CWE-89), path manipulation (CWE-22), nonexistent access control (CWE-285), lack of auditing (CWE-778), trust boundary violation (CWE-501), dangerous file upload (CWE-434), etc.

Software development teams usually apply multiple approaches to help prevent vulnerabilities. The approaches toward vulnerability detection can be classified into two categories: static analysis and security testing. Static analysis looks into the source code of the system under test (SUT) without actually executing it and reports potential vulnerabilities. Automated static analysis approaches use different software metrics related to source code complexity, like lines of code, coupling, and cohesion, to predict vulnerabilities [5], [19]. Most approaches predict software vulnerabilities in general. With appropriate adaptions or configurations, they can also be applied to predict specific vulnerabilities which are unique to the SUT (e.g., XSS vulnerability in web applications). However, a previous study [19] 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 result evaluation, while the second one leads to the situation that security experts need to build new prediction model for each new project which is neither practical nor realistic.

Security testing is complementary to static analysis by executing the SUT on the real or virtual environment. These include taint analysis and vulnerability scanning. In taint analysis [7], untrusted user data is labeled as "tainted" at runtime, which is cleared only if the data passes a dedicated sanitization function. If the data which still carries the taint information reaches a security sensitive sink (e.g., a webpage that displays the data), the system is considered as vulnerable. Taint analysis requires the source code which is not always available in security testing. In addition, extra engineering effort (e.g., adding new database columns for tracking user input [7]) is required to implement taint analysis.

Another approach which requires less engineering effort is vulnerability scanning. Automated scanners like ZAP [20] and SecuBat [21] are widely used to detect vulnerabilities (especially 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 [22]) and analyze immediate responses of the system for indicators of if the attack is successful. However, successful attacks do not always manifest themselves in the immediate responses. For instance, in the eBay case [2], it was hard to tell whether the attackers were successful right after the malicious XSS code was saved into the system. The judgment could not be made until the malicious links were displayed to victims. This limitation can be addressed by systematic requirements dependency detection.

4. The Common Weakness Enumeration (CWE) is a community developed dictionary of software weakness types. See http://cwe.mitre.org

2.2 Requirements Dependency and Traceability

Carlshamre *et al.* [12] investigated requirements in five industrial projects and showed that it took between 2.5 and 3 hours to detect pairwise dependencies of 20 requirements (190 assessments). With 40 requirements, for instance, the pairwise dependency analysis would take in the vicinity of 12 hours, signifying clear scalability difficulties. To reduce the number of assessments, Carlshamre *et al.* [12] proposed a couple of methods: 1) identifying and removing singular requirements, 2) focusing on four types (i.e., migration to a new platform, changes to core functionality, changes to core data structures, and major changes to user interfaces) of highly dependent requirements and performing pairwise assessments only on them. Experiments indicated that the second method was the most effective in effort reduction. The results showed that the four types of "most dependent" requirements only account for 20 percent of all requirements. However, assessing them could cover roughly 75 percent of all the dependencies. For vulnerability detection, uncovering all the requirements dependencies while keeping a low false positive rate (i.e., high recall and acceptable precision) is the ultimate goal.

Automated requirements traceability strives for high recall when recovering the links of various software artifacts [14], [15], [16]. According to Rempel and Mäder [18], the dependency between two requirements belonging to a single work product represents a relationship characterized by vertical traceability illustrated in Fig. 1b. To support automated vertical traceability, Carlshamre *et al.* [12] and their previous research [23] propose that if the lexical similarity score (i.e., cosine coefficient) of two requirements is greater than 0.125, there is high possibility that interdependence exists between them. However, vertical traceability does not distinguish whether the interdependent requirements are security requirements or not.

Horizontal traceability, shown in Fig. 1a, concerns the relationship between artifacts that are part of different work products [18]. When HSRs such as the HIPAA regulatory codes are traced, the resulting horizontal traceability links are product-level security requirements that comply with the regulations. Cleland-Huang *et al.* [17] showed the effectiveness of probabilistic network models in automatically tracing HIPAA regulations. The maximum recall and the average precision in their experiment of ten systems are 79 and 36 percent respectively [17]. Using horizontal traceability, although regulatory compliance of every single HSR can be demonstrated, the requirements dependencies are not considered. Our work is motivated by leveraging both the security focus of horizontal traceability and the dependency relation of vertical traceability in order to systematically detect vulnerabilities in software systems and products. We detail our approach in the next section.

3 Requirements Dependency Analysis

Fig. 2 overviews our approach which consists of three steps: (1) manually identify and classify the dependencies among HSRs like the regulatory codes in HIPAA, (2) automatically

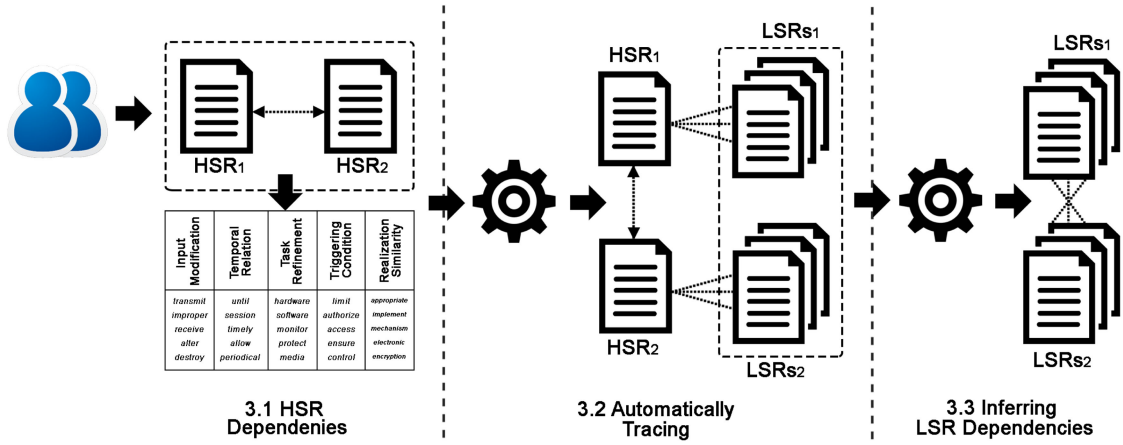


Fig. 2. Three main steps of our approach: Section 3.1 describes our manual analysis of dependencies among the HSRs, resulting in five dependency categories and several indicator terms in each category, Section 3.2 introduces the use of the indicator terms as relevance feedback in automated requirements traceability so as to identify the LSRs that trace to the HSRs, and Section 3.3 presents the automated inference of LSR-dependency based on the previous two steps.

TABLE 1
Twelve HIPAA Regulatory Codes (Adopted From [17]) and 29 FIPS 200 Security Requirements in 13 Categories

Name and ID of HIPAA regulatory codes		Category name (ID) of FIPS 200 security requirements	
Access Control (AC_H)	Integrity Controls (IC)	Access Control (AC_F)	Incident Response (IR-I, IR-II)
Automatic Logoff (AL)	Person or Entity Authen -tication (PA)	Awareness and Training (AT-I, AT-II)	Media Protection (MP-I, MP-II, MP-III)
			Personnel Security (PS-I, PS-II, PS-III)
Mechanism to Authenticate Electronic Protected Health Information (APHI)	Audit Controls (AUD)	Audit and Accountability (AU-I, AU-II)	Configuration Management (CM-I, CM-II)
	Integrity (IC)		Identification & Authentication (IA)
	Encryption (TED)	Risk Assessment (RA)	System & Services Acquisition (SA-II, SA-III, SA-IV)
Emergency Access Procedure (EAP)	Encryption and Decryption (SED)	System & Communication Protection (SC-I, SC-II)	Certification, Accreditation, and Security Assessments (CA-I, CA-II, CA-III, CA-IV)
Transmission Security (TS)	Unique User Identification (UUI)	System & Information Integrity (SI-I, SI-II, SI-III)	

trace an individual HSR to LSRs, and (3) automatically infer the dependencies among the LSRs based on the previous two steps. The novelty of our approach lies in not only the combination of vertical (first step and third step) and horizontal (second step) traceability, but also the new set of security-related types of requirements dependencies grounded in our manual analysis. This set extends current literature [11], [12], and offers two benefits: guiding the selection of appropriate trace retrieval algorithms and directing the generation of test cases to uncover software vulnerabilities.

3.1 High-Level Security Requirements (HSR) Dependencies

Despite challenges introduced by the ever complex and interconnected software applications, regulations like HIPAA provide guidance on information security. Detecting dependencies between HSRs defined in regulations could benefit not only security testing but also other software engineering tasks, such as compliance checking and certification. The completeness of regulations ensures that dependencies between HSRs have a broad coverage of the real-world scenarios. The relatively small size of HSRs in regulations reduces the effort on dependency assessment. The stability of regulations reduces the effort spent on dependency 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 in many systems and products of the given domain.

In this research, we perform manual dependency analysis of the security requirements defined in HIPAA and FERPA. For HIPAA, a set of 12 security requirements is studied in [17]. For FERPA, the United States Department of Education lists the standard of "Minimum Security Requirements for Federal Information and Information Systems" (FIPS 200) [24] 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 [25]. The 29 FIPS 200 HSRs, along with the 12 HIPAA HSRs, are summarized in Table 1. The full descriptions and dependencies of HSRs can be found in [26].

Two students, each having at least one-year research experience on security requirements and related topics, spent approximately 24 hours individually to detect HSR dependencies. Cohen's kappa between their results was 0.63, indicating a "good" inter-rater agreement level. The discrepancies were resolved in a two-hour meeting in which the third researcher participated. Jointly, the researchers reached consensus, reviewed all the dependency detection results, and categorized the HSR dependencies into five categories. These categories represent the first contribution of our work, which we explain by using representative examples.

1) Input Modification. If HSR_1 creates, modifies, or deletes an element or attribute of an entity that is regulated by another requirement HSR_2, then HSR_1 depends on HSR_2 which we denote as $HSR_1 \rightarrow HSR_2$. An example is AU-I→SC-I in FIPS 200, where AU-I creates information system audit

TABLE 2
Type of Requirements Dependencies in the Literature With Examples or Explanations, and our HSR-Dependency Analysis Results

Release Planning [12]	Interaction Management [11]	High-Level Security (our approach)	# of dependencies HIPAA	FIPS 200
Requires ("printer" requires "driver")	Structure (Req_1 is similar to Req_2)	Input Modification	5	11
Temporal ("add" is done before "delete")	Time (Req_1 is temporally related to Req_2)	Temporal Relation	2	3
Cvalue ("manual" decreases value)	Task (Req_1 describes a task for Req_2)	Task Refinement	2	7
Icost ("waiting" increases cost)	Causality (Req_1 is a consequence of Req_2)	Triggering Condition	4	4
Or ("draw" or "import" an image)	Resource (Req_1 & Req_2 rely on a resource)	Realization Similarity	3	2

records and SC-I monitors information transmitted or received by the information system. Now, assume a system-level feature LSR_1 satisfies AU-I by recording audit information in a log file but does not follow the format specified in the feature LSR_2 which corresponds to SC-I. Then, a potential security risk is that LSR_2 would miss attacks (e.g., tampering) or illegal communications due to logs not matching the required format. Thus the dependency, $LSR_1 \rightarrow LSR_2$, shall be thoroughly checked and such checks are driven naturally by LSR_1's creation, modification, and deletion actions.

2) Temporal Relation. If there is a clear temporal relationship between HSR_1 and HSR_2, such as "Automatic Logoff" and "Access Control" in HIPAA, we mark AL→AC_H. Only after a user logs into the system by passing the access control does it become necessary for the system to automatically log out the user when a predetermined time period of inactivity is reached. A security issue may occur on the day we begin the daylight savings. Suppose AC_H records that a user logs into the system at 1:59am on March 11. This user might be forcefully logged out after only one minute because AL may think that she has been inactive for one hour (i.e., the system time is automatically adjusted to 3 am). This error harms availability, one of the key aspects of security. Systematic tests can therefore be derived by considering the temporal scenarios.

3) Task Refinement. If requirement HSR_1 describes the general goal, and another one HSR_2 specifies a detailed approach toward the goal, we then note $HSR_1 \rightarrow HSR_2$. 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→MP-II and MP-I→MP-III. Imagine that a system adds a new type of media (e.g., flash files) as well as a new requirement LSR_1 to satisfy MP-I with respect to the new media files. Suppose LSR_2 provides sanitizing features complying with MP-III. If the dependency $LSR_1 \rightarrow LSR_2$ fails to be recognized (e.g., automatically scanning and removing sensitive information do not happen to flash files), then security breach may occur.

4) Triggering Condition. If HSR_1 has to react to the change while requirement HSR_2 is taking place, the dependency $HSR_1 \rightarrow HSR_2$ is recorded. For example, in FIPS 200, after AC_F ("access control") rejects login request from the same IP address n times (where n is defined by the security policy) 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 punish this IP address by adding it to the blacklist. Therefore PS-III→AC_F records a triggering condition. A potential security risk will happen when AC_F handles login requests in parallel, changing the condition. An attacker may send two requests from one IP address at the same time, and AC_F may record both rejections for that IP address in one log item, resulting in PS-III treat the two failed requests as one rejection. The vulnerability caused by neglecting the PS-III→AC_F dependency would allow attackers to try more combinations before being blocked, leading to weaker defenses. Sometimes, the requirements dependency also exists in the opposite direction. For instance, AC_F→PS-III shows that AC_F shall automatically disable the login function to the IP addresses in the blacklist after PS-III applies the punishment. The circular dependencies between AC_F and PS-III highlights the importance of identifying all interdependent requirements and then using the knowledge to test security vulnerabilities.

5) Realization Similarity. If two HSRs are realized by using the same or similar techniques, they tend to have a bidirectional dependency. For example, SED and TED in HIPAA both need encryption technology, though the former refers to the encryption and decryption of storage and the latter refers to those of transaction. If two system-level features LSR_1 and LSR_2 satisfy SED and TED by using different encryption approaches, developers might be confused and mistakenly use storage-encryption to secure transaction reports, potentially harming data integrity.

Table 2 compares our dependency types to the work on software release planning [12] and interaction management [11]. An important distinction is that our dependency analysis focuses on security requirements, and hence can be viewed as a tailored categorization of the requirements interaction types outlined by Robinson *et al*. [11]. Moreover, we significantly enhance the testability of the requirements dependencies, signified by the new "input modification" and "realization similarity" types. In contrast, the dependencies used in release planning are unique in their own ways, e.g., Cvalue and Icost concern customer value and implementation cost respectively [12]. As Table 2 shows, we identified a total of 16 and 49 dependencies in HIPAA and FIPS 200 respectively. The complete HSR dependency lists can be found in our online repository [26].

3.2 Automatically Tracing HSRs to LSRs

As shown in Fig. 2, the second step of our approach is aimed at automatically tracing an *individual* HSR at the regulation and standard level to a candidate set of LSRs at the system and product level. This set, $\{LSR_{i1}, LSR_{i2}, \ldots, LSR_{im}\}$, intends

TABLE 3
Indicator Terms Used to Incorporate Relevance Feedback (RF) Into Automatically Tracing HSRs to LSRs

Input Modification	Temporal Relation	Task Refinement	Triggering Condition	Realization Similarity
transmit	until	hardware	limit	appropriate
improper	session	software	authorize	implement
receive	timely	monitor	access	mechanism
alter	allow	protect	ensure	electronic
destroy	periodical	media	control	encryption

to capture the "complying with" relation to the security policy HSR_i. We develop automated solution based on trace recovery methods [14]. In particular, Hayes *et al.* [16] stated that requirements tracing should incorporate requirements analyst's feedback into the dynamic trace generation process, and Cleland-Huang *et al.* [27] showed that indicator terms were effective in tracing nonfunctional requirements. For example, "update" and "release" are among the indicator terms for "maintainability" whereas "user" and "learn" are for "usability".

Inspired by the prior work [16], [27], we codify the indicator terms during our manual HSR analysis, and Table 3 lists these terms. To identify these terms, we examined each dependency category and used two heuristics for extraction: (1) a term appeared frequently in a category, and (2) a term appeared mostly in only one category but seldom in other categories. Our manual work thus resembled the mechanism of detecting nonfunctional requirements indicator terms, and our future work would experiment automated methods [27] to identify these terms. We further incorporate the indicator terms in the automated requirements tracing via *relevance feedback* (RF). The basic idea of RF is to recognize positive and negative results from the retrieved list so as to modify the original (trace) query by weighting more on the terms appearing in the positive results and less on the terms from the negative ones [28]. Hayes *et al.* [16] applied standard RF:

$$Q_m = (\alpha \cdot Q_o) + (\beta \cdot \frac{1}{|D_r|} \sum_{d_j \in D_r} d_j) - \left(\gamma \cdot \frac{1}{|D_i|} \sum_{d_k \in D_i} d_k\right) \quad (1)$$

where Q_o is the original trace query (e.g., a single HSR), Q_m is the modified query, D_r is the set of relevant (positive) candidate links, and D_i is the irrelevant (negative) set. Intuitively, the modified query takes the initial query while adding the weighted $d_j \in D_r$ and subtracting the weighted $d_k \in D_i$. Weighting parameters α, β, and γ are used to assign different emphases to Q_o, positive feedback, and negative feedback, respectively. Empirically speaking, $\alpha = 1.0$, $\beta = 0.75$, and $\gamma = 0$ are shown to consistently improve a varied range of queries [28] and commonly used in automated requirements traceability [16], [29], [30].

A couple of extensions of the standard RF are worth noting. Panichella *et al.* [29] presented an adaptive version of RF. Rather than applying the standard RF to every pair of tracing source and target, the adaptive version checks certain conditions before applying the RF. The conditions are defined on the basis of the software artifacts' verbosity and the trace links already classified [29]. While both standard and adaptive RF work on the link level (e.g., every $d_j \in D_r$ receives the positive weighting defined by β), Wang *et al.* [30] proposed to apply RF at the term level, i.e., only certain terms of d_j, instead of all the terms of d_j, would be positively contributing to the modification of the trace query Q_m. Different from [30], the indicator terms listed in Table 3 are used in term-based RF.

3.3 Inferring LSR Dependencies

The third step of our approach, as presented in Fig. 2, is to establish LSR dependencies. In fact, with the completion of the first step (HSR dependency analysis) and the second step (tracing individual HSR to candidate LSRs), this last step is carried out in a straightforward and automatic way. If HSR_i depends on HSR_j ($HSR_i \rightarrow HSR_j$) at the security policy level, and $\{LSR_{i1}, LSR_{i2}, \ldots, LSR_{im}\}$ and $\{LSR_{j1}, LSR_{j2}, \ldots, LSR_{jn}\}$ are candidate product-level requirements traced to (complied with) HSR_i and HSR_j respectively, then we posit $LSR_{ik} \rightarrow LSR_{jl}$ in the same way as $HSR_i \rightarrow HSR_j$, where $1 \leq k \leq m$, $1 \leq l \leq n$, and the two interdependent LSRs are different requirements.

To illustrate the third step, we revisit the Facebook's breach [3]. Suppose "upload birthday video" is traced to the HSR_i that permits posting by authorized users and "view as" is traced to the HSR_j that grants limited access to other accounts. In addition, the dependency $HSR_i \rightarrow HSR_j$ is of type "Triggering Condition", meaning that "permitting posting" shall react to "granting limited access to other accounts". Due to $HSR_i \rightarrow HSR_j$ and the horizontal traceability, our third step detects that "upload birthday video" depends on "view as" and their dependency is also of type "Triggering Condition". This system-level dependency can be investigated before releasing the "upload birthday video" feature, potentially preventing Facebook's security breach. We next present the quantitative evaluation of our approach.

4 Experimental Evaluation

The objective of our experimental evaluation is to assess the effectiveness of our approach in detecting requirements dependencies as they relate to security. As mentioned in Section 3.1, we perform manual HSR analysis on security policies in the healthcare and education domains. Consequently, we want to choose products and systems in these two domains, and also multiple projects in each domain for the purpose of HSR applicability and reusability. To maximize replicability, we searched the ten healthcare projects studied by Cleland-Huang *et al.* [17] and found some were no longer available (e.g., ClearHealth). We thus selected three healthcare projects, CARE2X, iTrust, and WorldVistA, which shared running demos or source code. For the education domain where FIPS 200 would apply, we chose two open source projects: Moodle[5] and Scholar@UC.[6] Both Moodle and Scholar@UC are production systems deployed for real-life uses.

Table 4 provides some basic characteristics of the five subject systems in our experiment. The project-level LSRs

5. https://moodle.org
6. https://scholar.uc.edu

TABLE 4
Five Projects Used to Evaluate Requirements Dependency Detection

HSRs	Project	LSR format	# of LSRs	# of D
HIPAA	CARE2X	feature	24	22
	iTrust	use case	36	43
	WorldVistA	feature	116	193
FIPS 200	Scholar@UC	user story	144	267
	Moodle	feature	1182	1396

(HSRs: High-Level Security Requirements; LSRs: Low-Level Security Requirements; D: Dependency Among the LSRs).

are documented in different formats: While iTrust adopts use case diagram and descriptions, other systems use lightweight methods, such as feature requests and user stories, to support agile development and continuous deployment [13], [31]. In all cases, natural languages are used to convey the requirements, making information retrieval algorithms suitable for automatically generating the traceability information.

To evaluate our approach presented in Fig. 1c, we consider *vertical* trace recovery mechanisms to be the baseline methods. In particular, the baseline methods trace LSRs to establish interdependencies among themselves *without* taking HSRs into account. Following the observation made by Carlshamre *et al.* [12], if two requirements have a textual similarity score that is greater than 0.125, we mark that there is a dependency between these two LSRs. When calculating textual similarity, we not only use TF-IDF [12] but also enhance it with standard RF [16], adaptive RF [29], and term-based RF [30], as discussed in Section 3.2. The baseline group thus has four methods for detecting the dependencies among the LSRs: vertical requirements tracing (VRT) with TF-IDF, VRT with TF-IDF enhanced by standard RF, VRT with TF-IDF enhanced by adaptive RF, and VRT with TF-IDF enhanced by term-based RF.

In contrast, our approach explicitly considers HSRs and their interdependencies. From a traceability perspective, our approach differs from the baseline group in that the automation occurs *horizontally*, i.e., from an individual HSR which acts as the trace query to a candidate set of LSRs which are the tracing targets. The LSR dependencies are then established by combining such horizontal traceability with the HSR dependencies identified manually. We therefore refer to our approach as hybrid, and in correspondence to the baseline group, we assess four methods implementing our approach: hybrid requirements tracing (HRT) with TF-IDF, HRT with TF-IDF enhanced by standard RF, HRT with TF-IDF enhanced by adaptive RF, and HRT with TF-IDF enhanced by term-based RF.

The experimental results are summarized in Table 5 where three metrics (i.e., Recall, Precision, and F_2 measure) are computed to evaluate the dependency detection accuracy of different approaches:

$$R = \frac{|D \cap T|}{|T|} \tag{2}$$

$$P = \frac{|D \cap T|}{|D|} \tag{3}$$

$$F_2 = \frac{5 \cdot R \cdot P}{R + 4 \cdot P} \tag{4}$$

where R, P, F_2, D, T refer to "Recall", "Precision", "F_2 measure", "detected dependencies", and "true dependencies" respectively. The answer set of the true dependencies was constructed by the authors of this paper, and the rightmost column of Table 4 shows the total number of true dependencies in each subject system. To facilitate replication and cross validation, the answer set along with all other materials can be found in our online repository [26]. In Table 5, each project is traced by VRT and HRT, and the average recall, precision, and F2 values are reported. In the bottom of Table 5, averages across the five projects are summarized.

We perform Wilcoxon signed-rank test [32] to draw statistical inferences between a particular VRT method and its HRT counterpart. The results show that our HRT approach achieves significantly better accuracy when compared to the VRT baseline. We deem these results important as they extend the existing literature: Although the accuracy of TF-IDF, standard RF, adaptive RF, and term-based RF has been demonstrated in tracing *individual* requirements (e.g., the average F_2 of term-based RF is 48 percent in [30]), our results (e.g., the average F_2 of Table 5 is 73 percent) show that these automated tracing methods are effective in detecting requirements *dependencies*, too.

Among the different methods that we experimented, it seems that term-based RF is the most suitable for implementing our HRT approach (e.g., term-based RF is the only method that improves the F_2 measure at the 0.01 level in all five projects). We also noticed that the performance across different systems varies. Of the five experimental projects, CARE2X is the only one where all features are submitted by end users. Surprisingly, the term-based RF implementation of our approach achieves the second highest performance on this project. iTrust is the one that achieves better performance than CARE2X. One difference between these two projects is that iTrust's requirements are recorded in a relatively heavy-weight format (i.e., use cases) and CARE2X adopts the lightweight format of feature requests. This difference may indicate that how requirements are documented and modeled could impact dependency identification. Testing the hypothesis requires future work.

From Table 5, we can observe a theme that is in common for five projects: our HRT approach significantly improves recall, but not precision, consistently. We speculate the main reason being the underlying assumption that we have about the third step where dependencies among LSRs are automatically inferred, i.e., $\text{LSR}_{ik} \rightarrow \text{LSR}_{jl}$ if $\text{HSR}_i \rightarrow \text{HSR}_j$, LSR_{ik} is a candidate link of HSR_i, and LSR_{jl} is a candidate link of HSR_j. This assumption may cause some false positives, hurting precision. The tradeoff is the high recall level which is favored by the requirements tracing community [14].

While the results of Table 5 show the accuracy improvements (especially recall improvements) of our approach over the VRT baseline, the manual cost of HSR-dependency analysis shall not be neglected. As mentioned in Section 3.1, we spent about 50 hours in total manually identifying the dependencies among the twelve HIPAA HSRs and those among the 29 FIPS 200 HSRs. The consistently high accuracy levels across five unrelated projects in two different domains

TABLE 5
Results for Requirements Dependency Detection

Project	Approach	Automated Requirements Tracing Method											
		TFIDF			Standard RF			Adaptive RF			Term-Based RF		
		R	P	F_2	R	P	F_2	R	P	F_2	R	P	F_2
CARE2X	VRT	0.67	0.42	0.59	0.63	0.67	0.64	0.71	0.63	0.69	0.77	0.64	0.74
	HRT	0.64	0.49	0.60	0.71 $*$	0.57	0.68 $*$	0.77 $*$	0.61	0.73 $*$	0.88 $**$	0.64	0.82 $**$
iTrust	VRT	0.64	0.55	0.62	0.65	0.61	0.64	0.73	0.65	0.71	0.79	0.67	0.76
	HRT	0.72 $*$	0.53	0.67 $*$	0.73 $*$	0.57	0.69 $*$	0.75	0.63	0.72	0.91$**$	0.71$*$	0.86 $**$
WorldVistA	VRT	0.52	0.42	0.50	0.53	0.47	0.51	0.62	0.48	0.59	0.72	0.46	0.65
	HRT	0.64 $*$	0.47 $*$	0.60 $*$	0.65 $**$	0.48	0.61 $**$	0.67 $*$	0.51	0.63 $*$	0.87 $**$	0.52 $*$	0.77 $**$
Scholar@UC	VRT	0.69	0.37	0.59	0.74	0.38	0.62	0.71	0.37	0.60	0.74	0.41	0.64
	HRT	0.72	0.43 $*$	0.63	0.81	0.44 $*$	0.69	0.82 $*$	0.43	0.69 $*$	0.89 $**$	0.45	0.74 $**$
Moodle	VRT	0.43	0.44	0.43	0.54	0.46	0.52	0.58	0.45	0.55	0.67	0.43	0.60
	HRT	0.49 $*$	0.47	0.49 $*$	0.62 $*$	0.49	0.59 $*$	0.65 $*$	0.49	0.61 $*$	0.79 $**$	0.51 $*$	0.71 $**$
Average	VRT	0.59	0.44	0.55	0.62	0.51	0.59	0.67	0.52	0.63	0.74	0.52	0.68
	HRT	0.64	0.47	0.60	0.70	0.51	0.65	0.73	0.53	0.68	0.87	0.57	0.78

VRT: vertical requirements traceability (baseline approach); HRT: hybrid requirements traceability (our approach); $*$: p-value of the Wilcoxon signed-rank test is less than 0.05; $**$: p-value of the Wilcoxon signed-rank test is less than 0.01.

present encouraging results in the applicability and reusability of our manual HSR dependencies. We thus make the HIPAA and FIPS 200 dependencies available in [26] with the anticipation that others can build on our work in at least two dimensions. First, the HIPAA and FIPS 200 dependencies can help automatically detect LSR dependencies in other healthcare or education systems. Second, the operational insights drawn from our manual analysis (e.g., the five HSR-dependency types and their indicator terms) can be used for building automated tools to identify and classify HSR dependencies in other policies, regulations, and standards.

5 Case Study on Vulnerability Detection

Like traceability [33], requirements dependency is not an end in itself but a means to support various software engineering tasks. In this research, our main goal is to uncover vulnerabilities before a software system is released or updated. In another word, we want the requirements dependencies established to be exploited in system-level, black-box testing so as to identify implementation defects compromising security. We report in this section a case study that we carried out to investigate vulnerability detection within a system's real-life context.

The contemporary phenomenon of our investigation is the Scholar@UC project, which has been deployed and continuously serving its user base since February 2016. Scholar@UC is a digital repository that enables the University of Cincinnati (UC) community to share its research and scholarly work with a worldwide audience. Its mission includes preserving the permanent intellectual output of UC (e.g., publications, presentations, datasets, etc.) and enhancing discoverability of these resources. UC faculty and students, for instance, can use Scholar@UC to store, organize, and distribute their scholarly creations in a durable and citable manner.

The development of Scholar@UC evolved from a couple of legacy Web systems and the principal technological platforms are the Samvera Hyrax repository architecture, Apache Solr server, Ruby on Rails engine, and Blacklight interface. As shown in Table 4, our approach identifies 267 dependencies among Scholar@UC user stories. For each pair of interdependent user stories, our system-level security testing was performed manually by taking advantage of the attack payloads from the XSS Filter Evasion Cheat Sheet [22].

Fig. 3 illustrates our security testing which is informed by the requirements dependency: "create new work"[7]→"view work's citation".[8] When creating new work, we inject different XSS payloads into the permissible entries. For example, Fig. 3a shows that the payload—XSS testing: $<$script$>$prompt("Enter password")$<$/script$>$—is fed into the "Title" field when a generic work is created. Because of the requirements dependency, our next testing step is to "view work's citation" as shown in Fig. 3b. It is at this second testing step that an "Enter password" window pops up, as shown in Fig. 3c, which detects a vulnerability in the implementation of Scholar@UC.

The vulnerability illustrated in Fig. 3 is one of a dozen that we detected in Scholar@UC while the software was deployed and serving. In a one-hour meeting with the project's core developers, we shared our findings which the development team was able to replicate. The team then took further actions to report four vulnerabilities to the broader Samvera Hyrax community,[9] leading to the mitigation of these or similar XSS attacks. Given that Hyrax has more than 110 developers making code contributions and the Hyrax framework underlies many web applications used in organizations like universities, public libraries, and presses,[10] we believe the detected vulnerabilities are significant which further indicates our approach based on requirements dependency analysis complements existing vulnerability discovery techniques.

Fig. 4 presents an important lesson learned from our case study. Using requirements dependencies to perform security testing has a direct influence on *test path selection*. In Fig. 4, the four XSS vulnerabilities reported in the GitHub issue are manifested in an *optimal* manner, and here, optimality means that one more testing step would be a waste and one less step would be insufficient to reveal the vulnerability. For example, after a new work is created, there are many test paths that one can follow: add a DOI link, share with other users, link to existing works, etc. However, "view work's citation" is among the optimal tests to

7. https://trello.com/c/LHaTnSnF/
8. https://github.com/uclibs/scholar_use_cases/blob/master/display_download/display_download_use_cases.md
9. https://github.com/samvera/hyrax/issues/3187
10. https://samvera.org/samvera-partners/

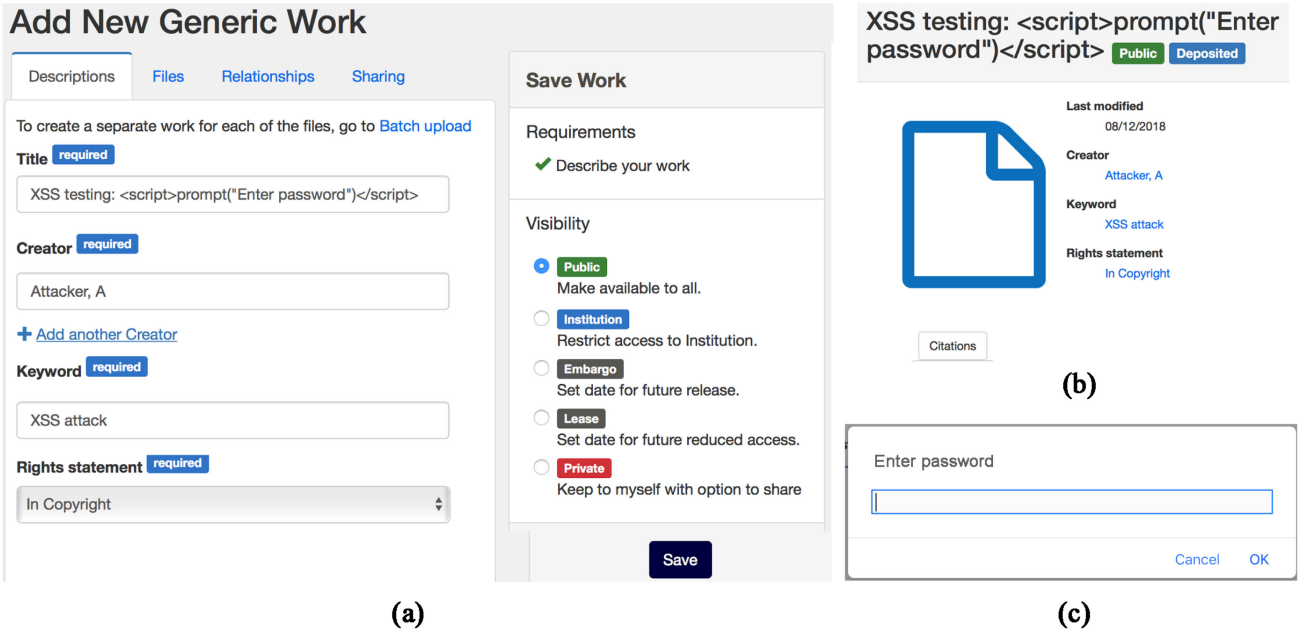


Fig. 3. An XSS vulnerability detected in Scholar@UC: **(a)** submit work, **(b)** export citation, and **(c)** pop-up window created by XSS code in work's title.

perform due to the "create new work"→"view work's citation" dependency. Fig. 4, therefore, depicts a set of optimal test paths for detecting vulnerabilities.

Another lesson learned in our Scholar@UC study is that the requirements dependencies tend to be compositional when they are used in system-level vulnerability detection. This has a couple of implications. Even though requirements dependency is commonly analyzed in a pairwise manner (e.g., in prior research [11], [12] and in our approach shown in Fig. 1c), the actual uses of the dependencies can be in a more networked fashion. The test-path graph of Fig. 4 illustrates this point, in which injecting XSS payloads and revealing vulnerabilities become more efficient. Second, the compositional characteristic shows the importance of high recall in identifying requirements dependencies, which further confirms the underlying assumption made in our hybrid traceability approach, i.e., any LSRs are interdependent if their corresponding HSRs are interdependent. This hybrid mode of explicitly considering HSR dependencies helps to improve recall and to facilitate LSR dependency test path construction, as shown in Fig. 4.

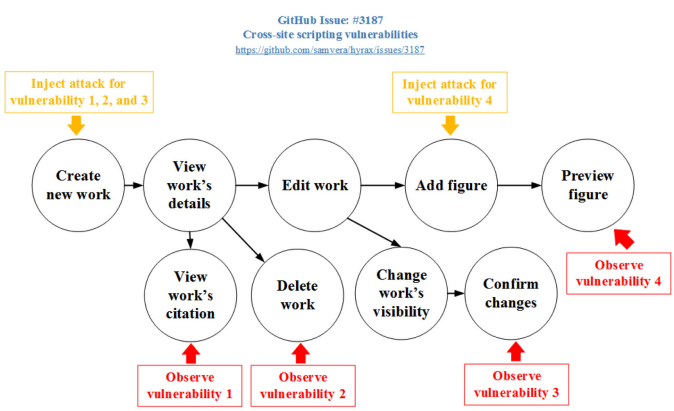


Fig. 4. Security testing paths informed by requirements dependencies.

6 Discussion

This section discusses the limitations of our approach and the threats to validity of our experimental evaluation and case study. The first step of our approach, which analyzes the HSR dependencies, is done manually. Although future automation is possible [27], our intention is to make sure that all dependencies are identified and all identified dependencies are valid. Additional justification of manual analysis at this step comes from the relatively small number and steady evolution of HSRs. Although our evaluation with five projects show the reusability of the manual analysis results, the HSR dependencies are further validated by a security expert who has had over ten years of industrial software development and operations experience. All our results are shared in [26] to facilitate replication, cross validation, and expansions.

A threat to construct validity is that we rely on HSRs like HIPAA to automatically trace *security* requirements at the product and system level. Approaches distinguishing security and non-security requirements exist. For example, Wang *et al.* [34] developed a linear regression classifier based on features, such as stakeholders and comments, to identify security requirements in open-source projects, and their experiments on three systems showed an average of $F_2 = 84$ percent in security requirements classification. In our approach, due to the consideration of textual requirements only, no stakeholder or comment features are leveraged. While integrating a security requirements classifier like the linear regression model [34] may improve accuracy, our automated tracing is built on related work in HSRs [17] and dependability requirements [30].

When tracing HSRs to LSRs, a potential threat to internal validity is our use of indicator terms without further distinguishing the dependency type information. In another word, the terms presented in Table 3, rather than the terms of certain type or types, are used in RF algorithms. One reason is that these indictor terms, in line with the nonfunctional requirements indicator terms [27], are fairly common across

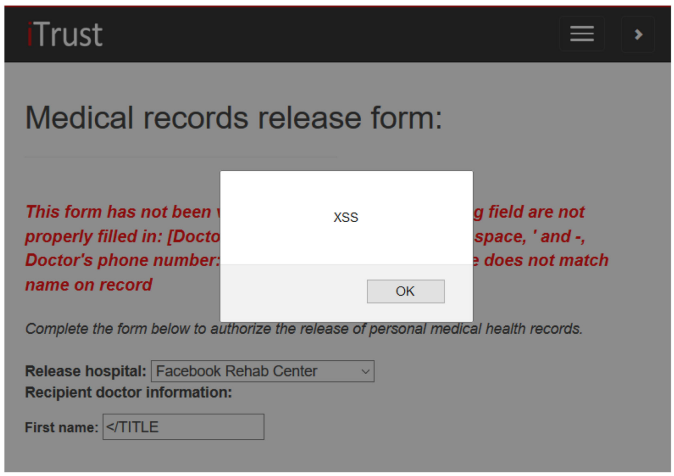


Fig. 5. Vulnerability detected in iTrust.

documents. In our work, these indicators appear in both HIPAA and FIPS 200. Another reason is that, when the LSR dependencies are used, e.g., in security vulnerability testing, their type information can be automatically inferred from the HSRs, which avoids overly depending on indicator terms for dependency type identification.

In the case study on the Scholar@UC project, a limitation lies in our use of only XSS attack payloads [22]. Although we tried the payloads in all the 267 interdependent Scholar@UC requirements, not every pair of requirements was applicable and only a dozen vulnerabilities were detected. Notably, XSS payloads apply most directly to the "Input Modification" type of LSRs. For other types, security testing might have to devise different payloads or attack vectors, e.g., trust boundary violation (CWE-501) may be used to test LSRs which follow "Triggering Condition" dependencies. At any rate, the test paths informed by LSR dependencies (cf. Fig. 4) can be valuable in discovering security vulnerabilities.

Threats to external validity concern the generalizability of the findings. Beyond the five projects that we experimented with, it is interesting to analyze more healthcare and educational systems given that the HSR dependencies of HIPAA and FIPS 200 have already been established. A challenge is the lack of answer sets for new systems, which makes it difficult to compute metrics like recall and precision. In contrast, vulnerability discovery does not require any answer set. For a software system, if an attack succeeds, then we are confident that a previously unknown vulnerability is detected. For this reason, we tried to generalize our results from the Scholar@UC case study.

One surprising observation came from our security testing on iTrust. Fig. 5 illustrates this situation where the malicious code "</TITLE><SCRIPT>alert("XSS")</SCRIPT>" is injected into the "First name" textbox of the medical records release request form. Due to the detected LSR dependency, we were expecting to reveal the vulnerability, if any, after traversing to another requirement ("view updated demographics information by a doctor or representative"). However, a vulnerability is revealed at the same page as shown in Fig. 5. Upon our analysis, the reason is that iTrust implements user input sanitization procedures, and in this case, the full payload does not pass iTrust's sanitization. Yet, when the page is reloaded and prompted for the user to update the demographics information again, a pop-up XSS attack shows. Interestingly, iTrust treats the first ">" in the original payload as the ending symbol of the textbox and executes the malicious code in the later part, and hence "</TITLE" ends up being displayed in the "First name" textbox of the reloaded page. Our serendipitous detection of this iTrust vulnerability not only shows the importance of having a diverse set of payloads in security testing, but also indicates that dependency of the same requirement (in addition to dependency between different ones) may be worth considering which will update one of the core constructs of our proposed approach.

7 Conclusion

A high-level security policy, regulation, or standard describes the requirements that an organization shall meet to protect its assets. When a product or system fails to comply with the high-level security requirements, breaches occur [35]. In this paper, we advance vulnerability discovery by focusing not only on individual security requirements, but on their interdependencies. We propose a semi-automatic approach by integrating horizontal and vertical traceability to identify requirements dependencies. We then show how the dependencies can be used in performing security tests to uncover vulnerabilities.

Manually analyzing HIPAA and FIPS 200 security requirements allows us to detect dependencies of five projects' requirements. Compared to the baseline methods, our approach significantly improves recall at 81 percent. Our case study on Scholar@UC helped discover four previously unknown vulnerabilities, contributing to the overall security of the Samvera Hyrax community. Future work of our study includes: 1) investigating automated ways of identifying indicator terms for different types of requirements dependency; 2) applying our approach to other industrial projects within healthcare and education domains to further assess reusability; 3) developing automated security testing tools with the path selections informed by the requirements dependencies and the payloads exploited from diverse sources like the XSS Filter Evasion Cheat Sheet [22]; and 4) extending our approach toward self-dependencies and the dependencies over a network of requirements.

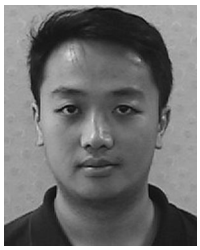
Acknowledgments

Funding is provided in part by the U.S. National Science Foundation (Award CCF-1350487) and the Ohio Cyber Range.

References

[1] National vulnerability database, "CVSS severity distribution over time," 2020, Accessed: Oct. 2020. [Online]. Available: https://nvd.nist.gov/general/visualizations/vulnerability-visualization
[2] M. Kumar, "Worst day for eBAY, multiple flaws leave milliions of users vulnerable to hackers," 2014, Accessed: Oct. 2020. [Online]. Available: https://thehackernews.com/2014/05/worst-day-for-ebay-multiple-flaws-lea
[3] M. Isaac and S. Frenkel, "Facebook security breach exposes accounts of 50 million users," 2018, Accessed: Oct. 2020. [Online]. Available: https://www.nytimes.com/2018/09/28/technology/facebook-hack-data-breach
[4] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proc. Int. Conf. Softw. Eng.*, 2008, pp. 171–180.
[5] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, Nov./Dec. 2011.

[6] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Trans. Softw. Eng.*, vol. 32, no. 1, pp. 4–19, Jan. 2006.
[7] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of SQL injection and cross-site scripting attacks," in *Proc. Int. Conf. Softw. Eng.*, 2009, pp. 199–209.
[8] O. Tripp, O. Weisman, and L. Guy, "Finding your way in the testing jungle: A learning approach to web security testing," in *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 347–357.
[9] L. K. Shar, H. B. K. Tan, and L. C. Briand, "Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 642–651.
[10] J. Antunes, N. F. Neves, M. Correia, P. Verssimo, and R. F. Neves, "Vulnerability discovery with attack injection," *IEEE Trans. Softw. Eng.*, vol. 36, no. 3, pp. 357–370, May/Jun. 2010.
[11] W. N. Robinson, S. D. Pawlowski, and V. Volkov, "Requirements interaction management," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 132–190, Jun. 2003.
[12] P. Carlshamre, K. Sandahl, M. Lindvall, B. Regnell, and J. Nattoch Dag, "An industrial survey of requirements interdependencies in software product release planning," in *Proc. Int. Requirements Eng. Conf.*, 2001, pp. 84–93.
[13] N. Niu *et al.*, "Requirements socio-technical graphs for managing practitioners' traceability questions," *IEEE Trans. Comput. Social Syst.*, vol. 5, no. 4, pp. 1152–1162, Dec. 2018.
[14] J. Cleland-Huang, O. Gotel, J. H. Hayes, P. Mäder, and A. Zisman, "Software traceability: Trends and future directions," in *Proc. Future Softw. Eng.*, 2014, pp. 55–69.
[15] J. Hayes, A. Dekhtyar, and J. Osborne, "Improving requirements tracing via information retrieval," in *Proc. Int. Requirements Eng. Conf.*, 2003, pp. 138–147.
[16] J. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing candidate link generation for requirements tracing: The study of methods," *IEEE Trans. Softw. Eng.*, vol. 32, no. 1, pp. 4–19, Jan. 2006.
[17] J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Emenecker, "A machine learning approach for tracing regulatory codes to product specific requirements," in *Proc. Int. Conf. Softw. Eng.*, 2010, pp. 155–164.
[18] P. Rempel and P. Mäder, "Preventing defects: The impact of requirements traceability completeness on software quality," *IEEE Trans. Softw. Eng.*, vol. 43, no. 8, pp. 777–797, Aug. 2017.
[19] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics versus text mining," in *Proc. Int. Symp. Softw. Rel. Eng.*, 2014, pp. 23–33.
[20] Open web application security project, "OWASP Zed Attack Proxy (ZAP)," 2020. Accessed: Oct. 2020. [Online]. Available: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project
[21] S. Kals, E. Kirda, C. Krügel, and N. Jovanovic, "SecuBat: A web vulnerability scanner," in *Proc. Int. Conf. World Wide Web*, 2006, pp. 247–256.
[22] Open web application security project, "XSS filter evasion cheat sheet," 2020. Accessed: Oct. 2020. [Online]. Available: https://owasp.org/www-community/xss-filter-evasion-cheatsheet
[23] J. Nattoch Dag, B. Regnell, P. Carlshamre, M. Andersson, and J. Karlsson, "Evaluating automated support for requirements similarity analysis in market-driven development," in *Proc. Int. Work. Conf. Requirements Eng.: Found. Softw. Quality*, 2001, pp. 190–201.
[24] R. S. Ross, S. W. Katzke, and L. A. Johnson, "Minimum security requirements for federal information and information systems," 2006, Accessed: Oct. 2020. [Online]. Available: https://www.nist.gov/publications/minimum-security-requirements-federal-information-and-information-systems
[25] United States Department of Education, "*Protecting student privacy while using online educational services: Requirements and best practices* US Department of Education," 2014, Accessed: Oct. 2020. [Online]. Available: https://tech.ed.gov/wp-content/uploads/2014/09/Student-Privacy-and-Onli%ne-Educational-Services-February-2014.pdf
[26] W. Wang and N. Niu, "Artifacts of 'detecting software vulnerabilities via requirements dependency analysis'," 2020, Accessed: Oct. 2020. [Online]. Available: http://dx.doi.org/10.7945/q8h2-t712
[27] J. Cleland-Huang, R. Settimi, X. Zou, and P. Solc, "The detection and classification of non-functional requirements with application to early aspects," in *Proc. Int. Requirements Eng. Conf.*, 2006, pp. 36–45.
[28] I. Ruthven and M. Lalmas, "A survey on the use of relevance feedback for information access systems," *Knowl. Eng. Rev.*, vol. 18, no. 2, pp. 95–145, Jun. 2003.
[29] A. Panichella, A. De Lucia, and A. Zaidman, "Adaptive user feedback for IR-based traceability recovery," in *Proc. Int. Symp. Softw. Syst. Traceability*, 2015, pp. 15–21.
[30] W. Wang, A. Gupta, N. Niu, L. D. Xu, J.-R. C. Cheng, and Z. Niu, "Automatically tracing dependability requirements via term-based relevance feedback," *IEEE Trans. Ind. Informat.*, vol. 14, no. 1, pp. 342–349, Jan. 2018.
[31] N. Niu, S. Brinkkemper, X. Franch, J. Partanen, and J. Savolainen, "Requirements engineering and continuous deployment," *IEEE Softw.*, vol. 35, no. 2, pp. 86–90, Mar./Apr. 2018.
[32] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bull.*, vol. 1, no. 6, pp. 80–83, Dec. 1945.
[33] N. Niu, W. Wang, and A. Gupta, "Gray links in the use of requirements traceability," in *Proc. Int. Symp. Foundations Softw. Eng.*, 2016, pp. 384–395.
[34] W. Wang, K. R. Mahakala, A. Gupta, N. Hussein, and Y. Wang, "A linear classifier based approach for identifying security requirements in open source software development," *J. Ind. Inf. Integr.*, vol. 14, pp. 34–40, Jun. 2018.
[35] Ö. Kafali, J. Jones, M. Petruso, L. Williams, and M. P. Singh, "How good is a security policy against real breaches?: A HIPAA case study," in *Proc. Int. Conf. Softw. Eng.*, 2017, pp. 530–540.



Wentao Wang (Member, IEEE) received the BSc degree in computer science from Shanghai Maritime University, Shanghai, China, in 2007, the MEng degree in software engineering from the Beijing Institute of Technology, Beijing, China, in 2010, and the PhD degree in computer science and engineering from the University of Cincinnati, Cincinnati, OH, USA, in 2019. He is currently a member of technical staff at Oracle America, Inc. His research interests include software requirements engineering, requirements traceability, and software security.

Faryn Dumont is currently working toward the BSc degree in computer science from the University of Cincinnati. She is currently an ORISE Participant at the United States Environmental Protection Agency.

Nan Niu (Senior Member, IEEE) received the BEng degree in computer science and engineering from the Beijing Institute of Technology, Beijing, China, in 1999, the MSc degree in computing science from the University of Alberta, Edmonton, AB, Canada, in 2004, and the PhD degree in computer science from the University of Toronto, Toronto, ON, Canada, in 2009. He is currently an associate professor with the University of Cincinnati's Department of Electrical Engineering and Computer Science. His research interests include requirements engineering, scientific software development, and human-centric computing.

Glen Horton received the BS degree in computer science from Wright State University, in 1998, and the MS degree in information technology from the University of Cincinnati, in 2020. He is currently the head of the Application Development Unit at the University of Cincinnati Libraries. His research interests include open source software communities, collaborative software development, agile development methods, and mining software repositories.