Recovering Fine Grained Traceability Links Between Software Mandatory Constraints and Source Code

Alejandro Velasco([✉]) and Jairo Hernan Aponte Melo([✉])

Departamento de Ingeniería de Sistemas e Industrial,
Universidad Nacional de Colombia, Bogotá, Colombia
{savelascod, jhapontem}@unal.edu.co

Abstract. Software traceability is a necessary process to carry out source code maintenance, testing and feature location tasks. Despite its importance, it is not a process that is strictly conducted since the creation of every software project. Over the last few years information retrieval techniques have been proposed to recover traceability links between software artifacts in a coarse-grained and middle-grained level. In contexts where it is fundamental to ensure the correct implementation of regulations and constraints at source code level, as in the case of HIPAA, proposed techniques are not enough to find traceability links in a fine-granular way. In this research, we propose a fine-grained traceability algorithm to find traces between high level requirements written in human natural language with source code lines and structures where they are implemented.

Keywords: Software traceability · Information retrieval · Static code analysis · Software maintenance · Program slicing · Natural language processing · Healthcare

1 Introduction

Software traceability is a research area in software engineering that aims to recover traces and links between high-level artifacts (e.g. documentation, use case diagrams, requirements specifications) and source code artifacts (e.g. classes and methods) [12, 26]. Due to its nature, the traceability of the code constitutes an imperative role in code comprehension and helps to perform a wide variety of tasks such as bug tracking, feature location, and software testing. Traceability focuses on making easier the assurance of stakeholder's needs and the correct implementation of functional and non-functional requirements in any software system [33].

Although traceability should be carried out from the beginning of any software project [37], in practice, it is an unusual activity. Developers usually center their efforts more on building the software system than on making documentation artifacts. Similarly, it is a common practice of developers to face the documentation of the system in the final stages of any project to fulfill deadlines [37]. These practices may cause serious problems when assuring the quality of the constructed artifacts as well as the verification of the compliance of final products with the initial requirements and needs of stakeholders.

https://doi.org/10.1007/978-3-030-32475-9_37

In the context of critical software systems, it is crucial that the software products comply with security, privacy, and safety regulations (e.g., HIPAA law for healthcare information systems). Software companies and regulatory entities, such as the HHS (U. S. Department of Health and Human Services) Office for Civil Rights, must assure regulatory compliance of software systems [3]; typically, regulators conduct manual inspection of the source code to achieve such purposes [14, 17, 35]. To support people on critical mandatory constraint's verification (which is a difficult and error-prone traceability task), research has devoted to develop automated techniques on traceability links recovery. However, existing techniques only go down to link classes and methods with high-level constraints. Mandatory constraint's compliance requires going to deeper granularity levels in order to confirm and assure that software systems implement mandatory regulations and requirements.

Fine-grained traceability link recovery is intended to fully support requirement compliance of software, by linking critical mandatory constraints to specific code structures such as statements, conditions, variable assignments and basic code blocks. To accomplish such objective, we developed a software traceability technique which based on heuristics, IR techniques and static software code analysis, is able to identify traces between software mandatory constraints and source code structures.

2 Related Work

Requirement mapping is an expensive task that is required to ensure the software compliance of constraints and features given by stakeholders and organizations [14]. In the particular case of HIPAA [1], there are several studies conducted to classify the statutes by topology in order to make the validation process a more comprehensive task. HIPAA law is divided in three principal sets of regulatory constraints, in the particular case of healthcare software systems, HIPAA establish a set of standards and rules that must be covered and taken into account in order to protect the security and confidentiality, and ensure the correct management of all patients' data [3]. Breaux [15, 17], defined a semantic model by identifying language patterns to extract rights and obligations from HIPAA statutes, the requirement classification was conducted for privacy rules on HIPAA regulations. Based on the Breaux methodology, two frameworks were constructed and evaluated [16, 27, 38]. Alshugran et al., proposed as well a process to extract privacy requirements from HIPAA, since the law regulations are written in a complex and dense format, authors proposed a set of methods to analyze, extract and model privacy rules [9]. There are also studies that intend to find the best way to evaluate the legal compliance of HIPAA requirements in software healthcare systems. Maxwell et al. proposed a production rule model to encourage requirement engineers to keep trace between law and high level artifacts across every development process [30], authors validated the proposed technique against iTrust, an HIPAA compliant healthcare system.

2.1 Traceability Links Recovery

Most techniques for traceability link recovery between software requirements and source code operate at coarse-grained and middle-grained levels of granularity [2, 10, 11, 13, 18, 21, 24, 28, 29, 31]. Fasano [24] proposes ADAMS, a tool for automatic traceability management; the recovery link technique proposed, uses Latent Semantic Indexing (LSI) as an improvement of the Vector Space Model technique [20, 23], under the assumption that almost every software system has high level software artifacts with well-defined hierarchical structure. ADAMS is able to recover traceability links between software high level documentation and classes in the source code.

Marcus and Maletic [29] address the problem of recovering traceability links between methods and documentation, using LSI, which extract the meanings (i.e., semantics) of the documentation and source code, using all the comments and identifier names presents in the source code.

Paloma et al. [31] designed a tool called CRYSTAL (Crowdsourcing RevIews to SupporT App evoLution) to create traceability links between commits, issues (given a software release) and user reviews, using IR techniques with some adaptations to remove useless words, calculate the textual similarity of the artifacts, and select the candidate links according to a criterion (i.e., a threshold value).

De Lucia et al. [28] adopt the use of smoothing filters to reduce the effects of noise in software artifacts. By an empirical study, the authors describe a significant improvement of the IR techniques Vector Space Model and Latent Semantic Indexing.

Diaz et al. [21] proposed the usage of code ownership to improve the candidate links generation under the assumption that if a developer authored code that is linked to a particular high level artifact, then other code developed by the same author is likely to be associated with the same artifact. This technique was adopted to tackle the problem of vocabulary mismatch present in the associated artifacts. Their solution was named TYRION (TraceabilitY link Recovery using Information retrieval and code OwNership).

2.2 Alternative Methods and Improvements for Traceability Link Recovery Techniques

There are also approaches that aim to produce results at a fine-grained granularity level, between features and specific source code structures like statements, decisions and basic blocks [22]. Those strategies, mainly take advantage of the execution traces of defined features in a software system, heuristics, and source code static analysis techniques [19, 32, 34–36]. Wong et al. [36] propose a technique based on execution slices, which takes as an input set of test cases that exercise and a set of test cases that do not exercise the feature of interest. Using dynamic information extracted by running the instrumented software system (i.e., list of statements executed), the technique is able to distinguish between code that is unique to a feature, and code that is common to several features.

Dagenais and Robillard [19] propose a model based on heuristics and source code static analysis, to recover traceability links between the software documentation, and the software API documentation. The authors designed a meta-model representation of the involved artifacts (i.e., documentation, source code and support channels) to understand the context in which a code-like term is used.

Sharif and Maletic [5, 34] developed an approach to support the evolution of traceability links. The process to update and evolve traceability links is supported by the differentiation between versions of the involved artifacts in a traceability link. The authors compare XML tags to address the new locations and evolve the links.

2.3 Limitations of Adopted Strategies

Current IR-based techniques for recovering traceability links, designed to associate software requirements to code, have focused on coarse-grained granularity (i.e., files and classes), they trace high-level software artifacts to files, classes, up to methods. The reason for this is that granular source code structures (e.g., if-statements or exception handling statements) usually do not contain enough textual information that matches the vocabulary of high-level artifacts [23]. This leads to low accuracy of IR-based techniques. Therefore, it is necessary to define new techniques or improve the existing ones if we want to generate traceability links between high-level documents (such as software mandatory constraints) and code structures (e.g., conditionals, assignations, method calls).

3 Dataset Description

In order to trace high level constraints into source code by using a software traceability technique, we define a taxonomy to classify HIPAA statutes that are more concerned with software implementation standards and regulations. Not all sections written in the HIPAA rules are related to software healthcare management systems; since in this study we aim to trace regulations related to software implementations, we filtered the most related HIPAA statutes and user data security regulations (as stated in the privacy rule); then, we extracted a set of law statutes that apply for our particular problem. Table 1 summarizes the HIPAA administrative simplification, in this study we decided to analyze the Security and Privacy rule of HIPAA since it contains the major part of regulations related to the implementation of healthcare management systems.

We take as input those statutes derived from HIPAA security and privacy rules as were defined in the administrative simplification; then we identified those that are suitable to associate with source code implementations and configurations in healthcare systems. As a result, we defined a taxonomy of rules in which we classified the law constraints according to three categories.

The first category (A) defines the rules that could be easily traced into code. It consists of regulations that mainly refer to functional requirements that must be implemented in every HIPAA compliant healthcare system. The second category (B) has those statutes that define constraints related to software implementation and non-functional requirements, still suitable to be traced to code. Finally, the third category (C) groups statutes and standards related to the way in which organizations and health care providers assume practices and methods to interact with healthcare systems in order to protect the private information of patients, and that therefore are unsuitable

Table 1. Taxonomy of HIPAA Security and Privacy regulations.

ID	Category	Subpart	Statutes samples
A	Potentially suitable to trace into code	Security Standards: General Rules Administrative safe guards Technical safeguards	164.306(a)(1), 164.306(a)(2), 164.306(a)(3), 164.308(a)(1)(ii)(D), 164.308(a)(5)(ii)(C), 164.308(a)(5)(ii)(D)
B	Moderate suitable to trace into code	Administrative safe guards	164.308(a)(1)(i), 164.308(a)(1)(ii), 164.308(a)(4)(ii)(A), 164.308(a)(4)(D)(B)
C	Indirectly related to health care software systems	Security Standards: General Rules Administrative safe guards Physical safeguards	164.306(a)(1), 164.306(a)(2), 164.306(a)(3), 164.306(c), 164.306(d), 164.306(d)(1), 164.306(d)(2), 164.306(d)(3)

to be traced to code. Table 1 shows a summary of the HIPAA taxonomy from privacy and security rules that we considered in this study.

3.1 Healthcare Systems

In order to design a fine-grained traceability algorithm with the ability to trace and associate links between high level artifacts and software source code structures, we identify healthcare software systems whose source code is available. Thus, we focused in java open source healthcare systems with available documentation (e.g. use cases, user stories, user manuals). It is also important to mention that such health care systems were explicitly defined as HIPAA compliant in official pages.

Our search process identified four candidates: iTrust [5, 7], OpenMRS [4], OSCAR [6] and TAPAS [8]. These open source projects have available documentation that describes in different ways the features and characteristics implemented by each system. Table 2 summarizes the main characteristics of these four health care systems.

Table 2. Summary of open source health care systems chosen to analyze in this study.

Name	Repository	Version
TAPAS	https://sourceforge.net/projects/tap-apps/	v. 0.1
iTrust	https://sourceforge.net/projects/it	v. 21
OSCAR	https://sourceforge.net/projects/scarmcmaster	v. 14.0.0
OpenMRS	https://github.com/openmrs	v. 1.12.x

Each of the health care systems have conventional features that implement standards defined in HIPAA law. The features that we could evidence in all systems by observing the deployed applications include user authentication, patient historical data management, appointments definitions, diseases reports, audit controls and data encryption.

We observed the extension of files that interact with the source code in some way from each system, as described in Table 3. For source code analysis we took into account .java .js and .jsp extensions. Such files contain statements written in java and javascript, and also html and custom tags; .xml and .properties files have relevant information about constants and variables that are used within the source code, hence were also relevant for our purpose. Finally, .sql files contain sql sentences with significant information about data and entities related to the application domain and are referenced in the source code.

Table 3. Summary of source code artifacts that were taken as the corpus of the traceability algorithm

	iTrust	OpenMRS	OSCAR	TAPAS
JAVA	936	1545	3810	221
JSP	266	447	1655	0
JS	3636	416	875	0
XML	14	820	193	14
SQL	192	2	711	0
PROPS	7	39	39	2
TOTAL SC ARTIFACTS	5051	3269	7283	237

3.2 Extracting Requirements from High Level Artifacts

Since we want to associate high level requirements with source code, one of the problems that we had to face is the extraction of specific requirements from the available documentation on each health care system.

We setted apart several phrases and paragraphs from requirements explicitly declared in the software high level artifacts (query of our algorithm). Then, we extracted software requirements embedded in sentences from the documentation of all four systems, according to the next format:

$$[Article] + [Subject] + [ObligationVerb] + [Complement]$$

The extracted requirements were textually taken from documentation without any modification. In some cases, following the precise definition of the format leads to the extraction of phrases with no sense or context. For that reason, in such particular cases, we included in the extraction additional information (i.e. terms that were not taken into account when the filter was applied) from the surrounding text of such phrases complete them. Table 4 shows some examples of specific requirements extracted from artifacts of each system.

Finally, we selected all the extracted requirements that were more suitable to trace into code according to the proposed taxonomy of HIPAA Security and Privacy regulations. To put it another way, we associated each extracted requirement with a HIPAA statute and focused on those in the categories (A) and (B) of the proposed classification (Table 2).

4 Fine Grained Traceability Algorithm

Different approaches have been adopted to conduct a traceability link recovery process. In order to support regulatory compliance verification at a granular level, we proposed a technique that involve information retrieval (IR) techniques, static code analysis and heuristics derived from observations after a process of manual analysis of source code implementations.

The proposed algorithm takes advantage of search algorithms developed in information retrieval techniques as well as heuristic derived from observations and static analysis of source code. Figure 1, shows a summary of our proposed technique.

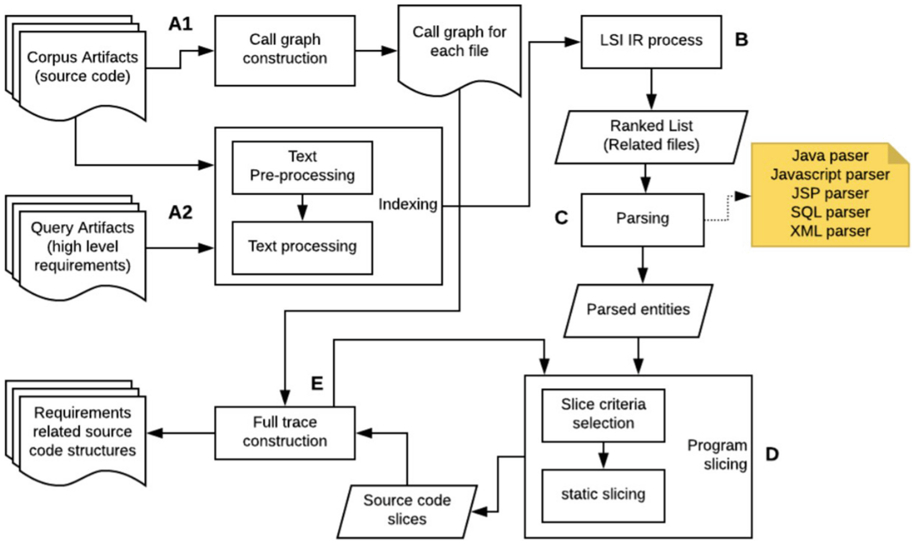

Fig. 1. Proposed fine-grained traceability link recovery process

4.1 A1: Call Graph Construction from Source Code Files

The files and artifacts that contain the source code and relevant structures present in the implementation of a system are all of a very different nature. Java classes are constituted by attributes and methods, javascript code files are batch processing files that are stored in plain text, JSP files are a combination of java code, javascript code, tomahawk and html tags. The process of A1 as depicted in Fig. 1, can be summarized in the generation of a plain text file with the information of methods headers (i.e. access modifier [optional], return type, name, parameter list, exception throws [optional]) and their path in the different source code artifacts.

4.2 A2: Indexing Phase

Before going through the algorithm of information retrieval, several processing tasks are performed on the text for both the corpus and the query. The query is a list of requirements writing in natural human language to be traced into the corpus, which is the source code of a selected system.

When dealing with two entries of a different nature, the treatment of the terms that make up the corpus and the query is essentially different in both cases. With respect to the treatment performed on the query, tokenization, stemming, stopword removal, part of speech (a.k.a. POS) tagging, punctuation removal, number spell out and word embedding [3, 5, 8, 16, 25] task are performed. On the other hand, the tasks carried out with the terms that make up the corpus include stop word removal, splitting by camel case, punctuation removal and tokenization.

4.3 B: Information Retrieval Process

Once the corpus and the query are indexed, they are treated as entries for the information retrieval algorithm to obtain a ranked list of the possible corpus files that are most likely related to the text of the query requirements.

After executing the algorithm of information retrieval, k files are obtained in order of relevance for each requirement; that is, if the query file contains a set of n requirements in total, the list of ranked links will contain a total of kn most related files to each entry in the query file.

The determination of the number of links taken into account (k) for each requirement was determined empirically. In other words, after several tests of execution we determined that the optimal value for k fluctuates between 25 and 30 files per requirement. This number is sufficient to obtain the most strictly relevant files, ignoring those that are least related to requirements.

4.4 C: Parsing Phase

Once the list of possible files more related to a given requirement based on textual similarity is obtained, the next phase of the algorithm consists of analyzing source code implementations and perform abstractions through program slicing, to find structures at source code level that are related to the implementation of a particular requirement. In order to correctly extract different structures within the code, we use parsers for each type of file that was included in the corpus file. The parsers statically analyze code elements present in .java, .js, .jsp, .xml, .sql, .properties and plain text files. Parsers receive as input a source code fragment and give us an Abstract Syntax Tree (AST) representation of it in a way that make it easier to process.

4.5 D: Program Slicing Process

After executing the parsers for each type of source code file, it is possible to extract code fragments of interest since we are able to perform an exhaustive static analysis of source code. For each of the files that were identified by the information retrieval algorithm (LSI), silicing criteria were defined taking into account the subjects present in the description of each requirement. Within the source code there are terms (e.g. subjects, adjectives, verbs) of the problem context and therefore it is possible to define slicing criteria from the names of variables, objects, classes and methods. To illustrate this point, let's assume that a requirement r1 contains a total of 4 subject names; so it is possible to define 4 slicing criteria to conduct a program slicing process in each file obtained from the retrieval information algorithm for r1 after executing the process defined in B (Fig. 1). For each source code file, a slicing process is performed to obtain the code lines within each file that affect or are affected by a particular criterion. The result of this operation is a set of slices for each file ranked in B.

4.6 E: Full Trace Construction Phase

The information given by the process of abstraction of the source code through program slicing is not enough to build entirely the trace of a requirement. To put it another way, consider a slice of source code obtained after applying the process defined in D; it is highly likely that such fragment of code contains calls to other methods within the system that do not necessarily contain terms in common with a high level requirement but that fulfill a very important role in the implementation. This problem can also be seen as a consequence of delegation.

In this phase of the algorithm the fragments of code obtained in the slices are analyzed and, for each call to an external method that is not defined in the parent file, a search process is carried out according to the definition header of the method that is invoked with the support of the mapping file obtained in A1. Each of the methods that are referenced are also included in the trace of the requirement and recursively, a slicing process D is performed.

To carry out the process of evaluation and testing of our approach, we developed a tool in Java that implements the traceability technique previously exposed. It was named Fine Grained Traceability Hunter (FGTHunter). We implemented a set of tools designed for static analysis of source code, as well as information retrieval algorithms.

Table 4. Some specific requirements extracted from the available documentation of selected healthcare systems

ID	Requirement	System	Documentation
ITRUST-1	"An HCP is able to create a patient [S1] or disable a selected patient [S2]. The create/disable patients and HCP transaction is logged (UC5)"	iTrust	http://agile.csc.ncsu.edu/iTrust/wiki/doku.php?id=requirement

(*continued*)

Table 4. (*continued*)

ID	Requirement	System	Documentation
OSCAR-1	"Your password is stored in an encrypted format such that even the system administrator cannot find out what it is. If you have forgotten your user name and/or password, your administrator can reset the password for you but he/she cannot tell you what the original password was"	OSCAR	http://oscarmanual.org/oscar_emr_12/General%20Operation/access-preferences-and-security/accessing-oscar
TAPAS-1	"The system must have the ability to manage users in the system. Like clinical data, users should not be able to be deleted from the system as they will be tied to activities in a record"	TAPAS	http://tap-apps.sourceforge.net/docs/use-cases.html#UCSYSADM-01

5 Results

We executed the traceability algorithm implemented in the FGTHunter tool for each of the open source systems. We collected the results and evaluated the precision of our technique to find traceability links between high level artifacts and source code lines. The algorithm was able to successfully find a large number of the high-level requirements filtered for this study, along with new traceability links that were not originally found by the manual construction of traces.

To evaluate the performance of our technique, we calculated the F1 score for each extracted requirement that was associated with a HIPAA statute; then, we calculated the average of F1 scores obtained in each statute. Results were grouped for each of the analyzed systems. Table 5 summarizes our findings.

The audit management is a fundamental part of any software system that aims to keep track of all the operations that are performed by every actor that use the system. Table 5 shows the average of the harmonic mean (average of F1 score) for the requirements in each healthcare system that are related to HIPAA statute 164.312 (b). The precision of FGTHunter for iTrust and OSCAR systems had an approximate value of 0.4 whereas in TAPAS the precision was much lower. In the process of observation that was made when performing the manual code inspection, we noticed that very few code lines in TAPAS handled the audit control; this may explain the low precision for this case.

The access control of the information within a system ensures the correct manipulation of the data. In the four systems that we analyzed, the access of the information was restricted by the definition of roles and user permissions to see, read, modify or eliminate the data. Table 5 shows the F1 score means of FGTHunter obtained after executing the algorithm to find the lines of code that most are related to the requirements associated with the statute 164.312 (a) (1) of HIPAA in each system. The

Table 5. Summary of results for all healthcare systems analyzed in this study.

HIPAA statute	Description	Average F1 score
164.312(b)	"Implement hardware, software, and/or procedural mechanisms that record and examine activity in information systems that contain or use electronic protected health information"	iTrust: 0,39 OSCAR: 0,39 TAPAS: 0,15
164.312(a)(1)	"Implement technical policies and procedures for electronic information systems that maintain electronic protected health information to allow access only to those persons or software programs that have been granted access rights as specified in §164.308(a)(4)"	iTrust: 0,25 OSCAR: 0,39 OPENMRS: 0,66 TAPAS: 0,39
164.312(a)(2)(i)	"Assign a unique name and/or number for identifying and tracking user identity"	iTrust: 0,43 OSCAR: 0,65 OPENMRS: 0,77
164.312(d)	"Implement procedures to verify that a person or entity seeking access to electronic protected health information is the one claimed"	iTrust: 0,6 OSCAR: 0,87 TAPAS: 0,46
164.312(a)(2)(iii)	"Implement electronic procedures that terminate an electronic session after a predetermined time of inactivity"	iTrust: 0,30 OSCAR: 0,32 TAPAS: 0,46
164.308(a)(5)(ii)(C)	"Procedures for monitoring log-in attempts and reporting discrepancies"	iTrust: 0,48 OSCAR: 0,73 OPENMRS: 0,66
164.308(a)(1)(ii)(D)	"Implement procedures to regularly review records of information system activity, such as audit logs, access reports, and security incident tracking reports"	iTrust: 0,4 OSCAR: 0,39
164.312(e)(2)(i)	"Implement security measures to ensure that electronically transmitted electronic protected health information is not improperly modified without detection until disposed of"	iTrust: 0,41
164.308(a)(5)(ii)(D)	"Procedures for creating, changing, and safeguarding passwords"	iTrust: 0,53 OSCAR: 0,69
164.312(a)(2)(iv)	"Implement a mechanism to encrypt and decrypt electronic protected health information"	OSCAR: 0,83
164.308(a)(7)(ii)(A)	"Establish and implement procedures to create and maintain retrievable exact copies of electronic protected health information"	OSCAR: 0,96 TAPAS: 0,58
164.308(a)(4)(i)	"Establish and implement procedures to create and maintain retrievable exact copies of electronic protected health information"	TAPAS: 0,32
164.312(c)(2)	"Implement electronic mechanisms to corroborate that electronic protected health information has not been altered or destroyed in an unauthorized manner"	TAPAS: 0,63
164.312(c)(1)	"Implement policies and procedures to protect electronic protected health information from improper alteration or destruction"	TAPAS: 0,51

average harmonic mean has an acceptable level (greater than 0.4 points) in OPENMRS, OSCAR and TAPAS; on the other hand, for iTrust the algorithm had the lowest performance. According to our observations, the access control by roles in iTrust was handled by access restrictions defined in the configuration .xml files for the TOMCAT server.

Assigning a unique identifier to each entity of a system ensures the correct manipulation of the data and facilitates the control of the information integrity. According to our observations, the restrictions of non-repetition were always defined at the database level, that is, the traceability of the requirements associated with the Statute 164.312 (a)(2)(i) of HIPAA was carried out analyzing .sql files with thousands of code lines impacting negatively on the precision of our technique. However, according to the results, for OSCAR and OPENMRS systems the algorithm reaches its highest performance, probably because at the application level, the uniqueness of the identifiers for each entity was also validated.

Ensuring the correct implementation of access control mechanisms is a fundamental aspect in any system that manipulates critical information. Such control strategies range from verification of passwords and access codes to role management within the system. Considering the wide spectrum of artifacts in each system that may be related to HIPAA Statute 164.312 (d), it is natural that the accuracy tend to be very high; however in TAPAS that value is low, maybe because the access control in this application is not clearly defined at the application level (i.e. there is no login forms or access restrictions) and externally they must control the access to the information by applying restrictions policies.

Finishing the session after a period of inactivity is a measure of additional protection that usually exists to avoid improper manipulation of the data. The session time limit for a given user is usually specified in a particular line of code within the application, either through a property file, a database record or a global variable. Considering the quantity of artifacts analyzed and the reduced number of code lines in which the requirements related to statute 164.312 (a)(2)(iii) of HIPAA are implemented, for iTrust and OSCAR the level of precision was very low. According to Table 5, TAPAS is the exception probably because in many parts of the code the mechanisms of termination of sessions are repeated, once a particular operation has started.

As a security measure, many information systems keep a record of unsuccessful attempts prior to login for a particular service. Depending on the number of failed attempts, an account is blocked for a period of time or indefinitely until an administrator decides to unblock it. Table 5 summarizes the traceability results for the requirements related to HIPAA Statute 164.308 (a)(5)(ii)(C). iTrust was the system with less precision in this aspect, probably because there are very few code lines where the control of failed attempts is made.

When a user of the system views or edits the information of a particular patient, the audit information should be able to be consulted to follow the detail of the modifications in critical data, as is established by the statute 164.308 (a)(1)(ii)(D) of HIPAA. Table 5 shows the average of the harmonic mean for the systems that implemented control mechanism of visualization for audit data. In general, the performance of the traceability algorithm was acceptable, above 0.4.

As a measure of securing information in the event of a disaster, data security should be periodically generated from the information contained in the database in order to be effectively restored. The HIPAA statute 164.308 (a)(7)(ii)(A) refers to this topic. OSCAR was the system in which our algorithm reached a higher precision, while TAPAS was the opposite case. This particular behavior can be proved when we compare the source code artifacts from both systems that were involved in the implementation of such statute, from our findings in the GOLDSET we can observe that OSCAR contains more lines of code associated with data backups and database restoration than TAPAS.

6 Conclusions

Traceability in software systems is a necessary and important process that must be conducted from the beginning of any project where a large number of people work and delegate tasks. Especially in contexts where it is required to follow strict and critical rules to ensure the correct treatment of data in sensitive information, such is the case of medical health systems that obey written legal regulations such as HIPAA.

From the results it was possible to appreciate that the precision of FGTHunter obtained an acceptable performance (in general, more than 0,4 F1 score mean for each requirement) when in the code there were present good design practices and conventions to name entities according to the context of the problem.

Techniques of static code analysis play a predominant role in the construction of a totally autonomous technique to recover traceability links between high level requirements and source code artifacts. An information recovery technique alone would not achieve a level of precision adequate enough to build the full trace of a requirement.

The restrictions and standards written in legal texts are usually described in a very technical language and detached from a particular context. For this reason, as a step prior to the execution of the algorithm, it was necessary to filter and extract official documentation requirements for each software system that had a relationship with some HIPAA statute within the taxonomy that we defined.

Although it is not a mandatory task in the execution of our technique, when starting from requirements that are defined from a specific context (i.e. system domain) the precision of the technique would improve.

Regulations related to audit control standards and session expiration in the implementation of healthcare systems were in general terms the ones that achieved less precision when analyzed with our traceability algorithm (over 0,35 F1 score mean for each related system). Very few lines and source code structures related with these requirements were successfully mapped by our algorithm for each system; This may be explained probably due to the few places within the source code where these requirements were implemented, increasing in that way the recall of our technique and decreasing the precision. With respect to the other statutes defined in the privacy and security rule of HIPAA, the performance of our algorithm was acceptable (over 0,4 F1 score mean) and in some cases excellent (over 0,8 F1 score mean).

Although, it is very difficult to reach a perfect degree of precision in a traceability technique based on an information retrieval algorithm without the help of an external person to delimit the list of ranked links, our proposal would undoubtedly facilitate the work of those people who seek to certify a system compliance with rules and standards at a source code level, as in the case of HIPAA.

7 Future Work

Our technique was designed for projects written in JAVA EE; future work may require to improve our traceability algorithm so that it would be generic in any programming language independently of the syntax and particular rules. For this purpose we could use controlled techniques of machine learning, and train models constantly with the new trends in programming and design patterns. We could also follow the original approach of our technique and define syntactic analyzers for each programming language in such a way that the source code slicer and code analysis algorithms can be applied.

To evaluate the performance of our technique in other regulations different to HIPAA, we would have to conduct an adequate study of the structure of such regulations and find software systems whose source code and documentation is available for analysis.

An empirical study to assess the performance of our tool with real users and regulation reviewers would give us an important feedback for real situations. This point should be boarded in future improvements for our algorithm.

References

1. Health Insurance Portability and Accountability Act of (1996)
2. Huang, J., Gotel, O., Zisman, A. (eds.): Software and Systems Traceability, p. 27. Springer, London (2012). https://doi.org/10.1007/978-1-4471-2239-5
3. HIPAA Compliance & Enforcement (2018). http://www.hhs.gov/hipaa/for-professionals/compliance-enforcement/. Accessed 27 Sept 2018
4. Home - Documentation - OpenMRS Wiki (2018). https://wiki.openmrs.org/. Accessed 11 Nov 2018
5. iTrust — Medical Free/Libre and Open Source Software (2018). www.medfloss.org/node/542. Accessed 11 Nov 2018
6. OSCAR EMR — Site (2018). http://oscarmanual.org/oscar_emr_12. Accessed 11 Nov 2018
7. start [iTrust] (2018). https://152.46.18.254/doku.php. Accessed 11 Nov 2018
8. TAPAS Home (2018). http://tap-apps.sourceforge.net/docs/srs.html. Accessed 11 Nov 2018
9. Alshugran, T., Dichter, J.: Extracting and modeling the privacy requirements from HIPAA for healthcare applications (2014)
10. Antoniol, G., Canfora, G., Casazza, G., Lucia, A.D.: Information retrieval models for recovering traceability links between code and documentation, San Jose, CA, pp. 40–49 (2000)
11. Antoniol, G., Canfora, G., Casazza, G., Lucia, A.D., Merlo, E.: Tracing object-oriented code into functional requirements. In: Program Comprehension, Proceedings, Limerick, pp. 79–86 (2000)
12. Antoniol, G., Canfora, G., Casazza, G., Lucia, A.D., Merlo, E.: Recovering traceability links between code and documentation. IEEE Trans. Softw. Eng. **28**(10), 970–983 (2002)

13. Antoniol, G., Canfora, G., Lucia, A.D., Merlo, E.: Recovering code to documentation links in OO systems. In: Reverse Engineering, Atlanta, GA, pp. 136–144 (1999)
14. Avancha, S., Baxi, A., Kotz, D.: Privacy in mobile technology for personal healthcare. ACM Comput. Surv. **3**(1), 1–3 (2012)
15. Breaux, T., Antón, A.: Analyzing regulatory rules for privacy and security requirements. IEEE Trans. Softw. Eng. **34**(1), 5–20 (2008)
16. Breaux, T.D., Antón, A.: A Systematic Method for Acquiring Regulatory Requirements: A Frame-Based Approach (2007)
17. Breaux, T.D., Vail, M.W., Anton, A.I.: Towards Regulatory Compliance: Extracting Rights and Obligations to Align Requirements with Regulations (2006)
18. Capobianco, G., Lucia, A.D., Oliveto, R., Panichella, A., Panichella, S.: Improving IR-based traceability recovery via noun-based indexing of software artifacts. J. Softw. Evol. Proc. **25**(7), 743–762 (2013)
19. Dagenais, B., Robillard, M.P.: Recovering traceability links between an API and its learning resources (2012)
20. Deerwester, S., Dumais, S.T., Furnas, G.W., Landauer, T.K., Harshman, R.: Indexing by latent semantic analysis. J. Am. Soc. Inf. Sci. **41**(6), 391–407 (1990)
21. Diaz, D., Bavota, G., Marcus, A., Oliveto, R., Takahashi, S., Lucia, A.D.: Using code ownership to improve IR-based Traceability Link Recovery (2013)
22. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature Location in Source Code: A Taxonomy and Survey (2011)
23. Dumais, S.T.: Improving the retrieval of information from external sources. Behav. Res. Methods Instr. Comput. **23**(2), 229–236 (1991)
24. Fasano, F.: Fine-Grained Management of Software Artefacts, Paris (2007)
25. Goldberg, Y., Levy, O.: word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method, arXiv:1402.3722 [cs, stat] (2014)
26. Gotel, O.C.Z., Finkelstein, C.W.: An analysis of the requirements traceability problem. In: Requirements, pp. 94–101. Springs, CO (1994)
27. Kiyavitskaya, N., et al.: Automating the extraction of rights and obligations for regulatory compliance. In: Li, Q., Spaccapietra, S., Yu, E., Olivé, A. (eds.) ER 2008. LNCS, vol. 5231, pp. 154–168. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87877-3_13
28. Lucia, A.D., Penta, M.D., Oliveto, R., Panichella, A., Panichella, S.: Improving IR based Traceability Recovery Using Smoothing Filters (2011)
29. Marcus, A., Maletic, J.I.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In: Software Engineering. Proceedings, pp. 125–135 (2003)
30. Maxwell, J.C., Antón, A.I.: Checking Existing Requirements for Compliance with Law Using a Production Rule Model (2009)
31. Palomba, F., et al.: User reviews matter! Tracking crowdsourced reviews to support evolution of successful apps (2015)
32. Qusef, A., Bavota, G., Oliveto, R., Lucia, A.D., Binkley, D.: Recovering test-to-code traceability using slicing and textual analysis. J. Syst. Softw. **88**, 147–168 (2014)
33. Ramesh, B., Jarke, M.: Toward reference models for requirements traceability. IEEE Trans. Softw. Eng. **27**(1), 58–93 (2001)
34. Sharif, B., Maletic, J.I.: Using fine-grained differencing to evolve traceability links. In: TEFSE/GCT 2007, pp. 76–81, March 2007
35. Shen, W., Lin, C.L., Marcus, A.: Using traceability links to identifying potentially erroneous artifacts during regulatory reviews (2013)

36. Wong, W.E., Gokhale, S.S., Horgan, J.R., Trivedi, K.S.: Locating program features using execution slices, pp. 194–203 (1999)
37. Yadav, V., Joshi, R.K.: Evolution traceability roadmap for business processes, vol. 20, pp. 1–20. ACM, New York (2019)
38. Zeni, N., Mich, L., Mylopoulos, J., Cordy, J.R.: Applying GaiusT for extracting requirements from legal documents (2013)