Science of Computer Programming 121 (2016) 153–175

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Tracing known security vulnerabilities in software repositories – A Semantic Web enabled modeling approach

Sultan S. Alqahtani, Ellis E. Eghan, Juergen Rilling *

ARTICLE INFO

Article history:
Received 1 April 2015
Received in revised form 23 January 2016
Accepted 27 January 2016
Available online 4 February 2016

Keywords:
Semantic knowledge modelling
Semantic web
Software security vulnerabilities
Software traceability
Impact analysis

ABSTRACT

The introduction of the Internet has revolutionized not only our society but also transformed the software industry, with knowledge and information sharing becoming a central part of software development processes. The resulting globalization of the software industry has not only increased software reuse, but also introduced new challenges. Among the challenges, arising from the knowledge sharing is Information Security, which has emerged to become a major threat to the software development community, since not only source code but also its vulnerabilities are shared across project boundaries. Developers are unaware of such security vulnerabilities in their projects, often until a vulnerability is either exploited by attackers or made publicly available by independent security advisory databases. In this research, we present a modeling approach, which takes advantage of Semantic Web technologies, to establish traceability links between security advisory repositories and other software repositories. More specifically, we establish a unified ontological representation, which supports bi-directional traceability links between knowledge captured in software build repositories and specialized vulnerability database. These repositories can be considered trusted information silos that are typically not directly linked to other resources, such as source code repositories containing the reported instances of these problems. The novelty of our approach is that it allows us to overcome some of these traditional information silos and transform them into information hubs, which promote sharing of knowledge across repository boundaries. We conducted several experiments to illustrate the applicability of our approach by tracing existing vulnerabilities to projects which might directly or indirectly be affected by vulnerabilities inherited from other projects and libraries.

1. Introduction

The Internet has revolutionized our society and impacted the software industry [1], with knowledge and information sharing becoming a central part of software development, facilitating the globalization of the software industry [1]. This change in the information flow removes traditional project boundaries and promotes a free flow of information, resources and knowledge across projects. Globalization in the software industry [2] can have several facets ranging from out- and crowd-sourcing parts of a development process, the wide spread use of collaborative environments facilitating resource sharing, to the introduction of completely new software development paradigms such as open source. Open source software publishes source code and other related artifacts on the Internet using specialized code and artifact sharing portals such

* Corresponding author.
E-mail addresses: s_alqaht@encs.concordia.ca (S.S. Alqahtani), e_eghan@encs.concordia.ca (E.E. Eghan), juergen.rilling@concordia.ca (J. Rilling).

http://dx.doi.org/10.1016/j.scico.2016.01.005
0167-6423/© 2016 Elsevier B.V. All rights reserved.

as Sourceforge,[1] GitHub,[2] and Maven,[3] allowing these artifacts to be shared and reused globally. This reuse can take on different forms, such as integrating open source projects into existing software ecosystems (e.g., reuse of code libraries) or extending and customizing available projects to meet specific requirements, e.g., creating specialized Linux distributions [3].

Shared knowledge resources not only facilitate reuse and collaboration, they also introduce new challenges to the software engineering community. Knowledge and resources are no longer controlled by a single project or organization and instead are now distributed across multiple projects, organizations or even global software ecosystems. Given this new distributed nature of software systems and their resources, traditional analysis approaches and tools, developed for a project level context, no longer scale to the new global software context. Among the challenges arising from the knowledge sharing is Information Security (IS), which has emerged to become a major threat to the software development community. At its core, IS promotes that one should consider different security concepts (e.g. secure programming, knowledge about software security vulnerabilities and their analysis) in the development process. The importance of IS for the software community is reflected by the fact that it has become an integrated part of current software engineering best practices [4]. Different specialized software security and advisory databases (SSD) (e.g., National Vulnerability Database (NVD) [5]) have been introduced to track software known vulnerabilities and potential solutions to resolve them. These SSD can be seen as a direct response by the software industry to the ever-increasing number of software attacks, which no longer are limited to a particular project or computer but often affect now hundreds of different systems and millions of computers.

Software security repositories, like most other software repositories, can be considered trusted information silos that focus on modeling and managing specialized resources or knowledge. Software security repositories are typically not linked to other knowledge resources, such as source code repositories containing reported instances of these problems or project specific issue trackers. There are several reasons why the software community is still dealing with such information silos: 1) Establishing traceability among repositories is often difficult due to the lack of a common, standardized approach for modelling knowledge across repository boundaries [6]. 2) Reasoning and semantic integration are a challenge, since most repositories lack the required expressiveness (e.g. First Order or higher order logics) in their underlying knowledge modeling approach to support automated reasoning [7]. 3) Using relational modeling schemata, facts and schemata remain machine but not human accessible without additional tool support for extracting schemata information, making knowledge sharing more difficult. 4) While significant progress has been made by the mining software repository research community to identify potential (semantic) links among repositories by introducing handcrafted, specialized (trained) machine and data mining techniques within proprietary datasets, a key challenge remains – the results and information obtained remain still not reusable or shareable for later consumption by either humans or machines [8].

Given the growing importance of IS for the software domain and the challenges the software community faces in integrating software repositories, the paper introduces a Semantic Web enabled modeling approach which addresses this IS knowledge integration challenge. While our modeling approach supports the integration of a broad range of traditional software repositories (e.g., issue trackers, version control systems, Q&A repositories), we focus in this research in particular on how we can capture knowledge from specialized IS repositories (e.g., NVD) and seamless integrate this knowledge with one of the more widely used software build repositories (e.g., MAVEN). For our research, we take advantage of the Semantic Web and its supporting technologies such as ontologies and Linked Data to establish a common, standardized unified representation to integrate knowledge resources across existing repository boundaries. In addition, this formal representation allows us take advantage of Semantic Web reasoning services to infer both explicit and implicit knowledge to enrich the already modeled knowledge.

This unified knowledge model allows us to identify potential impact of vulnerabilities on software systems across project boundaries. For example, by linking the NVD security database to the Maven build repository we were able to show that not only 576 projects in Maven directly include known security vulnerabilities, but also that due to the ripple effect more than 400,000 projects or libraries might be potentially affected.

The key benefits of our research are manifold, including the ability to support:

- Transformation of traditional security and vulnerability information silos into information hubs.
- Matching of different ontologies through their shared concepts and semantic linking to support integration of knowledge.
- Identification of potential rippling impact of vulnerabilities across project boundaries based on project build dependencies.

The remainder of this paper is structured as follows: Section 2 introduces the motivation for our research. Section 3 describes in more detail background relevant to our research. Section 4 describes the approach used. Section 5 presents our experiments and findings. We discuss observations regarding our integration approach, and outline the potential threats to the validity of this approach in Section 6. Section 7 discusses relevant work, followed by Section 8, which discusses future work and concludes the paper.

[1] http://sourceforge.net/about.
[2] https://github.com/about.
[3] http://search.maven.org/.

Table 1
Derby versions and their dependent projects in Maven.

Derby version	Release year	Reported vulnerabilities in NVD	Direct dependencies in Maven Repository
10.1.1.0	2005	3	382
10.5.3.0	2009	1	0
10.11.1.1	2014	0	36

2. Problem statement

2.1. Illustrative example

Context: over the last decade, reuse and integration of open source components through API usage by third party systems has gained on importance in order to reduce development and maintenance cost.

Motivation: The following example illustrates how a combination of component reuse through API usage and a lack of traceability links between heterogeneous knowledge resources can not only impact trust in a system implementation but also the trust in API recommendations. In what follows, we introduce two scenarios to motivate our work. Both examples involve Apache Derby,[4] an open source DBMS implemented entirely in Java and is often embedded in other projects to support online transaction processing.

Scenario#1: **Missing knowledge integration**. A significant body of research exists in recommending APIs to developers. For example, in the work by Mileva et al. [9], the authors explicitly recommended developers to use an older version of Apache Derby (version 10.1.1.0) given its widespread usage/popularity rather than migrating to a newer version. However, like any software project, Apache Derby is also susceptible to security vulnerabilities. By recommending the older version of Derby (version 10.1.1.0.), the authors recommended a component that contained three known reported (in NVD) and unpatched security vulnerabilities (Table 1), rather than the newer version 10.5.3.0 which only contains one vulnerability.

Scenario#2: **Tracing vulnerabilities across project boundaries**. Establishing traceability links between heterogeneous software repositories allows for improved analysis and knowledge inference within projects and across project boundaries. In this scenario, we illustrate how one can take advantage of traceability links between the NVD repository and the Maven build repository, to determine the possible impact of a vulnerable component on other projects. Table 1 shows the number of projects in Maven that have a direct build dependency on different versions of Apache Derby. It should be noted that for the identification of dependent projects, we did not distinguish if a project actually makes use of a vulnerable Derby component. Nevertheless, the example shows that by being able to trace vulnerable components back to a build repository, maintainers and security experts can now benefit from applications such as impact analysis of vulnerabilities even for a global software ecosystem scale.

2.2. Research questions

In this paper, we aim to answer the following research questions:

- **RQ1**: To what extent are users/developers of open source components directly susceptible to known security vulnerabilities?
- **RQ2**: Can projects that are indirectly dependent on vulnerable components be identified?
- **RQ3**: Can a more formal knowledge representation be beneficial in determining transitive dependencies of vulnerable components in software ecosystems?

In what follows, we introduce background principles and our research methodology, which we used to establish traceability links between Maven and NVD project information in order to answer our research questions.

3. Background

3.1. The Semantic Web and ontologies in a nutshell

The term "ontology" originates from philosophy, where it denotes the study of existence. In computer science, a widely accepted definition has been introduced by Studer [10]: "An ontology is a formal, explicit specification of a shared conceptualization." Ontologies are typically used as a formal and explicit way to specify concepts and relationships in a domain of discourse. They can overcome portability, flexibility, and information sharing problems associated with databases [11].

[4] db.apache.org/derby/.

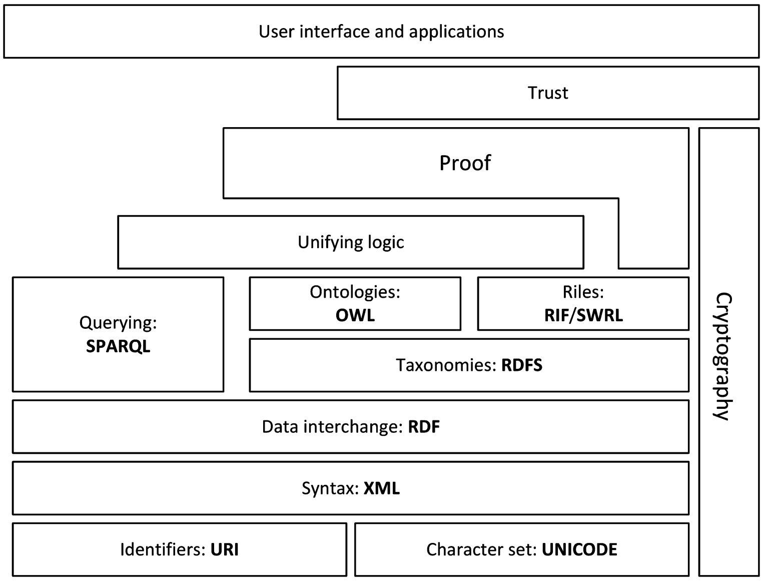


Fig. 1. Semantic Web architecture in layers [16].

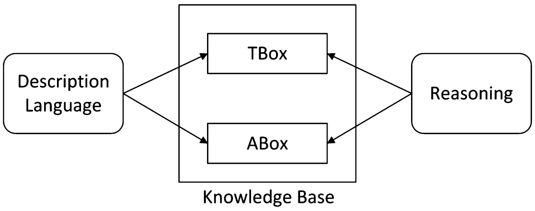


Fig. 2. Description logics system.

Compared to relational approaches, which assume complete knowledge (closed world assumption), ontologies support the modeling of incomplete knowledge (open world assumption) and the extendibility of the ontological model.

The Semantic Web (SW) allows for machine understandable Web resources that can be shared and processed by both software tools (e.g., search engines) and humans [12]. Ontologies are an important foundation of the SW, as they allow knowledge to be shared between different agents and the creation of common terminologies for understanding [12]. Moreover, the resulting data representation format becomes reusable rather than being proprietary to specific tasks. The current data-model used to represent meta-data in SW is the Resource Description Format (RDF) [13]. RDF is used to formalize meta-models in form of $<$ subject, predicate, object $>$, which are called triples. RDF triples make statements about resources, with a resource in the SW being anything: a person, project, software, a security bug, etc. In order to make triples persistent, RDF triples stores are used, with each triple being identified by an Uniform Resource Identifier (URI) [13].

The Web Ontology Language (OWL) [14] is used on top of the RDF layer (see Fig. 1). It is a standard modeling language put forward by the W3C[5] to pursue the vision of the SW. OWL provides for machine understandable (i.e., capturing semantics) information, allowing Web resources to be automatically processed and integrated. The widely used OWL sub-language OWL-DL is based on Description Logics (DLs) [15].

Description Logic (DL): A DL based knowledge representation system provides typical facilities to set up knowledge bases and to reason about their content [12]. Fig. 2 illustrates a typical Description Logic based knowledge system.

Such a knowledge base (KB) consists of two components – the TBox contains the *terminology*, i.e. the vocabulary of an application domain, and the ABox contains *assertions* about named individuals in terms of this vocabulary. The terminology is specified using description languages introduced previously in this section, as well as *terminological axioms*, which make statements about how concepts or roles are related with each other. In the most general case, terminological axioms have the form:

$$\mathrm{C} \sqsubseteq \mathrm{D}\ (\mathrm{R} \sqsubseteq \mathrm{S}) \text{ or } \mathrm{C} \equiv \mathrm{D}\ (\mathrm{R} \equiv \mathrm{S})$$

Where C and D are concepts (R and S are roles). The semantics of axioms is defined as: an interpretation **I** satisfies $\mathrm{C} \sqsubseteq \mathrm{D}$ ($\mathrm{R} \sqsubseteq \mathrm{S}$) if $\mathbf{CI} \subseteq \mathbf{DI}$ ($\mathbf{RI} \subseteq \mathbf{SI}$). A Tbox, denoted as **T**, is a finite set of such axioms. The assertions in an ABox are specified using

[5] W3c semantic web activity. W3C, 1994, www.w3.org.

Table 2
Repository statistics.

Statistics for the central repository	
Data index last refreshed	2015-09-02 15:14:21 UTC
Total number of artifacts indexed (GAV)	1,054,470
Total number of unique artifacts indexed (GA)	16,160
Current size of repository on disk	1,761,203 MB

concept assertions and role assertions, which have the form C(a), R(a, b), where C is a concept, R is a role, and a, b are names of individuals. The semantics of assertions can be given as: the interpretation **I** satisfies the concept assertion C(a) if a**I** $\in$ C**I**, and it satisfies the role assertion R(a, b), if (a**I**, b**I**) $\in$ R**I**. An Abox, denoted as **A**, is a finite set of such assertions.

A Description Logic system not only stores terminologies and assertions, but also offers services that allow to *reason* about them. Typical reasoning services for a TBox are to determine whether a concept is *satisfiable* (i.e. non-contradictory), or whether one concept is more general than another one (i.e. *subsumption*). Important reasoning services for an ABox are to find out whether its set of assertions is *consistent*, and whether the assertions in an ABox entail that a particular individual is an *instance* of a given concept description.

A DL knowledge base might be embedded into an application, in which some components interact with the KB by querying the represented knowledge and by modifying them, i.e. by adding and retracting concepts, roles, and assertions. However, many DL systems, in addition to providing an application programming interface that consists of functions with a well-defined logical semantics, provide an escape hatch by which application programs can operate on the KB in arbitrary ways [12].

In addition to RDF, OWL and OWL-DL, the Semantic Web community provides tools to process OWL semantics and RDF data. Jena [17] emerged as a Java framework for building applications and providing a programmatic environment for RDF and OWL. Reasoners (e.g., RDFS $++$,[6] Pellet[7]) can infer new facts about the designed ontology and form a set of asserted axioms. RDF databases, such as Virtuoso [18], Allegrograph [19], are used to materialize and store RDF triples. SPARQL is an RDF query language, that is, a semantic query language for databases, able to retrieve and manipulate data stored in RDF format.

The Semantic Web has been designed from ground up to address the integration challenge, traditional relational databases are facing, such as [20]: (1) The Semantic Web facilitates the creation of taxonomies using ontologies, which can be shared across applications and domains, which is in contrast to relational database, where schemata sharing and reuse is not natively supported [20]. (2) Semantic Web meta-models are extensible, allowing new knowledge to be added without affecting existing knowledge, unlike relational databases, where extending the schema becomes a time consuming operation, affecting often a complete database. For example, a change of index type (foreign key) might require dropping and recreating several other dependent database indices. (3) The Semantic Web makes relations explicit. In contrast, relational databases do not provide a consistent method to obtain the semantic of a relation. A query can join any two table columns, as long as their datatypes match – there is no interpretation of the meaning of the actual relation performed. As a result, relational databases are not machine interpretable, and the inference of knowledge (explicit or implicit) requires human interaction. (4) Linking data is a key property of the Semantic Web, with any resource being identified by its Uniform Resource Identifier (URI). These URIs, allow for a consistent identification of the same resource across various knowledge resources. This is in contrast to relational databases where resources are local and not universal, restricting the ability of relational databases to establish resource links outside their own local schema.

3.2. Maven

Maven, hosted by the Apache Software Foundation, is an open-source build automation tool used primarily for Java projects. In Maven, a software project defines its dependence on any of its artifacts as part of its xml configuration file (also called the POM file), which is stored in the central repository. Upon the build of a project, Maven dynamically downloads all the required Java libraries and Maven plug-ins from the Maven central repository into a local cache for use by the project. The Maven Central repository provides open source organizations with an easy, free and secure way to publish their components for access by millions of developers. The repository contains approximately 838,810 library versions, each having components such as jar files, source code, Javadoc and POM files. Tables 2 and 3 provide an overview of the Maven repository and its most downloaded artifacts.

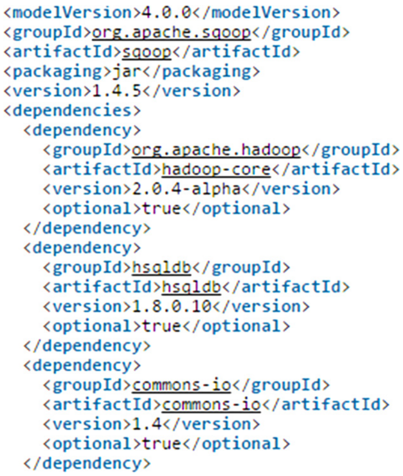
The Project Object Model (POM) is an XML file, which can be considered the basic unit of work in Maven, containing all relevant project information (see Fig. 3). This POM defines a groupId, artifactId, and version (GAV), the three required elements to describe every project. In what follows we describe briefly the main elements:

[6] http://franz.com/agraph/support/learning/Overview-of-RDFS++.lhtml.

[7] https://www.w3.org/2001/sw/wiki/Pellet.

Table 3
Top 10 downloads.

Top 10 most download artifacts in 2015-03-10	
org.codehaus.plexus	plexus-utils
junit	junit
org.apache.maven.plugins	maven-resources-plugin
org.apache.maven.plugins	maven-compiler-plugin
org.apache.maven.plugins	maven-surefire-plugin
commons-lang	commons-lang
commons-collections	common-collections
org.codehaus.plexus	plexus-interpolation
org.apache.maven.shared	maven-filtering
org.codehaus.plexus	plexus-compiler-api



```
<modelVersion>4.0.0</modelVersion>
<groupId>org.apache.sqoop</groupId>
<artifactId>sqoop</artifactId>
<packaging>jar</packaging>
<version>1.4.5</version>
<dependencies>
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-core</artifactId>
    <version>2.0.4-alpha</version>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <version>1.8.0.10</version>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>1.4</version>
    <optional>true</optional>
  </dependency>
```

Fig. 3. An excerpt of POM entry – dependencies.

< **groupId** >: A mandatory POM element which uniquely identifies the fully qualified name of the package the project belongs to. It has to be a domain name the vendor can control, and often contains the vendor name. For example, all core Maven project artifacts are under the groupId "*org.apache.maven*".

< **artifactId** >: Mandatory element which corresponds to the name of the project.

< **version** >: Mandatory element which denotes the particular version of a project.

< **dependencies** >: A key element of Maven, since almost every project depends upon others to run correctly. Maven automatically downloads the dependencies needed by the project and their transitive dependencies during the build process of the project.

3.2.1. Maven dependency management

Maven can manage both internal and external dependencies [21]. An external dependency of a Java project refers to libraries, such as Plexus,[8] Spring Framework[9] or Log4J.[10] Internal dependencies are project dependencies on service classes, model objects, or persistence logic.

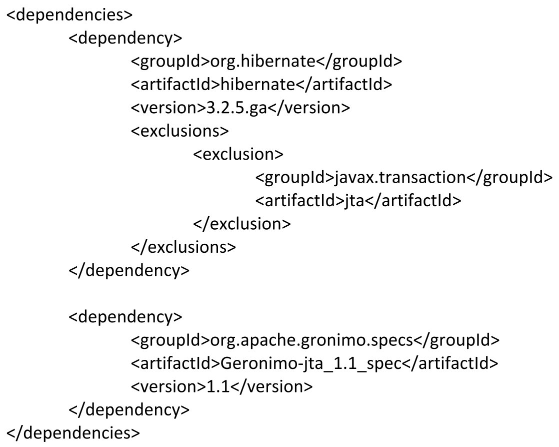
Transitive dependencies: If project-A depends on project-B, which in turn depends on project-C, then project-C is considered a transitive dependent of project-A. Part of Maven's appeal is that it can manage transitive dependencies and shield the developer from having to keep track of all of the dependencies required to compile and run an application [21]. As a result, one can just include a Java library (e.g. Spring Framework) without the need to specify this library's own dependencies.

Optional dependencies: Certain dependencies might only be required if specific features of a project are used. In order to reduce the footprint of an application, dependencies can be declared as optional (e.g., see dependencies for *commons-io* and *hsqldb* in Fig. 3).

[8] https://github.com/codehaus/plexus.

[9] https://spring.io/.

[10] http://logging.apache.org/log4j/2.x/.



```
<dependencies>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate</artifactId>
        <version>3.2.5.ga</version>
        <exclusions>
            <exclusion>
                <groupId>javax.transaction</groupId>
                <artifactId>jta</artifactId>
            </exclusion>
        </exclusions>
    </dependency>

    <dependency>
        <groupId>org.apache.gronimo.specs</groupId>
        <artifactId>Geronimo-jta_1.1_spec</artifactId>
        <version>1.1</version>
    </dependency>
</dependencies>
```

Fig. 4. Dependency exclusion and replacement.

Dependency exclusion and replacement: Maven creates internally a dependency graph to automatically deal with any conflicts and overlaps that might occur during the build process (e.g., projects on which the build process might depend on may not have declared their set of dependencies correctly). In order to address this special situation, Maven 2.x has introduced explicit dependency exclusion. Exclusions are applied for dependencies in your POM file that are specified through their *groupId* and *artifactId*. During the project build, such excluded artifact will not be added to your project's classpath. For example, Fig. 4 shows the exclusion of a Java transaction library from the transitive dependencies of Hibernate; this removed library is replaced through a new dependency on Apache's Geronimo Transaction library.

3.3. National vulnerabilities database (NVD)

In the software security domain, a software vulnerability refers to mistakes or simply facts about the security of software, networks, computers or servers that can create security risks and be used by hackers to gain access to system information or capabilities [22]. NVD [5] is a U.S government repository created to manage vulnerability data; it is a public data source. Its main function is to maintain standardized information about reported software vulnerabilities. Therefore, it creates lists of software vulnerabilities identifying vulnerabilities for different types of software systems. For the most part, software vulnerabilities in different software repositories contain custom keywords, or textual tags, which help to specify the type of problem that existed in their source codes or designs. However, these keywords are not consistent across different bug/problem reports. For that reason, the **Common Vulnerabilities and Exposures** (CVE) dictionary maintains information about publicly known vulnerabilities. It is a publically available dictionary for common vulnerabilities across different resources (development projects, open source systems, etc.). When a new vulnerability is revealed in a software product and becomes common to the security experts and interested developers, CVE will introduce an identifier number and a complete description list. The CVE details accessible through the unique identifier include the source URL, affected resources and related vulnerabilities from the same family group. Given the fast growing number of vulnerabilities, systems and developers, a classification scheme for CVEs was introduced, the **Common Weakness Enumeration** (CWE). CWE classifies the list of vulnerabilities found on CVEs in a more readable and relational style. CWE is used as a common language to describe software security weaknesses and provides a standard by which to classify and describe weaknesses.

NVD, CVE, and CWE are integrated to form a repository capturing information about software vulnerabilities and their related resources. This repository is public accessible and every two hours updated with the latest vulnerabilities information. Alternatively, this update information is also available as an XML feed.

4. Semantic global problem scanner (SE-GPS)

The research methodology used in our approach considers the fact that security concerns, vulnerabilities and project information are stored in different, heterogeneous knowledge resources. In order to make this knowledge machine processable and to allow for analysis across the individual knowledge resources, we introduce a unified ontological representation. This unified representation enables us to share knowledge and analysis results across resource boundaries, eliminating the traditional information silos these resources have remained in the past.

Fig. 5 provides an overview of our approach, by focusing on two of our ontologies: SECONT and MAVON.[11] Both ontologies are publicly available for reuse. The major steps in our research methodology include: a fact extraction process (Step 1a

[11] SECONT and MAVON are made publicly available at: https://github.com/segps/segps-ontologies.

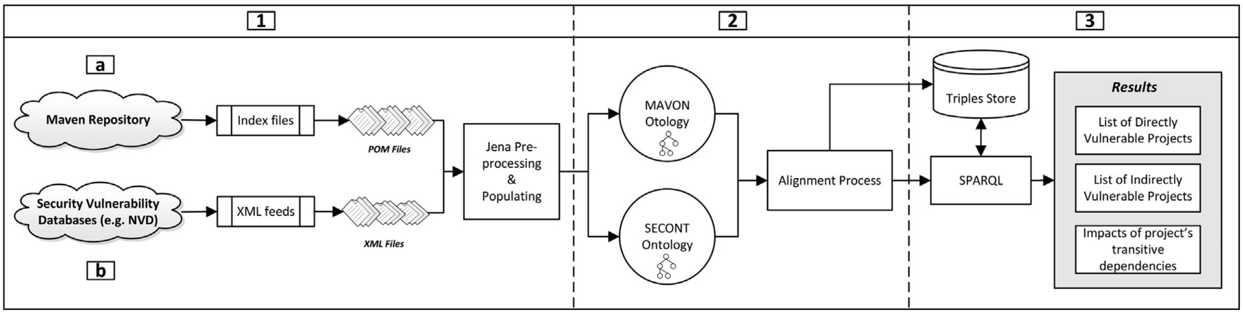

1
2
3
a
Maven Repository
Index files
POM Files
Security Vulnerability Databases (e.g. NVD)
XML feeds
XML Files
b
Jena Pre-processing & Populating
MAVON Otology
SECONT Ontology
Alignment Process
Triples Store
SPARQL
Results
List of Directly Vulnerable Projects
List of Indirectly Vulnerable Projects
Impacts of project's transitive dependencies

Fig. 5. Overall overview of our approach.

and 1b) which extracts facts from the Maven POM files and NVD XML feeds; an ontology population and matching (Step 2); and the use of SPARQL queries and RDFS + + reasoning to infer new knowledge based on security vulnerabilities and their project dependencies (Step 3).

Extracting Maven facts consists of transforming the Maven central repository index into a list of GAV (groupId, artifactId, and version) coordinates. The POM file for each GAV entry is parsed and the MAVON ontology with the extracted facts populated. For the fact extraction of the NVD repository, the NVD dataset is downloaded and updated with the latest XML feed. We then parse the extracted NVD data and populate the extracted facts in our SECONT ontology. Both ontologies are populated through the Jena[12] framework and then materialize through our triple store.

Through ontology matching (see Section 4.3) we ca an aligned and unified the ontological representation of these knowledge resources. SPARQL queries can now be used to retrieve both explicit and implicit knowledge captured in this knowledge base. We introduce several case studies (see Section 5), which take advantage of our unified representation to illustrate the applicability of our approach.

4.1. Ontology modeling

Different ontologies have been proposed in the last decade to model the software engineering and security domain. Common to the software engineering ontologies (e.g., EvoOnt [23], KOntoR [24] and LaSSIE [25]) is that they have been primarily focusing on conceptualizing a domain of discourse (e.g., issue trackers, source code, versioning systems). The modeling of workflow (e.g. [26]) and software engineering practices such as software maintenance have been addressed in [27,28] to provide developers with step-by-step guidance. However, common to all of these ontological approaches is that they lack the integration of both build systems (e.g., MAVON) and software security ontologies knowledge.

In software security, the state-of-the-art for knowledge engineering has mainly focused on the conceptualization of a domain of discourse (e.g., Intrusion Detection Systems, Information Prevention Systems), which in turn models security threats in network protocols, software viruses/worms, or host based activities (e.g., logs from top[13] or monit[14]).

Among the key decisions, we had to make during the ontology modeling was to decide whether to create a new software security ontology, or reusing exiting one. Our survey of existing knowledge modeling based approaches, showed that several software security related ontologies exist, such as: Herzog et al. [29], who presented an OWL-based information security ontology that models assets, vulnerabilities, threats, countermeasures, and their relationships. Their approach introduces in the total 88 threat classes, 79 asset classes, 133 countermeasure classes and 34 relations between them. Khadilkar et al. [30] modeled an ontology for the National Vulnerability Database (NVD) and described the concept of semantic web technology to represent the information. The ontology modeled only software products and generic security concepts of NVD data. Furthermore, portions of NVD have been mapped into RDF using a schema-based approach [31], and vulnerability descriptions remains as strings rather than RDF instances. It was the first ontology created for the NVD database to make the vulnerabilities database accessible to a larger audience. Undercoffer et al. [32] introduced an ontology taxonomy to extract the security characteristics of attacks from text found in software security bulletins. The proposed ontology consists of the three components: “Network Class”, which includes network layer information, such as TCP/IP; “System Class”, which represents operating systems; and “Attributes”, which describes the monitored process. In their research, Undercoffer et al. [32] studied specifically software security attacks relying on their observed behavior and introduced an “Attack Class” classification based on the “Means Class” (encapsulates the ways and methods used to perform an attack) and “Consequences Class” (encapsulates the outcomes of the attack). More et al. [33] used different data sources to extract security terms using OpenCalais.[15] Their methodology was based on two sections: identifying data streams from cyber-security sources (e.g., Streams from a

[12] jena.apache.org.

[13] http://linux.die.net/man/1/top.

[14] https://mmonit.com/monit/.

[15] http://new.opencalais.com/.

Table 4
MAVON ontology and its main concepts.

Concept	Semantic	Related properties
Project	Represents concept of a software project, which goes through various releases	belongsToGroup, hasVersion, hasProjectName
ProjectGroup	Each project belongs to a group. A group of projects are normally owned/developed by the same organization	containsProject, hasGroupName
ProjectVersion	Represents the concept of a version of a software library	dependsOn, hasParent, hasVersionNumber, hasGroupID, hasArtifactID

network based activity monitors like Wireshark[16]) and a modeling process (comprising of the ontology, knowledge-base, and the reasoning logic) developed using Semantic Web technology. Their methodology mainly focused on network intrusion detection systems. Their proposed ontology and knowledge-base were further extended from [32], with the authors adding rules to the reasoning logic. Iannacone et al. [34], describe an ontology developed for a cyber-security knowledge graph database. The proposed ontology is an incorporation of existing ontologies (e.g., [32,33,35]) in order to include all relevant concepts within the cyber-security domain.

Given the number existing ontologies that capture various aspects of security concepts and their definitions [31–39], an initial assumption of ours was that ontology reuse and extension should be a straight forward task, especially since both properties are considered to be a key advantage of the Semantic Web. However, a more detailed analysis of these ontologies showed that: 1) Our domain of discourse significantly differs from their application contexts. For our research, we mainly focus on high-level software security vulnerabilities and linking these high-level concepts to other software repositories. This is contrary to existing ontologies, which often conceptualize the complete security domain (hardware, network and software related concepts). As a result, we only would reuse a small subset of these ontologies for our research context. 2) Ontology reuse is further complicated by the fact that the same or similar concepts are captured and defined differently in the various ontologies, limiting the ability to reuse the concepts. 3) In our research context, the ability to infer knowledge and semantic at both the ABox and TBox level is an important aspect of our ontology design, which is in contrast to most existing ontologies in the security domain, which only focus on the conceptualization of a domain of discourse.

While we did not reuse an existing ontology in our modeling approach, we did however take advantage of the exiting security ontologies [31–39], to identify common core concepts (classes) related to software security vulnerabilities. We then modeled these core concepts in our Software Security Ontology (SECONT). In addition, SECONT also includes properties and concepts to support the creation of bi-directional links between security concerns and software artifacts in software engineering domain.

It should be noted, while our approach supports the (re-)establishing of traceability links across a variety of resources (e.g., issue trackers, Q&A repositories), we limit our discussion on creation of direct and indirect links to NVD (security domain) and the Maven (software repository domain) to ensure that our paper remains self-contained and focused. As part of our overall ontology design strategy we have followed an ontology modeling approach based on the OMG, which introduces several layers of ontology abstraction to facilitate concept reuse across abstraction levels.

4.1.1. MAven ONtology (MAVON)

As previously discussed, project POM files in Maven define unidirectional dependencies – to include only these libraries in the build process that a project depends on. However, while such a unidirectional dependency model works well for managing build dependencies, it restricts a user's ability to mine knowledge stored in the repository. For example, using Maven it is currently not possible for a user to identify all published library releases for a specific project. To overcome this challenge, we take advantage of the Semantic Web and its standardized knowledge modeling approach. More specifically, we introduce an ontology (MAVON) that allows us to provide analysis services, which no longer rely on Maven's proprietary analysis and knowledge modeling approach. Our MAVON ontology is based upon three core concepts: **Project**, a description of a software project; **ProjectVersion,** which models release, and version information and **ProjectGroup**, capturing related software products developed by the same organization. Fig. 6 provides an overview of the MAVON ontology, with Tables 4 and 5 describing in more detail concepts and associations introduced by our ontology.

4.1.2. Software sECurity ONTology (SECONT)

The SECONT ontology captures concepts and their relations found in the software security domain. SECONT's core security vulnerability-related classes (e.g., Vulnerability, Weakness, and Product) were derived from the NVD database schema. Furthermore, since the purpose of SECONT was not only to conceptualize the software security domain, but also to support bi-directional links between security concerns and software artifacts in software engineering domain, we enriched our core design with properties and concepts to support our ontology mapping process.

In what follows, we describe SECONT's core concepts in more detail.

[16] http://www.wireshark.org/.

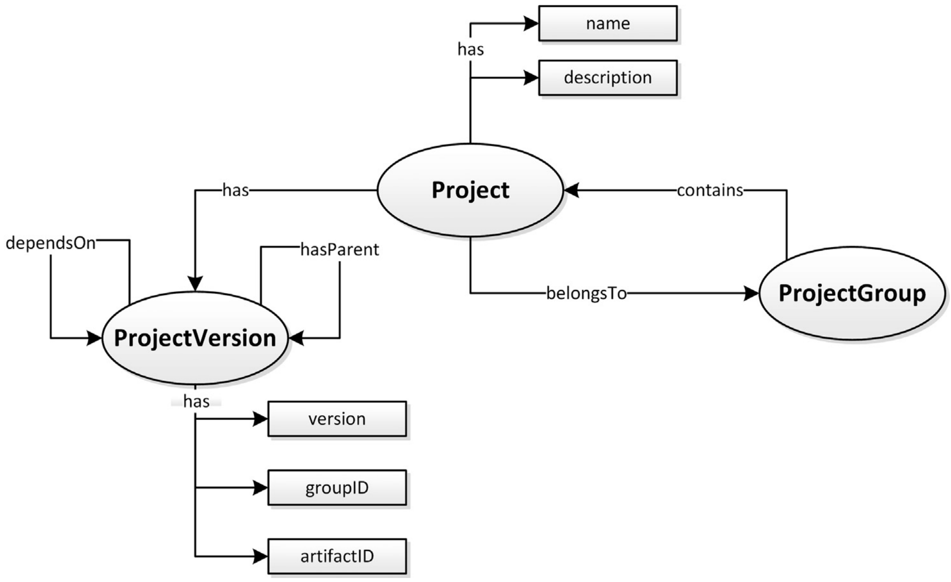


Fig. 6. Main concepts of the MAVON ontology.

Table 5
MAVON ontology and its properties.

Property	Semantic	Domain	Range
belongsToGroup	Each Project belongs to a ProjectGroup	Project	ProjectGroup
hasVersion	Each Project has a set of ProjectVersions	Project	ProjectVersion
hasProjectName	Represents the name of the Project	Project	Literal
containsProject	Each ProjectGroup contains a set of Projects	ProjectGroup	Project
hasGroupName	Represents the name of the ProjectGroup	ProjectGroup	Literal
dependsOn	Each version of a project may depend on other Project versions	ProjectVersion	ProjectVersion
hasParent	Each version of a project may inherit attributes of another Project version	ProjectVersion	ProjectVersion
hasVersionNumber	Represents the release number of the project version	ProjectVersion	Literal
hasGroupID	Represents the name of the group the version belongs to	ProjectVersion	Literal
hasArtifactID	Represents the name of the project the version belongs to	ProjectVersion	Literal

Vulnerability: In NVD, vulnerability is a core class that can be identified by its unique CVE-ID. The class also captures various vulnerability details, such as: date, severity score, vulnerability summary, sources (related to the same vulnerability). Each vulnerability has a list of affected products associated, described by its *Common Platform Enumeration* (CPE),[17] a standard machine-readable format for encoding names of IT products and platforms.

Weakness: The Weakness class in SECONT captures the vulnerability type identified through the CWE-ID. CWE [39] provides two different hierarchical perspectives on vulnerabilities: research, and development hierarchy. For our paper, we adopted the research view, which is a scientific classification based on types of weaknesses and their associated subclasses resulting in a hierarchical representation of popular software security weaknesses. In SECONT the Weakness captures vulnerability based on their weakness type and can be used to compute a vulnerability's severity score based on the CWE-ID.

Product: As part of our knowledge modeling approach, we have the original NVD entry further sub-classed by its *application* (e.g., APIs) and *operating system* (e.g. Android 2.3.2).

Fig. 7 shows a partial view of SECONT with its core concepts and entities relevant to the software security domain (additional details are shown in Tables 6 and 7).

4.2. SE-GPS knowledge engineering and evolution

The problem of ontology evolution is far from trivial. The Semantic Web is characterized by decentralization, heterogeneity, and lack of central control or authority. These new features have greatly contributed to the success of the Semantic Web but at the same time, also introduced several new challenges. In the following, we briefly describe our ontology design process for the SECONT and MAVON ontologies (Fig. 8) and how we address some of these knowledge evolution challenges.

[17] Common Platform Enumeration – http://cpe.mitre.org.

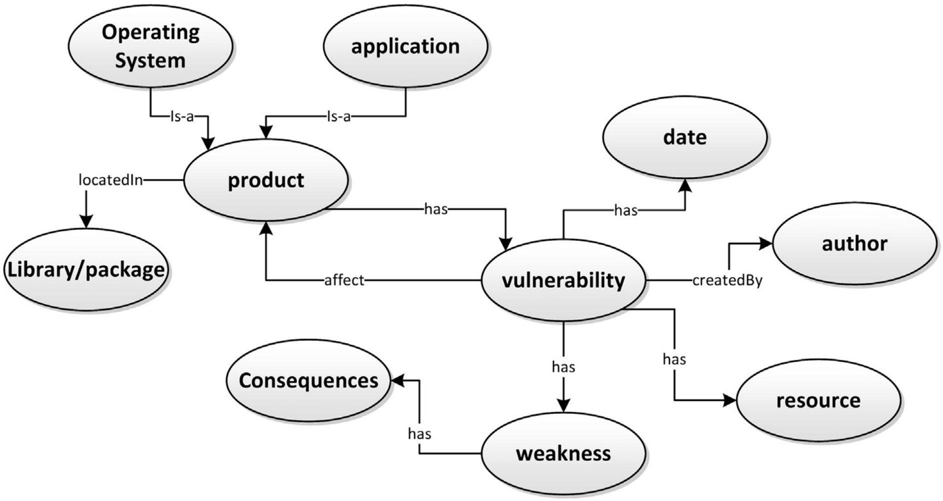


Fig. 7. Overview of security concepts in SECONT.

Table 6
SECONT ontology and its main concepts.

Concept	Semantic	Related properties
Vulnerability	Represents a flaw in a product that could allow an attacker to compromise the integrity, availability, or confidentiality of that product.	affectsProduct, hasWeakness, createdBy, hasResource
Product	Each vulnerability affects products either as software or as operating system.	hasVulnerability, locatedIn, hasProductName, hasVendorName, hasVersionNumber
Weakness	Represents the classification of each vulnerability type.	hasConsequences
Author	Represents a person who reports the security vulnerability. He/she might be the attacker, a software developer who discovers the vulnerability after a maintenance task, or a security expert identifying and reporting the vulnerability after analyzing the software project using auditing tools for security analysis.[18]	hasName, isPerson, hasURL

Table 7
SECONT ontology and its properties.

Property	Semantic	Suggested domain	Suggested range
affectsProduct	Each vulnerability and its set of products	Vulnerability	Product
hasWeakness	Each vulnerability has a type of attack (e.g. Denial of Service, Identity theft, etc.)	Vulnerability	Weakness
hasProductName	Represents the name of the affected product	Product	Literal
hasVendorName	Represents the company own that product	Product	Literal
hasVersionNumber	Represents the version number for the affected product	Product	Literal
createdBy	Represents each vulnerability has been created either by person (e.g. hackers, security expert working with security advisory companies)	Vulnerability	Author
hasVulnerability	It is an inverse-of affectProduct property. It represents that each vulnerable software system has a vulnerability id	Product	Vulnerability
locatedIn	Each identified vulnerability has a location in the software system; either that location is file, or package	Product	Location
hasConsequences	Represent the type of the action that is caused by classified vulnerability	Weakness	Consequences

As shown in Fig. 8 concepts (e.g., vulnerabilities, weaknesses, consequences) and build concepts (e.g., project). These concepts are then further validated by manual inspecting them against their instances, to ensure that they capture meaningful concepts. In order to avoid ambiguity in our design, we manually verified before updating our design that none of these extracted concepts or relations already exist within our ontologies. During the last modeling step, we perform an ontology matching (Section 4.3) to establish explicit or implicit links among our ontologies, allowing for a seamless integration.

Given that our ontologies are quite small in size (MAVON 34 Million triples and SECONT 32 Million triples), knowledge evolution in terms of adding new facts to the knowledge base works reasonable well. Updated vulnerability information is

[18] http://h3xstream.github.io/find-sec-bugs/.

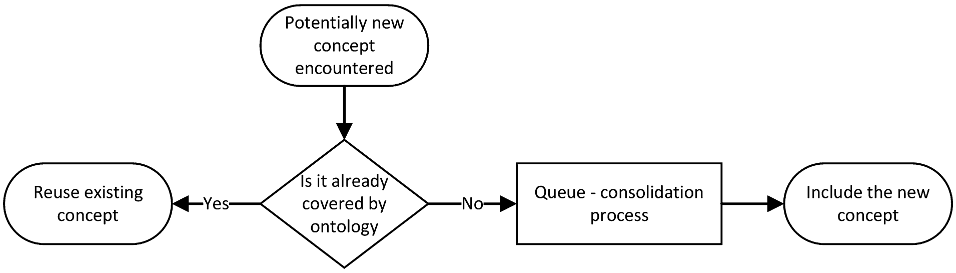

Potentially new concept encountered
Is it already covered by ontology
Yes
No
Reuse existing concept
Queue - consolidation process
Include the new concept

Fig. 8. SE-GPS informal design process when new concept encountered.

extracted from the regular XML feeds (every two hours) provided by NVD; the collected daily updates are then populated to the SECONT ontology at the end of the day. For the ontology population, we take advantage of the incremental update features supported by the triple store. We also perform periodical updates of the MAVON ontology, by downloading and extracting updated facts that have been published in the Maven repository. Furthermore, since our TBox design will not be affected by these periodical updates – no new concepts will be introduced, only changes to ABox (instance level) occur, our established traceability links at the TBox level remain stable and consistent throughout these periodical updates.

Result integration. It is not realistic to expect all the sources to share a single, consistent view at all times, in particular if one considers result integration, where results might be generated by different analysis approaches. Rather, we expect disagreements between individual users and tools during an analysis. An elegant model for managing (possibly conflicting) information from different sources has been proposed by [40]: Knowledge is structured into *viewpoints* and *topics*. Viewpoints are environments that represent a particular point of view (e.g., information stemming from a particular *tool* or entered by a *user*). Topics are environments that contain knowledge that is relevant to a given subject (e.g., design patterns, architectural recovery). These environments are *nested* within each other: viewpoints can contain either other viewpoints or topics. A topic can contain knowledge pertaining to its subject, but also other viewpoints, e.g., when the subject is another user.

Through this approach, we can explicitly distinguish between the knowledge a particular resource has about a certain topic and at the same time, manage possible conflicting knowledge about what a resource *believes other resources may believe* about the same topic, since this information is contained within different, nested viewpoints. These viewpoints create *spaces* within which to do reasoning: consistency can be maintained within a topic or a viewpoint, but at the same time, conflicting information about the same topic can be stored in another viewpoint. This allows us to collect and maintain as much knowledge as possible, attributing it to its sources, without having to decide on a “correct” set of information, thereby losing information prematurely. Viewpoints can be constructed as well as destructed through the processes of ascription and percolation. Stated briefly, the process of ascription allows incorporating knowledge from other viewpoints (users, tools) unless there is already conflicting information on the same topic. The mechanism of percolation is introduced for the deconstruction of nested knowledge. Here, some (assumed) held knowledge on a topic contained in a nested viewpoint may be percolated into its outer environments, up to the top-level viewpoint of a user (or the main environment) and be thus acquired as knowledge.

4.3. Ontology matching

In order for us to take full advantage of the knowledge captured in the SECONT and MAVON ontologies, we apply ontology matching techniques to establish traceability links among these ontologies. These links reduce the semantic gap between these ontologies and are essential pre-requisites for supporting seamless knowledge integration. As discussed in [41] matching techniques can be divided in two categories: (1) content-based matching, which focuses on internal information originating from the ontologies to be matched and (2) context-based, which matches external information originating from relations between ontologies or other external resources. In our case, we take advantage of context-based matching [41], which can be further divided into four major categories of ontology matching methods. These methods can be further distinguished depending on the type of data the algorithms work on: terminological (lexical mapping), structural, extensional and semantic (semantic mappings) methods. Terminological and structural methods make use of data captured as part of ontology descriptions (e.g., labels, comments, attributes and types, relations. with other entities); extensional methods work on instances – matching facts across ontologies; and semantic methods match ontologies based on the semantic interpretation of the models, using logic reasoning to infer correspondences. In our approach, we use a combination of terminological and semantic matching methods (shown in Fig. 9).

For the matching, we take advantage of the semantic equivalence which exists between *SECONT:Product* and *MAVON:ProjectVersion*, since both concepts refer in our modeling approach to the same entity.

In OWL, this equivalence relationship between classes and properties can be explicitly expressed through the use of *owl:equivalentClass* and *owl:equivalentProperty* constructs. These equivalence relationships provide an intuitive way to express relationships between ontology classes and properties.

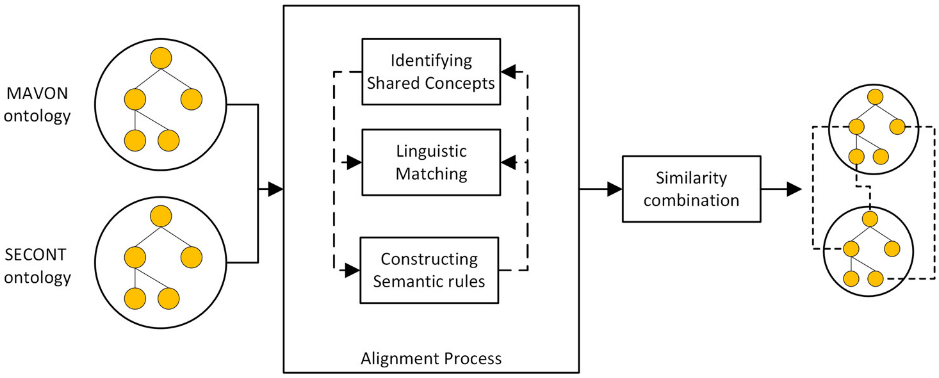
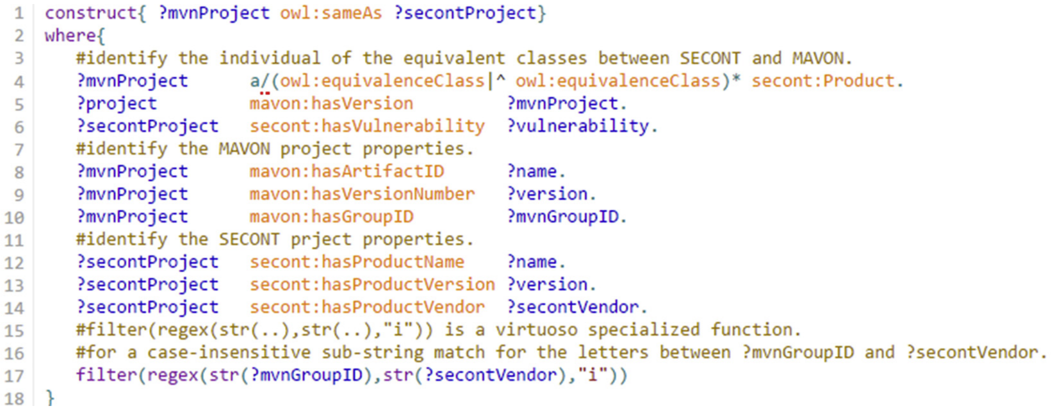


Fig. 9. Ontologies matching process.

```
secont:hasProductName     ≡ mavon:hasArtifactID
secont:hasVersionNumber ≡ mavon:hasVersionNumber
```

Listing 1. Equivalent relations with data properties.



```
construct{ ?mvnProject owl:sameAs ?secontProject}
where{
   #identify the individual of the equivalent classes between SECONT and MAVON.
   ?mvnProject      a/(owl:equivalenceClass|^ owl:equivalenceClass)* secont:Product.
   ?project         mavon:hasVersion           ?mvnProject.
   ?secontProject   secont:hasVulnerability    ?vulnerability.
   #identify the MAVON project properties.
   ?mvnProject      mavon:hasArtifactID        ?name.
   ?mvnProject      mavon:hasVersionNumber     ?version.
   ?mvnProject      mavon:hasGroupID           ?mvnGroupID.
   #identify the SECONT prject properties.
   ?secontProject   secont:hasProductName      ?name.
   ?secontProject   secont:hasProductVersion ?version.
   ?secontProject   secont:hasProductVendor    ?secontVendor.
   #filter(regex(str(..),str(..),"i")) is a virtuoso specialized function.
   #for a case-insensitive sub-string match for the letters between ?mvnGroupID and ?secontVendor.
   filter(regex(str(?mvnGroupID),str(?secontVendor),"i"))
}
```

Query 1. owl:sameAs constructing query.

In order, to capture the semantic relationships between individuals in MAVON and SECONT we use the *owl:sameAs* construct. For example, "*Secont#**mortbay:jetty:6.1.1*** *owl:sameAs Mavon#**org.mortbay.jetty:jetty:6.1.1***" since both instances correspond to the same vulnerable product "***jetty 6.1.1***". Using the *owl:sameAs* construct, we can now establish links between MAVON and SECONT individuals based on the following rules:

- Both *owl:equivalentProperty* pairs (*secont:hasProductName*, *mavon:hasArtifactID*) and (*secont:hasVersionNumber*, *mavon:hasVersionNumber*) must have the same range values (Listing 1) – individuals must therefore have identical product names and version numbers.
- The value for *secont:hasProductVendor* must be contained in *mavon:hasGroupID*'s value (see Query 1). This ensures that products with the same name and version are matched if and only if they are produced by the "same" company.

However, it should be noted that there is no guarantee that two ontologies in the same domain will align through shared concepts, due to ambiguity or lack of such shared concepts. In Section 6, we evaluate in detail the precision and recall for our approach by comparing it against the OWASP Dependency-Check tool [42].

5. Case studies

In what follows, we present results from three case studies, which we conducted to evaluate the applicability of our knowledge modeling approach. The three case studies are:

Identifying direct vulnerable projects – Identifies projects within the Maven repository containing vulnerabilities, which are already known and reported in the NVD repository (Fig. 10.1).

Identifying indirect vulnerable projects – Identifies vulnerabilities already reported in the NVD repository that have transitive dependencies on projects within the Maven repository (Fig. 10.2).

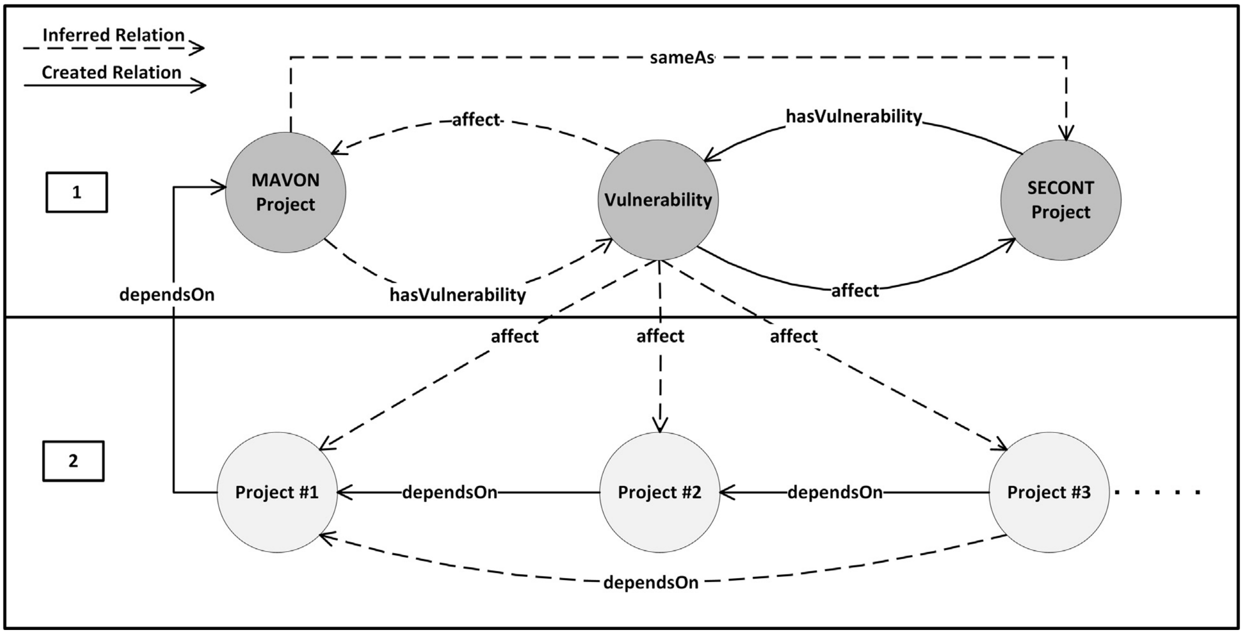


Fig. 10. Direct and indirect security vulnerabilities.

Table 8
Statistics of case study data.

Ontology	Projects	Versions	Vendors	Total triples
SECONT	24,807	145,101	13,990	3,2015,659
MAVON	92,989	795,311	15,883	3,4089,236

Usage-based transitive dependency impact – Studies the direct/indirect impact of vulnerable components on projects in Maven, based on the actual usage of these vulnerable components.

The objective of the case studies is to illustrate the applicability of our approach by answering the research questions introduced in Section 2.2. For each research question, we present its motivation, the analysis approach and a discussion of our findings. Table 8 shows some statistics for the dataset we used in our case studies.

5.1. Direct security vulnerabilities in MAVON

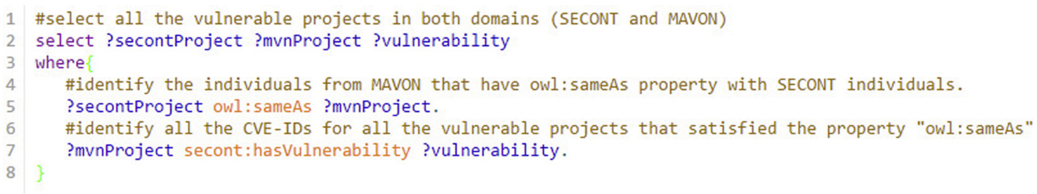
RQ1: *To what extent are users/developers of open source components directly susceptible to security vulnerabilities?*

Motivation: Software projects such as libraries, programs or utilities are components designed to perform some required functions [22], which can be reused by other programs. The Maven repository contains over 100,000 of such components with on average 9 different versions for each component. Open and closed source software projects leverage these components by reusing and integrating their available API functionalities (e.g.; data access, resource management, user interface interaction, and business functions).

Given such global components reuse, vulnerabilities identified in shared components will no longer be restricted to a single file or project, but instead can impact a large number of dependent components or even global software ecosystems [22]. This potential risk is further increased by the already reported large number of known vulnerabilities and coding mistakes. For example, MITRE[19] has classified in their CWE database [39] almost 1000 different classes of inadvertent coding mistakes or over 70,000 known vulnerabilities have so far been reported in NVD.[20]

Approach: In order to determine the potential impact of a vulnerability on open source projects, we first identify projects in MAVON, which have reported NVD vulnerabilities (captured in our SECONT ontology). We then establish a semantic link between the MAVON and SECONT ontologies using the following SPARQL query (Query 2).

[19] http://www.mitre.org/.
[20] As of July 1st, 2015.

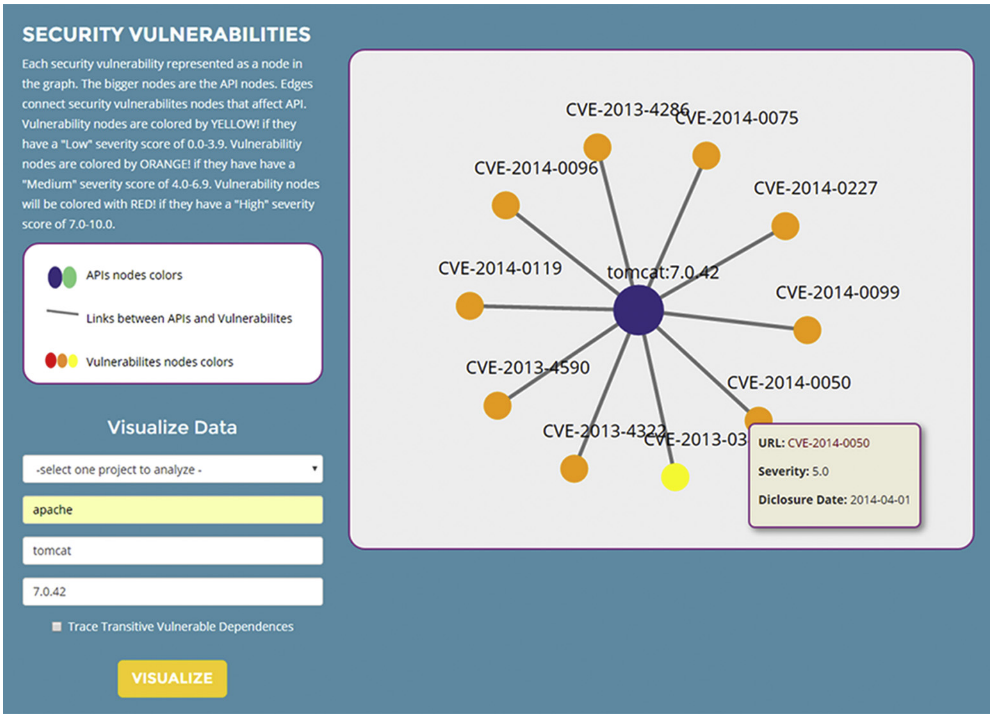


```
#select all the vulnerable projects in both domains (SECONT and MAVON)
select ?secontProject ?mvnProject ?vulnerability
where{
   #identify the individuals from MAVON that have owl:sameAs property with SECONT individuals.
   ?secontProject owl:sameAs ?mvnProject.
   #identify all the CVE-IDs for all the vulnerable projects that satisfied the property "owl:sameAs"
   ?mvnProject secont:hasVulnerability ?vulnerability.
}
```

Query 2. SPARQL query for listing equivalent vulnerable projects in MAVON and SECONT.

Table 9
Examples of linked vulnerabilities.

SECONT data	MAVON data	CVEs
Secont#sonatype:nexus:2.3.1	Mavon#org.sonatype.nexus:nexus:2.3.1	Secont#CVE-2014-0792
Secont#apache:poi:3.7	Mavon#org.apache.poi:poi:3.7	Secont#CVE-2014-3529
Secont#apache:axis:1.4	Mavon#org.apache.axis:axis:1.4	Secont#CVE-2014-3596
Secont#apache:wss4j:2.0.0	Mavon#org.apache.wss4j:wss4j:2.0.0	Secont#CVE-2015-0227
Secont#apache:wss4j:2.0.1	Mavon#org.apache.wss4j:wss4j:2.0.1	Secont#CVE-2015-0227
Secont#apache:wss4j:2.0.0-rc1	Mavon#org.apache.wss4j:wss4j:2.0.0-rc1	Secont#CVE-2014-3623



SECURITY VULNERABILITIES

Each security vulnerability represented as a node in the graph. The bigger nodes are the API nodes. Edges connect security vulnerabilites nodes that affect API. Vulnerability nodes are colored by YELLOW! if they have a "Low" severity score of 0.0-3.9. Vulnerabilitiy nodes are colored by ORANGE! if they have have a "Medium" severity score of 4.0-6.9. Vulnerability nodes will be colored with RED! if they have a "High" severity score of 7.0-10.0.

Visualize Data

-select one project to analyze -

apache

tomcat

7.0.42

URL: CVE-2014-0050
Severity: 5.0
Diclosure Date: 2014-04-01

Fig. 11. Security vulnerabilities affecting Tomcat 7.0.42.

Findings: Our analysis of Maven showed that 0.62% of all Maven projects include known vulnerabilities, which have been already disclosed in the NVD database. Table 9 shows some of these known vulnerabilities and their occurrences in the Maven build repository (e.g., Sonatype Nexus Version 2.3.1 contains the NVD vulnerability *CVE-2014-0792*).

Vulnerable projects identified in Maven may suffer from multiple vulnerabilities. Further analysis of our results showed that projects might often suffer from multiple vulnerabilities. Our analysis of the Maven repository showed that 43.92% of the 576 identified vulnerable projects suffer from multiple vulnerabilities, with Oracle's JavaFX 2.2 being the most vulnerable project in the repository, containing 40 known vulnerabilities. Providing developers with such insights can help avoid the reuse of APIs/components prone to vulnerabilities and therefore improve trust in their components/products.

We implemented a visualization tool – SE-GPS [43], which allows us to analysis and visualize individual Maven projects and all their reported vulnerabilities. For example, the screen capture in Fig. 11 shows the NVD vulnerabilities for *Tomcat Version 7.0.42*, which has 10 known security vulnerabilities.

Security vulnerabilities can affect several product releases. This type of analysis can guide software and security engineers to ensure consistent patching and regression testing of vulnerabilities across product lines or versions of a product,

Table 10
Critical vulnerabilities for android project.

Android version	CVE-IDs	# of direct dependencies
Mavon#com.google.android:android:2.2.1	CVE-2013-4787	360
Mavon#com.google.android:android:2.3.1	CVE-2013-4787	176
Mavon#com.google.android:android:2.3.3	CVE-2013-4787	351
Mavon#com.google.android:android:3.0	CVE-2013-4787	34
Mavon#com.google.android:android:4.2	CVE-2013-4787	1

Table 11
Evaluation projects.

Name	# of versions	Versions
Apache Axis	**1**	**1.4**
Apache CXF	**3**	**2.74, 3.0.0, 3.0.1**
Apache Hbase	**12**	**0.92.0–0.92.2, 0.94.0–0.94.7, 0.94.6.1**
Apache Commons-Compress	**5**	**1.0 to 1.4**
Apache Httpclient	**14**	**4.0, 4.0.1, 4.1, 4.1.1, 4.1.2, 4.2, 4.2.1–4.2.3, 4.3, 4.3.1–4.3.4**
Apache POI	**2**	**3.6, 3.7**
Apache Wicket	**22**	**1.4.0–1.4.8, 1.4.10–1.4.22**
Apache Wss4j	**3**	**2.0.0, 2.0.0-rc1, 2.0.1**
Mortbay Jetty	**12**	**6.0.1, 6.1.1, 6.1.4–6.1.7, 6.1.9, 6.1.12, 6.1.14–6.1.16, 6.1.19**
Neo4j	**1**	**1.9.2**
Sonatype Nexus	**11**	**2.0, 2.0.1–2.0.6, 2.1, 2.1.1, 2.2, 2.3.1**
Apache Syncope	**1**	**1.0.6**

$$mavon\colon dependsOn\ rdf\colon type\ owl\colon TransitiveProperty$$

Listing 2. Define transitive constraints.

since the same vulnerability might affect different product releases. For example, Table 10 shows security vulnerability *CVE-2013-4787*, which has been reported for five different Android versions.

Are vulnerabilities components still used even after a new patched version had been released? For example, in December 2010, Google released its Nexus S smartphone.[21] The phone was originally running on Android 2.3.3 – an Android version that already contained the security vulnerability discussed in Table 11. While the Nexus S received regular Android OS updates up to Android Version 4.2, an actual fix of the reported vulnerability (*CVE-2013-4787*) was only introduced with Android 4.2.2. However, this new Android version is no longer supported and distributed for the Nexus S, leaving existing users of the phone susceptible to attacks.

5.2. Indirect security vulnerabilities in MAVON

RQ2: *Can projects which are indirectly dependent on vulnerable components be identified?*

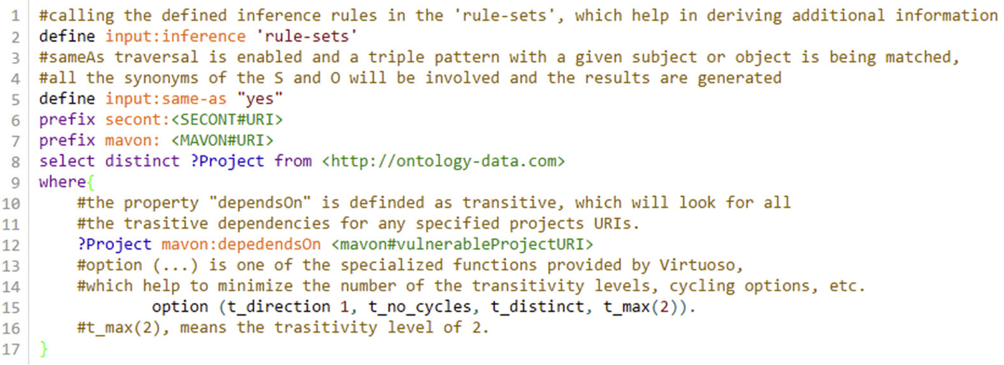
Motivation: Most software projects depend in their implementation on external components [22]. As our initial analysis of the NVD and Maven repository showed, the Maven repository includes 576 of such vulnerable projects/components. In what follows, we extend our analysis to include projects that potentially are indirectly affected by these vulnerable components. In order to compute this impact set, we take advantage of our Semantic Web based modeling approach and its support for inference services.

Approach: A relation P is said to be transitive if $P(a,b)$ and $P(b,c)$ implies $P(a,c)$; this can be expressed in OWL through the *owl:TransitiveProperty construct*. We define therefore *mavon:dependsOn* to be a bi-directional transitive property of type *owl:TransitiveProperty* (Listing 2). It should be noted that for the transitive dependencies analysis in this case study we do not distinguish if a project calls or makes use of the actual vulnerable part of the source code in that component.

With this transitive construct, we are now able to retrieve a list of all projects that have a direct and transitive dependency on the vulnerable library, and vice versa (see Query 3).

Findings: Our transitive vulnerable dependency analysis shows that while a project might not have any direct vulnerability reported in the NVD database, it can still depend indirectly on other vulnerable libraries and components. For our case study, we selected six open source libraries with reported vulnerabilities. The results of our study (Fig. 12) shows that similar to traditional impact analysis, the inclusion of transitive dependency levels will significantly increase the potential impact scope and therefore the set of potential affected projects. For example, Apache Derby 10.1.1.0 has 382 direct project

[21] https://en.wikipedia.org/wiki/Nexus_S.



```
#calling the defined inference rules in the 'rule-sets', which help in deriving additional information
define input:inference 'rule-sets'
#sameAs traversal is enabled and a triple pattern with a given subject or object is being matched,
#all the synonyms of the S and O will be involved and the results are generated
define input:same-as "yes"
prefix secont:<SECONT#URI>
prefix mavon: <MAVON#URI>
select distinct ?Project from <http://ontology-data.com>
where{
    #the property "dependsOn" is definded as transitive, which will look for all
    #the trasitive dependencies for any specified projects URIs.
    ?Project mavon:depedendsOn <mavon#vulnerableProjectURI>
    #option (...) is one of the specialized functions provided by Virtuoso,
    #which help to minimize the number of the transitivity levels, cycling options, etc.
            option (t_direction 1, t_no_cycles, t_distinct, t_max(2)).
    #t_max(2), means the trasitivity level of 2.
}
```

Query 3. Query for locating indirect vulnerabilities.

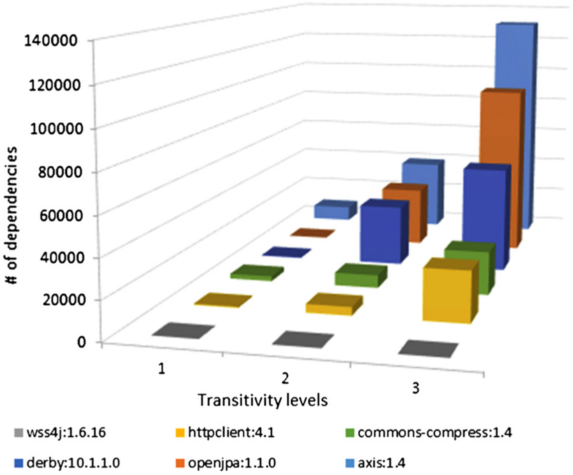


Fig. 12. Transitive dependencies for 6 selected projects which contain known vulnerabilities.

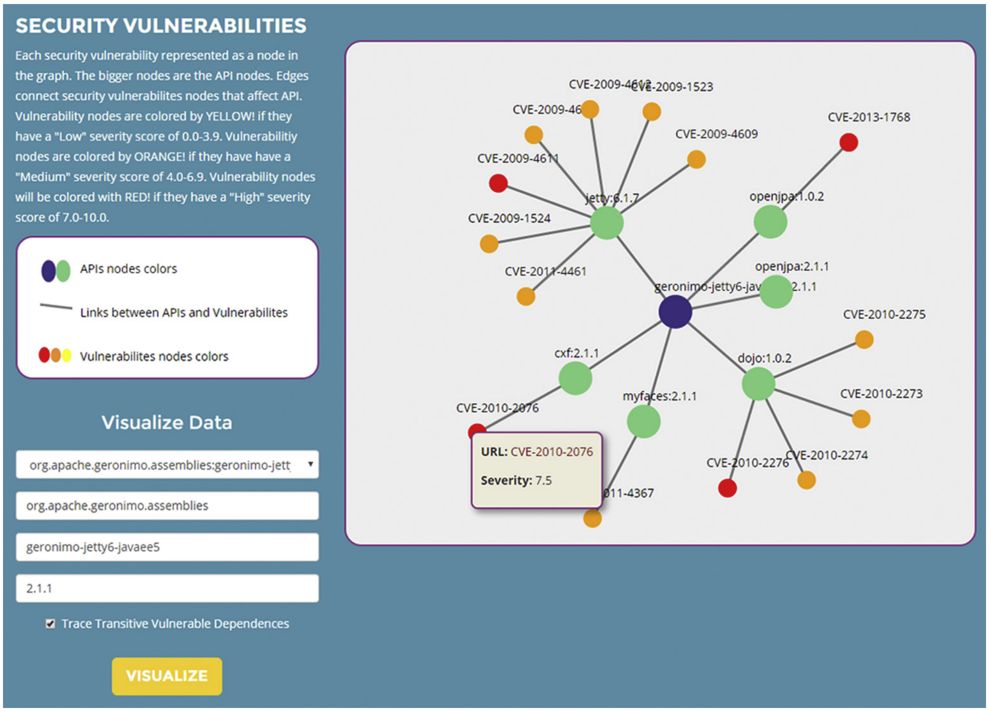
dependencies, the potential dependencies increase exponentially with the inclusion of additional transitivity level in the analysis (e.g., 130,000 projects, by including 3 levels of transitivity). This exponential growth is caused by our analysis used in this case study, which does not distinguish whether a dependent component/project actually makes use of the vulnerable code or not.

End-user support during the analysis once again provided by our SE-GPS visualization tool, which includes the option to display both direct and transitive dependencies, as well as the severity level of vulnerabilities (color-coded based on severity level reported by the NVD database). Fig. 13 shows an example, where the direct and indirect dependencies for the Geronimo-jetty6-javaee5 (version 2.1.1) project are visualized. While the project has no direct vulnerability reported in the NVD database, it indirectly depends on several components with known security issues.

RQ3: *Can a more formal knowledge representation be beneficial in determining transitive dependencies of vulnerable components in software ecosystems?*

Motivation: In RQ2, we introduce a transitive dependency analysis that identifies all potential transitive dependencies between a vulnerable component and other components. A limitation of this dependency analysis is however that it does not distinguish if a vulnerable part of a component was actually used by the dependent components. While this conservative analysis will result in a high recall, it will also generate many false positives. For example, a component might be dependent on a component that contains a vulnerability; however, the actual vulnerable part of that component might never be used. In what follows, we refine our original dependency analysis approach to include a static call-dependency graph that considers also the actual usage of vulnerable part of a component in its analysis.

Approach: Whenever a vendor confirms the existence of a vulnerability in a product, a patch is released to fix the security flaw. The NVD database sometimes contains links to such patches stored in a corresponding versioning repository. Through these links, it is now possible to locate the vulnerability in earlier version of that component. Having identified the location of vulnerability component part, we can now restrict our transitive dependency result set to include only components that actually depend on the vulnerable part of the source code.



SECURITY VULNERABILITIES

Each security vulnerability represented as a node in the graph. The bigger nodes are the API nodes. Edges connect security vulnerabilites nodes that affect API. Vulnerability nodes are colored by YELLOW! if they have a "Low" severity score of 0.0-3.9. Vulnerabilitiy nodes are colored by ORANGE! if they have have a "Medium" severity score of 4.0-6.9. Vulnerability nodes will be colored with RED! if they have a "High" severity score of 7.0-10.0.

●● APIs nodes colors

Links between APIs and Vulnerabilites

●●● Vulnerabilites nodes colors

Visualize Data

org.apache.geronimo.assemblies:geronimo-jett ▾

org.apache.geronimo.assemblies

geronimo-jetty6-javaee5

2.1.1

☑ Trace Transitive Vulnerable Dependences

VISUALIZE

URL: CVE-2010-2076

Severity: 7.5

Fig. 13. Transitive vulnerable dependencies for Geronimo-jetty6-javee5 version 2.1.1.

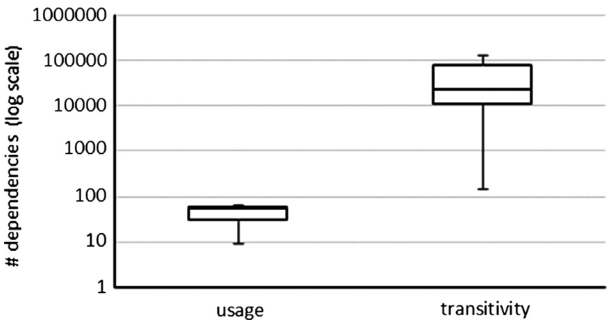


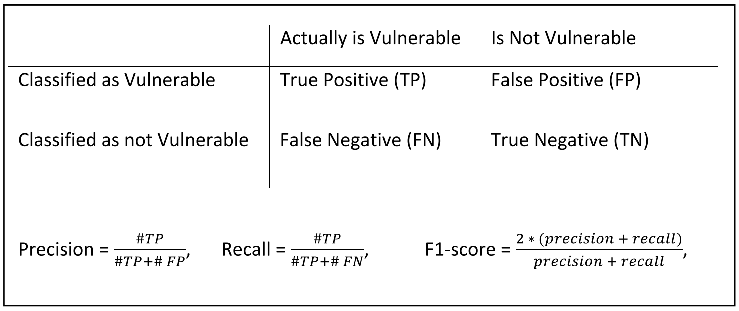
Fig. 14. Transitive dependency vs. actual usage of vulnerable APIs.

Findings: In this case study we compare the results from our initial transitive dependency study, with our refined analysis, that considers only the directly or indirectly call dependencies of components that use the vulnerable part(s). As our analysis shows (Fig. 14), without considering the actual usage of the vulnerable part of the component, the potential impact set for the six selected projects ranged from 151 to 130,612 (median of 22,859) projects. Including the usage of the vulnerable parts in the analysis, reduced the number of potentially impacted projects to 9–67 (median of 54), depending on the project analyzed.

6. Discussion and threats to validity

6.1. Discussion

As our case studies show, providing a unified and formal representation of original heterogeneous knowledge resources can indeed help eliminating existing information silos, by creating information hubs, which can share and integrate knowledge across knowledge boarders. For these information hubs, we take advantage of Semantic Web technologies and use ontology-matching methods (terminological and semantic matching methods) to link our ontologies (e.g., SECONT and MAVON). This unified schema is now machine-human accessible and can implicitly and explicitly linked through shared concepts. Moreover, unlike traditional mining software repositories techniques, our approach allows for analysis results and inferred knowledge to become part of the knowledge base and allow for their later consumption (processing) by either human or machines.



	Actually is Vulnerable	Is Not Vulnerable
Classified as Vulnerable	True Positive (TP)	False Positive (FP)
Classified as not Vulnerable	False Negative (FN)	True Negative (TN)

Precision = $\frac{\#TP}{\#TP+\#\,FP}$, Recall = $\frac{\#TP}{\#TP+\#\,FN}$, F1-score = $\frac{2 * (precision + recall)}{precision + recall}$,

Fig. 15. Link quality assessment.

Identifying known security vulnerabilities in software projects has been widely discussed in the literature. However, our approach differs from this existing work that it unifies two heterogeneous sources of information (software security repositories and build system repositories). Using OWL semantic relationships (e.g., *owl:sameAs*, *owl:equivalentClass* and *owl:equivalentProperty*) and RDFS++ reasoning, we can now infer new knowledge about vulnerable transitive dependencies to provide project stakeholders with insights on threats that may harm their projects. In addition, our analysis includes inter project and component dependencies, extending traditional vulnerability analysis beyond the individual project scope.

The results from our case studies show that the problem of depending on third party components with known security vulnerabilities is widespread in the software development community. These dependencies come with security implications, which are often unknown or ignored by developers, due to the lack of traceability links and tool support to identify these direct and indirect dependencies caused by the reuse of vulnerable components.

6.2. Threats to validity

6.2.1. Internal validity

Mining Maven and NVD repositories. Our work relies on the ability to mine facts from the Maven and NVD repositories to populate our ontologies. A common problem with mining software repository is that repositories often contain noise in their data due to ambiguity, inconsistency or incompleteness. This threat can be mitigated in our research context, since vulnerabilities published in NVD are manually validated and managed by security experts and therefore making this data less prone to noise. Similarly, the Maven repository captures dependencies related to a particular build file, while ensuring that the dependencies are fully specified and available, eliminating not only ambiguities and inconsistency at the project build but also for the complete dataset.

Other threats to the mining of these repositories are related to the fact that we only extracted vulnerabilities reported from 2002 to 2015 from the NVD database. Given the number of vulnerabilities, the broad range of affected projects and types of vulnerabilities reported, we consider the dataset as large enough to avoid any bias towards certain vulnerabilities or affected libraries.

Mapping accuracy. The ontology matching process used in our approach is based on terminological and semantic similarities. The two repositories, NVD and Maven, use different nomenclatures for identifying projects and vendors. While a human can easily recognize the mapping between two different but semantically equal concepts, this is not the case for an automated solution. Since our results rely on the automatic mapping between the MAVON and SECONT ontology, the accuracy of this mapping might not be considered adequate. In order to mitigate this threat, we conducted an evaluation, comparing the results obtain from our approach against the OWASP Dependency-Check tool. The OWASP tool, checks for any known, publicly disclosed, vulnerabilities on which a project is directly dependent. We randomly selected 87 vulnerable open-source Java projects (Table 11) and compared our results with the results obtained from OWASP Dependency-Check tool. For the evaluation of the mapping accuracy, we only considered direct dependencies between vulnerabilities and projects, since the OWSAP only supports this type of dependency mapping.

For the manual analysis of the mapping results, we used precision, recall and the F1 score (harmonic mean) [44] to measure the accuracy for both approaches. Precision is the fraction of projects classified as vulnerable that are actually vulnerable, while recall is the fraction of relevant (true positive) vulnerabilities that are retrieved. The F1 score, also called the harmonic mean, is a weighted average of the precision and recall to measure the accuracy of the two approaches (see Fig. 15).

For our comparison, we calculated precision, recall and F1-score for ten iterations, by incrementally increasing the number of projects being evaluated in each iteration. This incremental approach allowed us to assess the potential impact of the dataset size on the accuracy of our analysis. Table 12 and Fig. 16 show the precision, recall and F_1-score which we obtained for the different iterations. While the recall of our approach was consistently 100%, recall for the OWASP tool varied between 62.5% and 91.38%. This fluctuation in OWSAP's recall was caused by the fact OWASP requires JAR files to be available to be able to map the files. However, not all projects hosted in Maven are distributed with their JAR files.

Table 12
Precision and recall evaluation.

		Precision		Recall		F_1-score	
		OWASP	SE-GPS	OWASP	SE-GPS	OWASP	SE-GPS
Data points	**7**	83.33%	80.00%	62.50%	100.00%	71.43%	88.89%
	17	93.33%	81.81%	77.78%	100.00%	84.85%	90.00%
	27	96.00%	87.50%	85.71%	100.00%	90.57%	93.33%
	37	97.06%	90.47%	86.84%	100.00%	91.67%	95.00%
	47	97.73%	92.30%	89.58%	100.00%	93.48%	96.00%
	57	98.15%	93.54%	91.38%	100.00%	94.64%	96.66%
	67	96.72%	94.44%	86.76%	100.00%	91.47%	97.14%
	77	97.10%	95.12%	85.90%	100.00%	91.16%	97.50%
	87	97.10%	95.60%	76.14%	100.00%	85.35%	97.75%
	Avg:	**95.17%**	**90.09%**	**82.51%**	**100.00%**	**88.29%**	**94.70%**

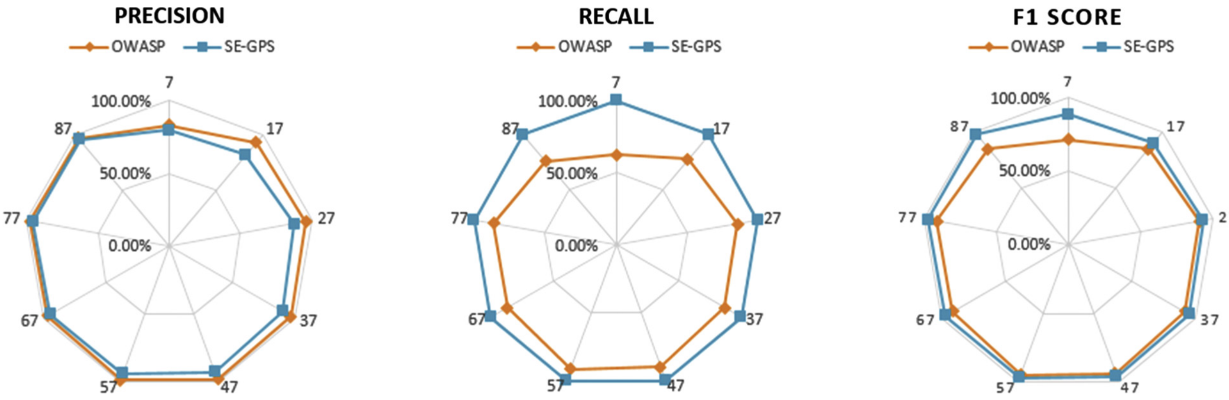


Fig. 16. Precision, Recall and F1 evaluations between SE-GPS and OWASP.

We also observed that OWASP's precision was on average 5% higher than for our approach, caused by the imprecisions in our matching algorithm. Our matching algorithm considers project dependencies with similar name and version number (e.g., ***org.apache.cxf:cxf:3.0.1***, ***org.apache.geronim.configs:cxf:3.0.1*** and ***org.apache.geronimo.plugins:cxf:3.0.1***) to be incorrectly referring to the same project.

Vulnerability patches and usage. The change-list of programming constructs used to the actual usage of vulnerable code fragments in vulnerable components depends on the availability of patch information. In NVD however, not all identified vulnerabilities include a complete references to their patches. Furthermore, we also observed cases, where these references exist only as a textual description of the patch instead of a URL to the actual source commit, limiting our ability to automatically extract the source code information related to such a particular patch.

6.2.2. External validity

The presented approach for identifying transitive dependencies might not be generalizable for non-MAVEN projects, since the presented case studies are based only on the use of the Maven dependency management system. However, given the flexibility and openness of our knowledge modeling approach, also dependency information from build repositories or resources other than Maven can be integrated in our approach. The quality of our analysis will however depend on the ability to extract these dependencies accurately. While the fact extraction process for other build systems (e.g. Ant,[22] Gradle[23] and MSBuild[24]) differs from the one we used for Maven, the core domain concepts remain the same for all of these repositories.

Another threat to validity for our research is that our evaluation has mainly focused on a quantitative analysis of the results from the case studies, limiting our ability to generalize the applicability and validity of the approach. In order to mitigate this threat, an additional qualitative analysis has to be performed in the form of user studies, which will allow for an evaluation of both, the applicability of the approach (e.g., SE-GPS) and the analysis of the result sets from an expert user perspective.

[22] http://ant.apache.org/.

[23] https://gradle.org/.

[24] https://github.com/microsoft/msbuild.

7. Related work

Analyzing software project artifacts at the security detection level has recently become a very active area of research. Such analysis provides valuable insights, when it comes to detect security-related discussions [45], identifying security/non-security related bug reports [46] or predicting vulnerable software components [47]. However, to the best of our knowledge, none of the existing research provides a Semantic Web based infrastructure that supports the semantic linking between security vulnerabilities reported in specialized software security repositories and traditional software repositories. Therefore, we briefly provide an overview of related work, including research on ontologies in cybersecurity and tracking unknown security vulnerabilities in software projects.

Cybersecurity related linked data for software products: Mulwad et al. [48] proposed an approach to detect and extract security vulnerabilities from text extracted from Web pages using trained Name Entity Recognition tool[25] and classification tools [49]. Their prototype uses Wikitology, a general purpose knowledge base derived from Wikipedia[26] as well as Yago [50] and Freebase [51]. The main contribution of their approach is that it extracts related software security concepts from text found on the Web and mapping them to related concepts from Wikitology [52]. Additionally, Joshi et al. [36] proposed a framework for creating ontologies from NVD entries as part of their approach. They identify software security entities and concepts extracted from [53] and link them to their own classifications based on DBpedia Spotlight [54].

Several other frameworks have been introduced in the literature, which take advantage of information available on the Internet. In [48], a framework was introduced which also took advantage of Wikitology [52] and in [36] a framework based on DBpedia was introduced. Both systems use slightly modified version of the Intrusion Detection System (IDS) ontology proposed by More et al. [33] to link cybersecurity entities and concepts with relevant terms extracted from the Internet.

Similar to [37] and [48], our approach also takes advantage of Semantic Web to capture, formalize, and instantiate different software vulnerabilities and their associated properties. However, our approach differs from these approaches, by unifying heterogeneous datasets (security and non-security related software sources) to explicitly model security concepts and link them across repository boundaries. Given our unified representation, we can detect propagations of known security vulnerabilities in software products and across large software ecosystems.

Tracking security vulnerabilities: A number of static analysis tools exist (e.g., [55]) that identify vulnerability in the source code. However, the objective of these tools differs significantly from our approach, by identifying and tracking security vulnerabilities only for a given project. This is in contrast to our approach, which focuses on a global dependency analysis of vulnerabilities using different sources of information. Mitropoulos et al. [56] and Saini et al. [57] used one of the known static analysis tool (e.g., FindBugs [58]), to find major security defects in Java source codes. The collected information was further used to study the evolution of security-related bugs in a given project [59]. While their approach detects security defects in the source code, our approach focuses on usage of known security vulnerabilities in software components at a global scale.

Mircea et al. [60] introduce their Vulnerability Alert Service (VAS) tool to notify users if a vulnerability is reported for a software systems. VAS depends on the OWASP Dependency-Check tool, which we compare with our SE-GPS approach in Section 6. VAS reports the vulnerable projects identified by the OWASP tool without further investigation; and just like OWASP, VAS does not support transitive dependencies analysis of vulnerable components.

8. Conclusions and future directions

In this paper, we introduce a Semantic Web based approach, which uses a unified ontological representation to establish bi-directional traceability links between security vulnerabilities databases and traditional software repositories. This modeling approach not only eliminates some of the traditional information silos in which these data resources have been resided, but also provides the foundation for various types of dependency analysis. More specifically, our approach currently supports the linking of vulnerabilities reported by NVD to projects captured by the Maven build repository. Given the expressiveness of our ontological knowledge representation, we can now take advantage of semantic inference services to determine both direct and transitive dependencies between reported vulnerabilities and potentially affected Maven projects. We also introduce a tool prototype to illustrate how our analysis results can be made accessible to developers. Through several case studies, we showed the applicability of our approach, highlighting the potential impact of reusing vulnerable components in a global software ecosystem context.

As part of our future work, we will focus on investigating potential vulnerability patterns based on the usage of components to provide additional insights in assessing and predicting the quality of software systems. We also plan to extend our SE-GPS tool, to include a software developer's context in the dependency analysis to further improve the relevance of analysis results. While our current version of SE-GPS does not support automatic vulnerability notifications, we plan to include such notifications in our next release of SE-GPS, by taking advantage of a developer's work and task context to provide situation aware analysis results.

[25] http://nlp.stanford.edu/ner/.

[26] http://www.wikipedia.org/.

References

[1] P. Vermesan, Ovidiu and Friess, Internet of Things: Converging Technologies for Smart Environments and Integrated Ecosystems, River Publishers, 2013.

[2] C. Jones, Globalization of software supply and demand, IEEE Softw. (1994) 17–24.

[3] A. Dolstra, NixOS: a purely functional Linux distribution, ACM SIGPLAN Not. 43 (2008) 367–378.

[4] T. Premkumar Devanbu, S. Stubblebine, Software engineering for security: a roadmap, in: Proc. Conf. Futur. Softw. Eng., ACM, 2000, pp. 227–239.

[5] NIST, National Vulnerability Database, http://web.nvd.nist.gov/view/vuln/search, 2007 (accessed December 15, 2014).

[6] R. Oliveto, G. Antoniol, A. Marcus, J. Hayes, Software artefact traceability: the never-ending challenge, in: IEEE Int. Conf. Softw. Maint., IEEE, 2007, pp. 485–488.

[7] N.F. Noy, Semantic integration: a survey of ontology-based approaches, ACM SIGMOD Rec. 33 (2004) 65, http://dx.doi.org/10.1145/1041410.1041421.

[8] Q. Yang, X. Wu, 10 challenging problems in data mining research, Int. J. Inf. Technol. Decis. Mak. 05 (2006) 597–604, http://dx.doi.org/10.1142/S0219622006002258.

[9] Y.M. Mileva, V. Dallmeier, M. Burger, A. Zeller, Mining trends of library usage, in: Proc. Jt. Int. Annu. ERCIM Work. Princ. Softw. Evol. Softw. Evol. Work, ACM Press, New York, New York, USA, 2009, pp. 57–62.

[10] R. Studer, V.R. Benjamins, D. Fensel, Knowledge engineering: principles and methods, Data Knowl. Eng. 25 (1–2) (1998) 161–197, http://dx.doi.org/10.1016/S0169-023X(97)00056-6.

[11] R. Laurini, Pre-consensus ontologies and urban databases, in: Ontologies for Urban Databases, 2007, pp. 27–36.

[12] F. Baader, I. Horrocks, U. Sattler, Description logics as ontology languages for the Semantic Web, in: Mech. Math. Reason., 2005, pp. 228–248.

[13] T. Berners-Lee, J. Hendler, O. Lassila, The Semantic Web, Sci. Am. 284 (2001) 34–43, http://dx.doi.org/10.1038/scientificamerican0501-34.

[14] W.O.W. Group, OWL 2 web ontology language document overview, second edition, http://www.w3.org/TR/owl2-overview/, 2012 (accessed December 1, 2014).

[15] C.J.H. Mann, The description logic handbook – theory, implementation and applications, Kybernetes 32 (2003), http://dx.doi.org/10.1108/k.2003.06732iae.006.

[16] S. Chabot, A review of "a semantic web primer,", J. Web Librariansh. 4 (2010) 97–98, http://dx.doi.org/10.1080/19322900903565408.

[17] Apache, Apache Jena https://jena.apache.org/, 2000 (accessed January 10, 2015).

[18] O. Software, OpenLink http://virtuoso.openlinksw.com/, 1992 (accessed January 10, 2015).

[19] J. Aasman, Allegro graph: RDF triple database, http://franz.com/agraph/allegrograph/, 2006 (accessed January 10, 2015).

[20] M. Würsch, G. Ghezzi, M. Hert, G. Reif, H.C. Gall, SEON: a pyramid of ontologies for software evolution and its applications, Computing 94 (2012) 857–885, http://dx.doi.org/10.1007/s00607-012-0204-1.

[21] I. Sonatype, Maven: The Definitive Guide, O'Reilly, 2008, http://www.sonatype.com/Books/Maven-The-Complete-Reference.

[22] Jeff Williams, A. Dabirsiaghi, The unfortunate reality of insecure libraries, Asp. Secur. Inc. 2012, pp. 1–26.

[23] C. Kiefer, A. Bernstein, J. Tappolet, Mining software repositories with iSPAROL and a software evolution ontology, in: Fourth Int. Work. Min. Softw. Repos., IEEE, 2007, 10 pp.

[24] H. Happel, A. Korthaus, S. Seedorf, P. Tomczyk, KOntoR: an ontology-enabled approach to software reuse, in: Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering, SEKE 2006, San Francisco, CA, USA, July 5–7, 2006, Knowledge Systems Inst. Graduate School, Skokie, Ill, 2006, pp. 349–354.

[25] P. Devanbu, R. Brachman, P.G. Selfridge, LaSSIE: a knowledge-based software information system, Commun. ACM 34 (1991) 34–49, http://dx.doi.org/10.1145/103167.103172.

[26] T.B. Lee, WorkFlowOntology, (n.d.) http://www.w3.org/2005/01/wf/ (accessed June 15, 2015).

[27] K.M. de O. Márcio Greyck Batista Dias, Nicolas Anquetil, Organizing the knowledge used in software maintenance, J. Univers. Comput. Sci. 9 (2003) 641–658.

[28] F.G. Francisco Ruiz, Aurora Vizcaíno Barceló, Mario Piattini, An ontology for the management of software maintenance projects, Int. J. Softw. Eng. Knowl. Eng. 14 (2004) 323–349.

[29] Almut Herzog, Nahid Shahmehri, C. Duma, An ontology of information security, Int. J. Inf. Secur. Priv. 1 (2007) 1–23.

[30] V. Khadilkar, J. Rachapalli, Sematic Web Implementaion Schema for National Vulnerability Database (Common Platform Enumeration Data), University of Texas, Dallas, 2010.

[31] A. Bizer, Christian, Seaborne, D2RQ-treating non-RDF databases as virtual RDF graphs, in: Proc. 3rd Int. Semant. Web Conf., Citeseer Hiroshima, 2004.

[32] J. Undercoffer, J. Pinkston, A. Joshi, A target-centric ontology for intrusion detection, in: Proc. IJCAI-03 Work. Ontol. Distrib. Syst., Morgan Kaufmann Publ., 2004, pp. 47–58.

[33] S. More, M. Matthews, A. Joshi, T. Finin, A knowledge-based approach to intrusion detection modeling, in: IEEE Symp. Secur. Priv. Work, IEEE, 2012, pp. 75–81.

[34] M. Iannacone, S. Bohn, G. Nakamura, J. Gerth, K. Huffer, R. Bridges, et al., Developing an ontology for cyber security knowledge graphs, in: Proc. 10th Annu. Cyber Inf. Secur. Res. Conf, ACM Press, New York, New York, USA, 2015, pp. 1–4.

[35] J. Undercoffer, A. Joshi, J. Pinkston, Modeling computer attacks: an ontology for intrusion detection, in: Recent Adv. Intrusion Detect, 2003, pp. 113–135.

[36] A. Joshi, R. Lal, T. Finin, A. Joshi, Extracting cybersecurity related linked data from text, in: IEEE Seventh Int. Conf. Semant. Comput., IEEE, 2013, pp. 252–259.

[37] A. Vorobiev, Jun Han, Security attack ontology for web services, in: Second Int. Conf. Semant. Knowl. Grid, IEEE, 2006, 42 pp.

[38] L. Coppolino, S. D'Antonio, I.A. Elia, L. Romano, From intrusion detection to intrusion detection and diagnosis: an ontology-based approach, in: Softw. Technol. Embed. Ubiquitous Syst., 2009, pp. 192–202.

[39] B. Martin, M. Brown, A. Paller, D. Kirby, S. Christey, 2010 CWE/SANS top 25 most dangerous software errors. MITRE, 2010.

[40] Y. Ballim, Afzal, Wilks, Artificial Believers: The Ascription of Belief, Psychology Press, 1991.

[41] L. Otero-Cerdeira, F.J. Rodríguez-Martínez, A. Gómez-Rodríguez, Ontology matching: a literature review, Expert Syst. Appl. 42 (2015) 949–971, http://dx.doi.org/10.1016/j.eswa.2014.08.032.

[42] S.S. Jeremy Long, OWASP dependency check https://www.owasp.org/index.php/OWASP_Dependency_Check, 2015 (accessed March 10, 2015).

[43] S.S. Alqahtani, E.E. Eghan, J. Rilling, SE-GPS http://aseg.cs.concordia.ca/segps, 2015 (accessed September 26, 2015).

[44] C.D. Manning, P. Raghavan, H. Schutze, Introduction to Information Retrieval, Cambridge University Press, 2008.

[45] D. Pletea, B. Vasilescu, A. Serebrenik, Security and emotion: sentiment analysis of security discussions on GitHub, in: Proc. 11th Work. Conf. Min. Softw. Repos., ACM Press, New York, New York, USA, 2014, pp. 348–351.

[46] M. Gegick, P. Rotella, T. Xie, Identifying security bug reports via text mining: an industrial case study, in: 7th IEEE Work. Conf. Min. Softw. Repos., IEEE, 2010, pp. 11–20.

[47] S. Neuhaus, T. Zimmermann, C. Holler, A. Zeller, Predicting vulnerable software components, in: Proc. 14th ACM Conf. Comput. Commun. Secur., ACM Press, New York, New York, USA, 2007, p. 529.

[48] V. Mulwad, W. Li, A. Joshi, T. Finin, K. Viswanathan, Extracting information about security vulnerabilities from web text, in: IEEE/WIC/ACM Int. Conf. Web Intell. Intell. Agent Technol., IEEE, 2011, pp. 257–260.

[49] E. Osuna, R. Freund, F. Girosi, An improved training algorithm for support vector machines, in: Neural Networks Signal Process. VII. Proc. 1997 IEEE Signal Process. Soc. Work., IEEE, 1997, pp. 276–285.

[50] F.M. Suchanek, G. Kasneci, G. Weikum, Yago: a core of semantic knowledge, in: Proc. 16th Int. Conf. World Wide Web, ACM Press, New York, New York, USA, 2007, p. 697.

[51] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, J. Taylor, Freebase: a collaboratively created graph database for structuring human knowledge, in: Proc. ACM SIGMOD Int. Conf. Manag. Data, ACM Press, New York, New York, USA, 2008, p. 1247.

[52] Z.S. Syed, T. Finin, A. Joshi, Wikitology: Using Wikipedia as an Ontology, University of Maryland, Baltimore, MD, USA, 2008.

[53] R. Lal, Information Extraction of Cyber Security Related Terms and Concepts from Unstructured Text, University of Maryland, Baltimore, MD, USA, 2013.

[54] P.N. Mendes, M. Jakob, A. García-Silva, C. Bizer, DBpedia spotlight: shedding light on the web of documents, in: Proc. 7th Int. Conf. Semant. Syst. – I-Semantics '11, ACM Press, New York, New York, USA, 2011, pp. 1–8.

[55] N. Rutar, C.B. Almazan, J.S. Foster, A comparison of bug finding tools for Java, in: 15th Int. Symp. Softw. Reliab. Eng., IEEE, 2004, pp. 245–256.

[56] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios, D. Spinellis, The bug catalog of the Maven ecosystem, in: Proc. 11th Work. Conf. Min. Softw. Repos., ACM Press, New York, New York, USA, 2014, pp. 372–375.

[57] V. Saini, H. Sajnani, J. Ossher, C.V. Lopes, A dataset for Maven artifacts and bug patterns found in them, in: Proc. 11th Work. Conf. Min. Softw. Repos., ACM Press, New York, New York, USA, 2014, pp. 416–419.

[58] D. Hovemeyer, W. Pugh, Finding bugs is easy, ACM SIGPLAN Not. 39 (2004) 92, http://dx.doi.org/10.1145/1052883.1052895.

[59] D. Mitropoulos, G. Gousios, D. Spinellis, Measuring the occurrence of security-related bugs through software evolution, in: 16th Panhellenic Conf. Informatics, IEEE, 2012, pp. 117–122.

[60] M. Cadariu, E. Bouwers, J. Visser, A. van Deursen, Tracking known security vulnerabilities in proprietary software systems, in: IEEE 22nd Int. Conf. Softw. Anal. Evol. Reengineering, IEEE, 2015, pp. 516–519.