2016 IEEE 27th International Symposium on Software Reliability Engineering

SV-AF – A Security Vulnerability Analysis Framework

Sultan S. Alqahtani Ellis E. Eghan Juergen Rilling
Department of Computer Science and Software Engineering
Concordia University
Montreal, Canada
{s_alqaht, e_eghan}@encs.concordia.ca, juergen.rilling@concordia.ca

Abstract - The globalization of the software industry has introduced a widespread use of system components across traditional system boundaries. Due to this global reuse, also vulnerabilities and security concerns are no longer limited in their scope to individual systems but instead can now affect global software ecosystems. While known vulnerabilities and security concerns are reported in specialized vulnerability databases, these repositories often remain information silos. In this research, we introduce a modeling approach, which eliminates these silos by linking security knowledge with other software artifacts to improve traceability and trust in software products.

In our approach, we introduce a Security Vulnerabilities Analysis Framework (SV-AF) to support evidence based vulnerability detection. Two case studies are presented to illustrate the applicability of our presented approach. In these case studies, we link the NVD vulnerability databases and the Maven build repository to trace vulnerabilities across repository and project boundaries. In our analysis, we identify that 750 Maven project releases are directly affected by known security vulnerabilities and by considering transitive dependencies, an additional 415604 Maven projects can be identified as potentially affected by these vulnerabilities.

Keywords—security vulnerabilities, software quality, software build systems, software dependencies.

I. Introduction

The Internet has not only changed our society but also was key to enabling the globalization of the software industry, with knowledge and information sharing becoming a central part of software development processes [1], [2]. As a result of this globalization, traditional project boundaries have been replaced with a free flow of information, resources and knowledge across projects. For example, open source software is published on the Internet through specialized code sharing portals e.g., Sourceforge[1], GitHub[2], to allow for components to be reused and extended by project developers. At the same time, this global reuse also introduces new challenges to the software engineering community, since not only components but also problems and vulnerabilities found in these reused components are shared. The situation is further exacerbated by the fact that existing analysis tools, developed for project level support typically do not scale to these new global software and development contexts. As a result, Information Security (IS) has emerged as a major challenge for the software domain and has become an integrated part of today's software development processes [2]. Specialized advisory databases (e.g. National Vulnerability Database (NVD)[3]) have been introduced to provide central repositories for tracking software vulnerabilities and potential solutions to resolve them. However, while these databases are knowledge rich resources, they have often remained information silos, disconnected from other knowledge in the software development domain, such as code or issue tracker repositories.

There exist several reasons for these information silos: 1.) A lack of standardized formalism for representing knowledge in the software engineering domain.. 2.) The inability to integrate seamlessly heterogeneous knowledge **resources** that would allow for both, establishing semantic links across existing knowledge and inferring new knowledge**.** 3.) No uniform resource identifiers across knowledge resources that support fact and analysis results sharing for consumption by either humans or machines.

Given the growing importance of IS for the software domain and the challenges the software community faces in integrating heterogeneous knowledge resources, this paper introduces a modeling approach that addresses this traceability challenge. More specifically, our approach takes advantage of the Semantic Web and its supporting technologies (e.g., ontologies, Linked Data, reasoning services) to establish a unified representation that supports knowledge integration across repository boundaries. In addition, using ontologies and Linked Data we can now enrich these repositories with explicit and implicit semantic links and take advantage of Semantic Web reasoning services, to create true information hubs. We introduce a Security Vulnerabilities Analysis Framework (SV-AF), which not only establishes traceability links between security databases and software repositories, but also enables practitioners being notified about potential security vulnerabilities introduced due to the indirect dependencies within their systems.

The remainder of this paper is organized as follows: Section II describes in more detail background relevant to our research.

[1] www.sourceforge.net
[2] www.github.com
[3] www.nvd.nist.gov

2332-6549/16 $31.00 © 2016 IEEE
DOI 10.1109/ISSRE.2016.12

Section III describes details of our proposed SV-AF. Section IV explains the methodology used to instantiate the framework. Section V discusses our case study design and findings. Section VI provides a discussion of our findings and potential threats to the validity. Section VII compares our work with related work, followed by Section VIII, which concludes the paper and discusses future work.

II. Preliminaries

A. *Software Build Systems*

Build systems transform the source code of a software system into deliverables by managing required project dependencies and automating the build process. Build files are either stored in source code repositories together with a project's source code (e.g. Ant build files) or in specialized build repositories, such as the Maven central repository[4]. The Maven repository is a large collection of Java artifacts to allow organizations to publish and make their software components available to developers. The Maven repository contains over 1 million artifacts (e.g., JAR files, source code, Javadoc), with each artifact being uniquely identified by its *groupId*, *artifactId* and *version number*. As part of the Maven build process, a software project defines through xml configuration files (POM files), unidirectional project dependencies on other artifacts. Upon a project build, Maven dynamically downloads all dependent Java libraries and plug-ins from the Maven central repository into a local cache to be accessible during the project build.

B. *Security Vulnerability Databases*

In the software security domain, a software vulnerability refers to mistakes or 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 [3]. The discovery of new software vulnerability is often first reported in software repositories (e.g., issue trackers, mailing lists) of the affected projects or discussed on Q&A sites (e.g., StackOverflow[5]). A common characteristic of such early vulnerability reporting is that descriptions (information) of vulnerabilities are dispersed across multiple resources, but also the descriptions tend to be incomplete, inconsistent and ambiguous across resources. Advisory databases (e.g., NVD) were introduced to address some of these shortcomings. Their key objective is to provide a central resource for reporting vulnerabilities, but also to standardize the reporting of vulnerabilities. To facilitate this standardization process, a **Common Vulnerabilities and Exposures** (CVE) dataset has been introduced to create a publically available dictionary for vulnerabilities to allow for a more consistent and concise use of security terminology. Once a new vulnerability is revealed and verified by security experts, this new vulnerability and other relevant information (e.g., unique identifier, the source URL, affected resources and related vulnerabilities from the same family group) will be added to the CVE database.

In addition to the CVE entry, the vulnerability will also be classified in the **Common Weakness Enumeration** (CWE) database. The CWE provides a common language to describe software security weaknesses and classifies them based on their reported weaknesses. NVD, CVE, and CWE are all part of a global effort to manage the reporting and classification of software vulnerabilities.

C. *The Semantic Web*

The Semantic Web has been defined by Berners-Lee et al. as "*an extension of the Web, in which information is given well-defined meaning, better enabling computers and people to work in cooperation*" [4]. It forms a Web from documents to data, where data should be accessed using the general Web architecture (e.g., URIs). Using this Semantic Web infrastructure allows data to be linked, just as documents (or portions of documents) are already, allowing data to be shared and reused across application, enterprise, and community boundaries. In a Semantic Web, data can be processed by computers as well as by humans, including inferring new relationships among pieces of data. For machines to understand and reason about knowledge, this knowledge needs to be represented in a well-defined, machine readable language. Ontologies provide a formal and explicit way to specify concepts and relationships in a domain of discourse. The Semantic Web uses the Resource Description Framework (RDF) as its data model to formalize the meta-data as subject-predicate-object triples, which are stored in triple-stores. Triple-stores are Database Management Systems (DBMS) for data modeled using RDF. Unlike Relational Database Management Systems (RDBMS), which store data in relations (or tables) and are queried using SQL, triple-stores store RDF triples and are queried using SPARQL [4]. The RDF data-model is domain independent and users define ontologies using an ontology definition language. The Web Ontology Language (OWL) [5] is an example of such a definition language and has been standardized by the W3C[6]. It supports the creation of machine understandable information to enable Web resources to be automatically processed and integrated. The sub-language, OWL-DL, is based on Description Logics (DLs)[6]. DL is a logic-based formalism using predicate calculus to define facts that can formally describe a domain. Therefore, DLs are a set of axioms called a **TBox** (e.g.$Doctor \sqsubseteq Person$) and set of facts called **ABox** (e.g. *{Parent(John), hasChild(John, Mary)}*). Both **TBox** and **ABox** form a knowledge Base (KB) and often written $\mathcal{K} = \prec \mathcal{T}, \mathcal{A} \succ$. The RDF data-model forms a graph where nodes (subject, object) are connected through edges (predicates). The SPARQL query language [7] is used to retrieve information from RDF data-model graphs.

III. Security Vulnerability Analysis Framework

A. *Knowledge Modeling*

One of the key premises of the Semantic Web is its ability to share and extend existing knowledge. Our knowledge modeling approach builds upon this premise, by reusing and

[4] http://search.maven.org
[5] http://www.stackoverflow.com
[6] https://www.w3.org/

extending the software engineering ontologies introduced in [8]. More specifically, we extend these ontologies, by focusing not only the semantic integration of additional traditional software repositories (e.g., build management) and specialized repositories (e.g., vulnerability databases) but also an ontology design that goes beyond the conceptualization of a domain of discourse, by focusing on the inference of new knowledge. We followed a bottom-up modeling approach, where we model first system specific concepts and iteratively abstracted higher-level shared concepts in upper-ontologies (see Fig. 1). The resulting four layer modeling hierarchy is similar to a metadata modeling approach introduced by the Object Management Group (OMG)[7]. Each of these layers differ in terms of their purpose and their design rationale. To improve the readability, we denote OWL classes in *italic,* individuals are underlined and a dashed underline is used for properties. For a complete description of our ontologies, we refer the reader to [9].

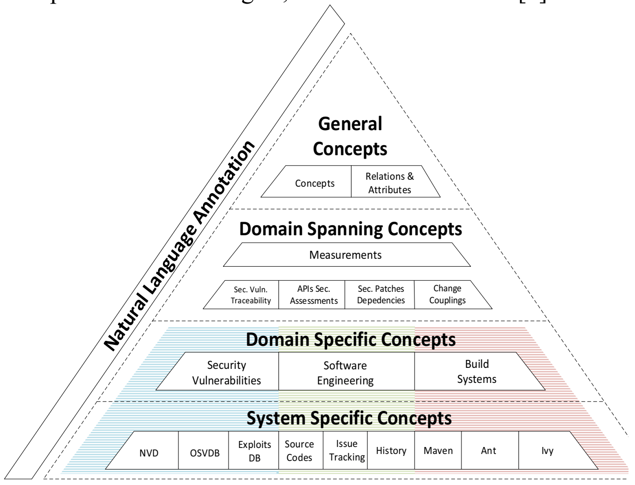

Fig. 1: The SV-AF Ontologies Abstraction Hierarchies

General Concept Layer - Classes in the top-layer model correspond to meta-meta level concepts - core concepts shared and extended by the lower modeling layers. Examples for such core concepts are: *Product, Reference, Activity, Stakeholder,* or *Artifact*. All concepts in this layer are subclasses of the *SeonThing* class (a subclass of *owl:Thing*, which captures the set of all individuals within our framework). Similarly the datatype properties and object properties in this layer are generic and shared across the abstraction layers. For example, the dependsOn object property captures the generic relationship between things - one *Product* dependsOn another *Artifact*.

Domain-Spanning Concepts – In this layer, concepts describe knowledge that is typically inferred from two or more ontologies. For example the measurements ontology acts as a general linking mechanism between ontologies. The ontology provides two basic concepts, *BaseMeasure* or *DerivedMeasure*. Adequate *BaseMeasure* instances are the size and numberOfDependencies in a *Product*. *DerivedMeasure* captures an aggregation of values from different subdomains. For example, the *DerivedMeasure* class includes the numberOfVulnerabilitiesPerApi instance, which is computed from measures collected from the source code, history, build system and the vulnerability ontologies. *SimilarityMeasure*, which is a subclass of *DerivedMeasure*, captures the similarity ([0,1]) between any two *SeonThing* instances.

Domain-Specific Concepts - The third layer in our knowledge model captures domain specific aspects; concepts that are common and reused across resources in a particular domain (e.g., domain of issue trackers). At the core of the domain specific layer we have several domain ontologies: (1) Software sEcurity Vulnerability ONTologies (SEVONT), (2) Software Evolution ONtologies (SEON) [8] and (3) Software Build Systems ONtologies (SBSON). For example, security databases are capturing a *Vulnerability* that has an associated *Event*. An event often can be further divided into *Action* and *Impact* - an attacker exploits a *Vulnerability* to produce an *Action*, which has an *Impact*.

System-Specific concepts - The bottom layer defines systems-specific concepts by extending the domain specific concepts to capture knowledge specific to a particular knowledge resource. For example, the system specific ontology for NVD extends the general SEVONT ontology with NVD specific information on the severity of vulnerabilities by adding a *Severity* concept.

B. Knowledge Engineering and Result Integration

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.

Knowledge Base engineering: In contrast to the top-down approach often used by knowledge engineers, we follow a data-driven, bottom-up approach (Fig. 2).

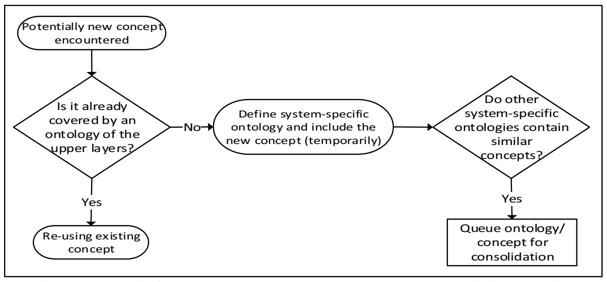
Potentially new concept encountered
Is it already covered by an ontology of the upper layers?
No
Define system-specific ontology and include the new concept (temporarily)
Do other system-specific ontologies contain similar concepts?
Yes
Re-using existing concept
Yes
Queue ontology/ concept for consolidation

Fig. 2: Knowledge engineering process to support result integration

When modeling a new knowledge resource or integrating new analysis results, during the interception phase, we first conceptualize the domain of discourse, by identifying its major concepts and relations. Before adding a concept to the knowledge base, we verify that a similar concept has not yet previously modeled in any of the upper SV-AF's layers, e.g.,

[7] http://www.omg.org/

the domain-specific layer, and re-use the existing concepts whenever possible. If no similar concept exist, we temporarily add the concept to its system-specific ontology, before considering consolidating it with other existing concepts. This consolidation process usually is postponed until we reach a sufficient understanding of the problem domain.

For example, given are two similar concepts found in different security vulnerability databases, we will first create two distinct system-specific concepts in both ontologies. Then we compare the results and move if commonalities are identified the two concepts to the domain-specific layer. Concepts modeled in more than one domain are promoted from the domain-specific to the domain-spanning layer.

Result integration: Our approach applies different types of analysis and combines different knowledge resources, it is not realistic to expect that all sources share a single, consistent view at all times. In particular, if one considers result integration, where different resources or analysis approaches might generate results, which are in a possible disagreement. The knowledge engineering community has proposed different approaches to manage such possibly conflicting information sources. For example in [10], an approach is presented that models this disagreement by structuring knowledge into viewpoints and topics. Using this approach *Viewpoints* represent a particular point of view (e.g., information stemming from a particular tool or knowledge resource), whereas *topics* capture knowledge that is relevant to a given subject (e.g., vulnerable artifact). These environments are nested within each other: viewpoints can contain either other viewpoints or topics. Using this nested modeling approach, a topic can now contain knowledge pertaining to its subject, but also other viewpoints, e.g., when the subject is another user [10]. 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 without having to decide on a "correct" set of information, thereby losing information prematurely.

C. *An Example Scenario: Modeling global vulnerability impacts using bi-directional dependencies*

Currently, there are a number of build systems which provide users with support for managing both internal components and external API dependencies.

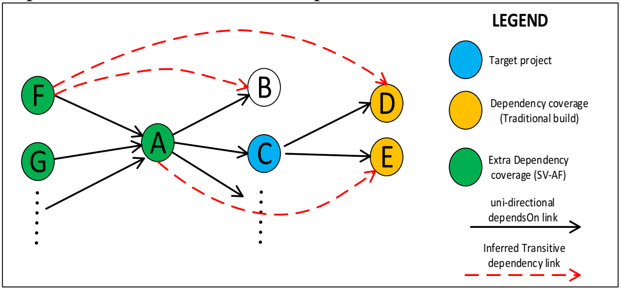
LEGEND
Target project
Dependency coverage (Traditional build)
Extra Dependency coverage (SV-AF)
uni-directional dependsOn link
Inferred Transitive dependency link
F
G
A
B
C
D
E

Fig. 3: Unidirectional vs. bi-directional dependencies

However, while such a unidirectional dependency model works well for managing build dependencies, it restricts a user's ability to further reason upon this knowledge. For example, using Maven, it is currently not possible for a user to identify all components or projects that depend either directly or indirectly on a specific project (see Fig. 3).To overcome this challenge, we take advantage of the Semantic Web and its standardized knowledge modeling approach, by introducing our SBSON ontology to capture the dependencies in the Maven repository.

Using SBSON we are now able to create a global bi-directional project dependency graph, which supports extra semantic analysis by taking advantage of semantic reasoning services. For example, in Fig. 3, using SBSON we can extend the Maven supported impact analysis on project *C*, by not only identify all components on which project C depends on (projects *D* and *E*), but also all projects which might depend on project C (projects *A*, *F* and *G*).

As discussed before, our SV-AF knowledge modeling approach allow analysis approaches to take advantage of the bi-directional dependencies in our knowledge model. In what follows, we not only illustrate how the Maven repository can be seamlessly integrated with NVD by modelling relevant concepts and their relations across the different abstraction layers in our knowledge modeling approach. We provide a concrete usage scenario, how our unified representation can support now for example impact analysis of known vulnerabilities across heterogeneous software repositories (NVD and Maven). The OWL classes and object properties used for the impact analysis example are shown in Fig. 4 (data properties have been omitted to improve readability of the figure).

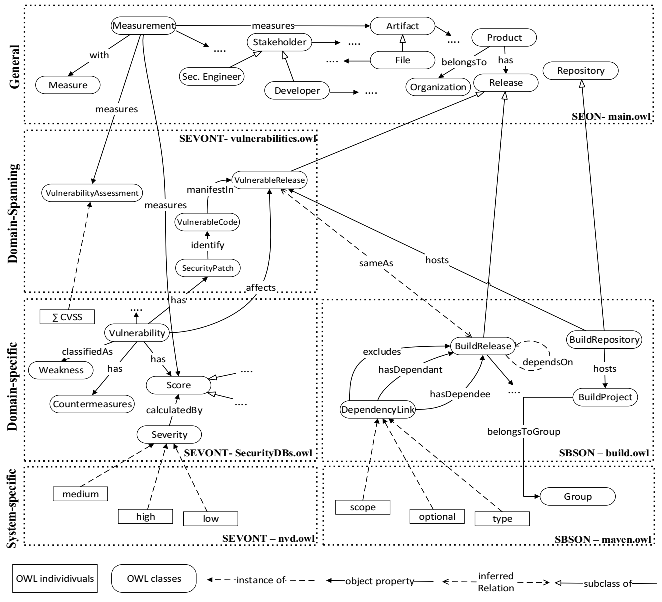

Fig. 4: SV-EF's ontologies and concepts involved in software vulnerability dependencies analysis

Modeling Vulnerable Release dependences: A *VulnerableRelease* is a software *Release* within the NVD database with a known *Vulnerability. A BuildRelease is* a software release within the Maven ecosystem. Using our ontology alignment process, we infer that a given

VulnerableRelease is sameAs a specific *BuildRelease*–. As such, the *VulnerableRelease* inherits the properties of the original *BuildRelease*, for example, the *VulnerableRelease* now dependsOn other *BuildRelease*. Given the support for bi-directional links in our model, a *Project* hosted in an ecosystem's *Repository* can now be identified as being potentially affected by a vulnerability when it directly or indirectly reuses a *VulnerableRelease*.

IV. Methodology

A. Overview

In what follows we introduce in more details our overall methodology (Fig. 5) which consists of three major steps: (i) Fact extraction and population, (ii) ontology alignment and (iii) tracing vulnerabilities across knowledge boundaries using knowledge inferencing/reasoning.

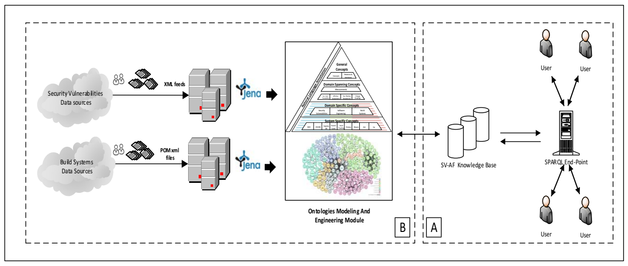
Security Vulnerabilities Data sources
XML feeds
Build Systems Data Sources
POM xml files
Jena
Ontologies Modeling And Engineering Module
B
A
SV-AF Knowledge Base
SPARQL End-Point
User

Fig. 5: SV-AF system overview

B. Facts Extraction and Population

Our SV-AF framework depends on several endogenous and exogenous data sources. **Endogenous data** sources are internal to a software development environment such as source code, issues trackers and build repositories. In contrast, **exogenous data** sources are external to a software development environment, such as vulnerability databases, Q&A sites.

The fact extraction process itself consists of extracting facts from the Maven POM files and the NVD XML update feeds (see Fig. 5 –B). For the ontology population, we use the Jena[8] framework to populate the corresponding artifact ontologies and materialize them using a triples-tore.

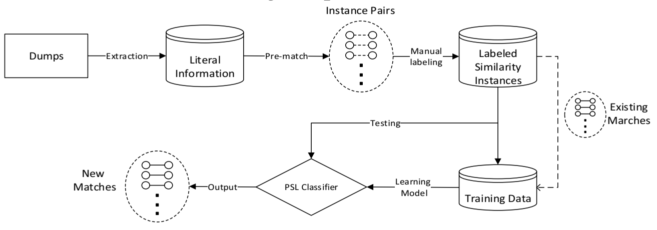

Fig. 6: Instances matching approach

C. Ontology Instances Aligment

For the alignment of instance in our ontologies, we take advantage of the Probabilistic Soft Logic (PSL) framework [11], which establishes weighted links between ontologies (Fig. 6).

PSL uses continuous variables to represent truth values, relaxing the standard Boolean values [11] traditionally used. The resulting probability distribution over literals is captured in a graph model, which can then be reasoned upon. The majority of the rules in PSL are soft-weighted rules, like rules stating that instances are similar if their names or their classes are similar (see Listing 1).

$$\begin{aligned} &\mathbf{1}.\boldsymbol{type}(\boldsymbol{A},\boldsymbol{instance}) \wedge \boldsymbol{type}(\boldsymbol{B},\boldsymbol{instance}) \wedge \boldsymbol{name}(\boldsymbol{A},\boldsymbol{X}) \\ &\wedge \boldsymbol{name}(\boldsymbol{B},\boldsymbol{Y}) \wedge \boldsymbol{similarID}(\boldsymbol{X},\boldsymbol{Y}) \\ &\wedge \boldsymbol{A.source} \neq \boldsymbol{B.source} \\ &\Rightarrow \boldsymbol{similar}(\boldsymbol{A},\boldsymbol{B}) \ \textbf{wight}\colon \mathbf{0.5} \end{aligned}$$

$$\begin{aligned} &2.type(A,instance) \wedge type(B,instance) \wedge name(A,X) \\ &\wedge name(B,Y) \wedge similarID(X,Y) \\ &\wedge A.source \neq B.source \\ &\wedge version(A,Z) \wedge version(B,K) \\ &\wedge similarID(Z,K) \\ &\Rightarrow similar(A,B) \ \text{wight: 0.8} \end{aligned}$$

Listing 1: PSL rules

For example, in Listing 1.1 the first PSL rule states that two instances A, B with similar names defined in different source ontologies are likely to be similar. "*similarID"* is a similarity function implemented using the Levenshtein similarity metric. Rules in PSL are labeled with non-negative weights. In Listing 1.2, the rule weights is used to indicate that projects with same names and versions are more likely to be similar than projects with same names only (Listing 1.1).

Using PSL we can establish owl:sameAs relations between similar instances found in the SEVONT and SBSON ontologies. In this example, two ontologies *NVD* and *Maven* are given as data sources and their corresponding instances $|NVD|$ and $|Maven|$. The number of possible instance pairs for these two ontologies is $|NVD| \times |Maven|$. In our example, similarity among instance pairs is determined based on the extracted literal information such as name, version and vendor. We used the PSL framework classifier to compute the similarity weights for the owl:sameAs links. For training purpose, we created a training dataset with manually labeled instance links to train the PSL classifier to establish the weights for the pre-defined rules. Having derived the semantic similarity weights for each instance pair, we can now assigned these weights to the **owl:sameAs** (see Fig. 7) links between the aligned instances and then materialized the alignment results to our knowledge base. Having the weighted alignment links between the two ontologies, a SPARQL query can now be written, to retrieve the vulnerability information from the *NVD* ontology and their corresponding instances in *Maven* ontology based on a given similarity threshold. For this query, we take advantage of RDFS++[9] reasoning to not only retrieve explicit but also infer implicit facts from the knowledge base. More specifically, our ontology design not only supports transitive but also

[8] jena.apache.org

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

subsumption reasoning, which are not supported by traditional relational databases.

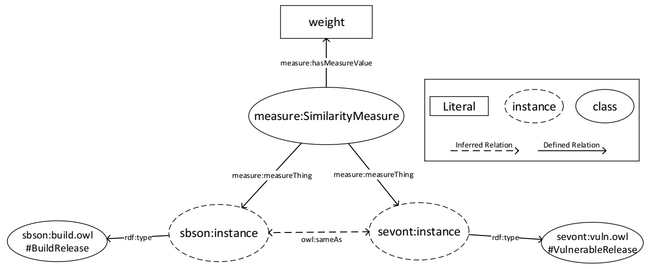

Fig. 7: weighted similarity modeling

D. Knowledge Inferencing and Reasoning

A key feature of many triples-stores is to provide scalability reasoning, by materializing reasoning results. In this section, we discuss how such reasoning capabilities are used in our approach to trace vulnerabilities across knowledge boundaries.

owl:sameAs inference: A commonly used predicate for inferring new knowledge is owl:sameAs, which is used to align two concepts. An example from our SBSON and SEVONT ontologies is shown in Fig. 8.

sevont: *ProjA*	*rdf*: *type*	*sevont*: *VulnerableRelease*
sbson: *ProjB*	*rdf*: *type*	*sbson*: *BuildRelease*
sevont: *ProjA*	*sevont*: *hasVulnerability*	*sevont*: *CVE − ID*
sevont: *ProjA*	**owl**: **sameAs**	*sbson*: *ProjB*.

Fig. 8: owl:sameAs rules example

Given is the following SPARQL query (Listing 2), which takes advantage of the owl:sameAs predicate if inference is enabled:

```
SELECT ?vulnerability ?realse
WHERE{
   ?release  rdf:type                  sbson:BuildRelease.
   ?release  sevont:hasVulnerability   ?vulnerability.
}
```

Listing 2: SPARQL query returning same as projects vulnerabilities

Without inferencing, the query result set would be empty, since no triple has as subject sbson:ProjB and predicate sevont:hasVulnerability. However, with inference enabled, it can now be infer that ProjB has a vulnerability (CVE-ID[10]) through the reasoner being able to establish a link between sbson:ProjB and sevont:ProjA using the owl:sameAs property.

owl:TransitiveProperty inference: A relation R is said to be transitive if R(a,b) and R(b,c) implies R(a,c); this can be expressed in OWL through the owl:TransitiveProperty construct. We define seon:dependsOn to be a bi-directional transitive property of type owl:TransitiveProperty (e.g., seon:dependsOn rdf:type owl:TransitiveProperty). Through 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 Listing 3).

```
SELECT ?project
WHERE{
   ?release   rdf:type                  sbson:BuildRelease.
   ?release   sevont:hasVulnerability   ?vulnerability.
   ?project   seon:dependsOn            ?release.
}
```

Listing 3: SPARQL query returning transitive vulnerable dependencies

V. Case Study

This section introduces the two case studies we use to evaluate the applicability of our knowledge modeling approach. More specifically, in case study #1, we identify semantically similar software projects that exist in Maven and contain known security vulnerabilities disclosed in the NVD database. The objective of this case study is to evaluate the applicability of our alignment process by comparing it against a specialized, existing dependency analysis tool [12]. For the second case study, we illustrate how semantic reasoning can enable semantic richer analysis services. More specifically, we show that our semantic rules can infer explicit and implicit security vulnerabilities by inferring transitive dependencies by traversing the bi-directional links.

A. Case Study Data

For the data collection and extraction in our case studies, we rely on two data sources: the NVD database and the Maven build repository. We download the latest version of the repository from the Maven.org website (Table I) and download all NVD vulnerability xml feeds from 1990 and 2016 (Table II). For case study #1, the number of releases and unique vulnerable products were used to evaluate our alignment approach, for integrating these two ontologies.

Table I: Maven Repository statistics

Repository	Projects	Releases	Last Update
Maven [13]	130,895	1,219,731	2016-01-28 16:34:07 UTC

Table II: NVD database statistics

Repository	# unique vulnerabilities	# unique vulnerable products	Period
NVD [14]	74945	109212	1990 - 2016

For our case study #2, the objective was to identify the potential transitive impact set of some vulnerable components on other systems. For the study, we selected five Apache projects (Table III) hosted in the Maven repository. The main criteria for selecting these projects was that at least some of their releases contain known vulnerabilities (identified in our case study#1) and the functionalities these products provide are widely reused by other projects. These five subjects vary in size (classes and methods) and application domain. Wss4J[11] is a Java implementation of the primary security standards for Web Services, Httpclient[12] is responsible of provides reusable components for client-side authentication, HTTP state management, and HTTP connection management. Apache Derby[13] is an open source relational database implemented

[10] Every CVE-ID is uniquely identified by the letters 'CVE', and eight digits. For example, CVE-2015-0235.

[11] https://ws.apache.org/wss4j/

[12] https://hc.apache.org/httpcomponents-client-ga/

[13] https://db.apache.org/derby/

entirely in Java, Hibernate Validator[14] allows expressing and validating application constraints using annotation-based constraints, and Apache OpenJPA[15] is a Java persistence project that can be used as a stand-alone plain old Java object (POJO) persistence layer or be integrated into any Java EE compliant container.

Table III: Subject systems and sizes for transitive dependencies analysis

ID	Subject Systems	Version	Size	
			Classes	Methods
P1	Wss4j	1.6.16	167	1610
P2	Httpclient	4.1	209	1180
P3	Derby	10.1.1.0	967	16354
P4	Hibernate-validator	4.1.0.Final	325	2642
P5	Openjpa	1.1.0	1201	18640

B. Case Study Results

Case Study 1: *Identifying open source components that are directly susceptible to known security vulnerabilities*

***Objective*:** The goal of this study is to evaluate the performance of our semantic similarity linking approach used to align two domain specific ontologies.

Approach: In order to align (link) these two ontologies (SEVONT and SBSON), we use the PSL framework to align project specific information found in both ontologies. We trained PSL using a corpus of 524 randomly selected project instance pairs for which manually created links. We then executed our PSL alignment rules on this training dataset to train our approach. As a result from this training, two concept instances in these ontologies can now be aligned with a degree of certainty, if A and B, with same names are defined in different ontologies ($\neg SameSource$) and have similar Vendors and same Version numbers. SameName, SimilarVendor, and SameVersion are a similarity functions implemented using a Levenshtein distance metric. In this example, the *SameProject(A,B)* is given a weight of 0.9 (Listing 4), which is based on result from the PSL training set. Fig. 9 shows the PSL inference results for our training dataset, with the weights for the $SameProject(A, B)$ alignment ranging from a minimum of 0.04 to a maximum of 0.42.

Using the semantic rule (Listing 4), PSL can now perform *maximum a posteriori (*MPE) reasoning [11] to infer the most likely values for a set of propositions and observed values for the remaining (evidence) propositions.

$$\begin{aligned}
&\boldsymbol{Source(A, SnA)} \wedge \boldsymbol{Source(B, SnB)}\\
&\quad \wedge \neg \boldsymbol{SameSource(SnA, SnB)}\\
&\quad \wedge \boldsymbol{Name(A, X1)} \wedge \boldsymbol{Name(B, Y1)}\\
&\quad \wedge \boldsymbol{SameName(X1, Y1)}\\
&\quad \wedge \boldsymbol{Vendor(A, X2)} \wedge \boldsymbol{Vendor(B, Y2)}\\
&\quad \wedge \boldsymbol{SimilarVendor(X2, Y2)}\\
&\quad \wedge \boldsymbol{Version(A, X3)} \wedge \boldsymbol{Version(B, Y3)}\\
&\quad \wedge \boldsymbol{SameVersion(X3, Y3)}\\
&\quad \Rightarrow \boldsymbol{SameProject(A, B)}\ \textbf{weight}: \boldsymbol{0.9}
\end{aligned}$$

Listing 4: SameProject Rules

For a full discussion on MPE reasoning, we refer the reader to [11]. The results of the PSL inference is a set of $A \times B$ SameProject weights that range from [0..1], with 0 two concept instances having no similarity and 1 corresponding to 100% similarity among instances.

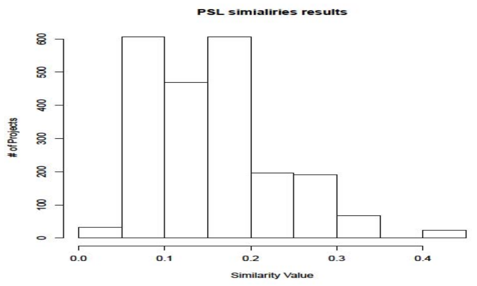

Fig. 9: Probabiltisitc PSL similarities results

As part of our knowledge modeling approach, we materialized the inferred semantic instance links (owl:sameAs) between the SEVONT and SBSON ontology, making this inferred knowledge a persistent part of our knowledge model. We add weights for each link, based on the inferred similarity values using the domain spanning similarity measure (*SimilarityMeasure*) class in our model (Section III-A).

Findings. Our study showed that 0.062% of all Maven projects contain known security vulnerabilities that have been reported in the NVD database. An example for such a vulnerability is shown in Table IV.

Table IV: Example of linked vulnerability

SEVONT fact	SBSON fact	Corresponding Vulnerability
Sevont-securityDB.owl#sonatype:nexus:2.3.1	*Sbson-build.owl#org.sonatype.nexus:nexus:2.3.1*	*Sevont-securityDB.owl#CVE-2014-0792*

Further analysis of our results showed that projects might often suffer from multiple vulnerabilities. We found also that 48.8% of the 750 identified vulnerable project releases suffer from multiple security vulnerabilities, with PostgreSQL 7.4.1 being the most vulnerable project in the repository, containing 25 known vulnerabilities. Giving this additional insight can guide system update decisions and help avoiding the reuse of APIs/components with known security vulnerabilities or components that might be prone to these type of vulnerabilities.

For example, in December 2010, Google released its Nexus S smartphone[16]. The phone was originally running on Android 2.3.3 – an Android version that already contained the security vulnerability discussed in Table V. 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 not supported and distributed for the Nexus S, leaving existing users of the phone susceptible to attacks. Our analysis also showed that the same vulnerability can affect multiple releases of a product. For example, security vulnerability CVE-2013-4787[17] has been reported for five

[14] http://hibernate.org/validator/
[15] http://openjpa.apache.org/
[16] https://en.wikipedia.org/wiki/Nexus_S
[17] https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-4787

different Android versions (Table V). For product maintainers this information can help in to ensure consistent patching and regression testing across product lines or different versions of a product.

Table V: Critical Vulnerabilities for Android Project

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

Evaluation: We evaluate the linking accuracy when aligning project instances (owl:sameAs) between our Maven and NVD ontologies.

During the first step of our evaluation, we compared the impact of the similarity weight thresholds ($w = 0.1$, $w = 0.2$, $w = 0.3$, and $w = 0.4$) in terms of precision, recall and F1 measure on the inferred links created by the PSL alignment process. Precision is calculated, with true positives being the number of project instance pairs correctly classified as similar, while false positives corresponds to the number of non-similar instance pairs that are incorrectly classified as same projects. For Recall, false negatives corresponds to the number of non-similar instance pairs that are incorrectly classified as being similar projects. The F1-score is the harmonic mean of precision and recall, giving equal weight to both measures.

Our analysis (Table VI) showed that an increase in the similarity threshold from 0.1 (low similarity) to 0.4 ((higher similarity) had limited effect on the precision (decrease from 0.98 to 0.75), recall was significantly lower (down from 0.68 to 0.01).

Table VI: owl:sameAs link (w) evaluation

	Precision				
Data Size	w=0.0	**w=0.1**	w=0.2	w=0.3	w=0.4
	0.77	**0.88**	0.98	0.93	0.75
	Recall				
500	0.77	**0.68**	0.30	0.03	0.01
	F1-score				
	0.77	**0.77**	0.46	0.05	0.01

A manual inspection of the inferred links showed that the low recall for the higher threshold values is due to the inconsistent capturing of vendor information within the two ontologies. NVD relies on the common name to identify a vendor, whereas Maven uses the fully qualified package name as the vendor name. For example, using a w=0.0, *org.apache.cxf:cxf:3.0.1*,*org.apache.geronim.configs:cxf:3.0.1* and o*rg.apache.geronimo.plugins:cxf:3.0.1* in SBSON will be considered the same instance as *apache:cxf:3.0.1* in *SEVONT* and therefore correctly linked. However, using a higher similarity threshold, these instances will no longer be linked. We use the similarity weight of $w = 0.1$ in all subsequent experiments due to its high F1-score.

We further evaluated the link quality by comparing our approach against the OWASP Dependency-Check tool [12], a specialized tool, which identifies direct dependencies between projects and publicly disclosed vulnerabilities. For the study, we apply the OWASP dependency check tool on our gold standard (see Section IV.B) and compare the detected dependencies against the links generated by our approach (Table VII). The low OWASP recall is because OWASP requires JAR files to be available to be able to map the files to the vulnerabilities. However, not all projects hosted in Maven are distributed with their JAR files.

Table VII: SV-AF vs. OWASP Dependency Check tool accuracy evaluation

Data Size	**SV-AF w=0.1**			**OWASP**		
	Precision	**Recall**	**F1-score**	**Precision**	**Recall**	**F1-score**
500	0.88	0.68	0.77	0.81	0.26	0.40

Case Study 2: *Identifying open source components that are directly and indirectly dependent on vulnerable components.*

Objective**:** In this study we evaluate how our framework can support the analysis of potential security vulnerability impacts on dependent software components. Furthermore, the case study illustrates the flexibility of our knowledge modeling approach and highlight how additional knowledge resources can be seamlessly integrated and reasoned upon.

Approach: For this case study, we extend our analysis to include transitive closure dependencies (Fig. 10) that not only identifies components that are directly but also indirectly affected by known vulnerabilities. For this impact analysis, we selected 5 open source Java projects (Table III) with known security vulnerabilities for which we do not distinguish if a component actually makes use (calls) a vulnerable component or not.

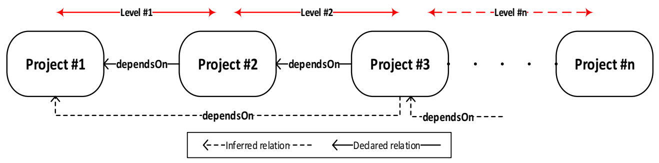

Fig. 10: Inferred project dependencies in SBSON

Findings: In what follows, we summarize the findings from our case study. We report on our transitive dependency analysis, which highlights also the benefits of our knowledge modeling approach, the ability to integrate knowledge resources while taking advantage of inference services provided by the Semantic Web. Given the bi-directional links we established between the NVD and the Maven repository, our analysis is no longer limited to identify whether a project depends on a vulnerable component. Instead, given a vulnerable component, we can now also provide a more holistic analysis, by identifying for a global context, which other projects potentially directly or indirectly depend on this vulnerable component.

Table VIII provides a summary of our analysis. In order to keep the results simple and readable, we consider only three levels of transitivity. For example, the vulnerable project Hibernate-validator 4.1.0 (P4) has a potential impact set of 3805 direct dependent projects (level 1) and 128109 dependent projects when we consider an additional two levels of transitivity (level 3).

Table VIII: Transitive dependencies on vulnerable components

ID	Component Name	# Vulner-abilities	CVE-IDs	Number of dependent components based on transitivity level (L) L1	L2	L3
P1	Wss4j 1.6.16	2	CVE-2015-0227 CVE-2014-3623	336	639	73
P2	Httpclient 4.1	2	CVE-2011-1498 CVE-2014-3577	685	4961	41326
P3	Derby 10.1.1.0	3	CVE-2005-4849 CVE-2006-7216 CVE-2006-7217	385	37999	66147
P4	Hibernate-validator 4.1.0.Final	1	CVE-2014-3558	3805	39295	128109
P5	Openjpa 1.1.0	1	CVE-2013-1768	74	49460	141303

Fig. 11 illustrates a typical usage scenario for modeling approach. While the Geronimo-jetty6-javaee5 (version 2.1.1) itself has no known vulnerability reported, the project depends on several components (level 1 dependencies) with known security issues (5 Java projects with a total of 15 known security vulnerabilities), making also Geronimo-jetty6-javaee5 potentially a very vulnerable component.

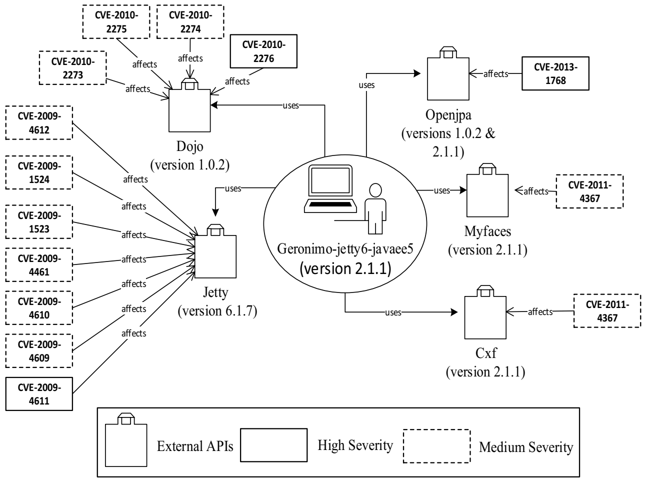

Fig.11: Geronimo-jetty6-javaee5 uses 5 projects (external APIs) from level 1 dependency and each project suffer from security vulnerabilities

VI. Discussion and Threats to Validity

As our experiments illustrate, a unified and formal knowledge modeling approach can indeed help eliminating existing information silos, by seamlessly linking and integrating knowledge resources. For the linking, we take advantage of a probabilistic semantic similarity measure to link instances in our ontologies (e.g., projects in SEVONT and SBSON). 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. In addition, rather than relying on proprietary analysis solutions, our modeling approach takes advantage of the Semantic Web technology stack, including standardized knowledge representation and inference services.

A. *Case Study 1*

Identifying known security vulnerabilities in software projects has been widely discussed in the literature [3], [15]–[17]. However, our approach differs from these existing works in that 1.) It unifies two heterogeneous knowledge resources (software security repositories and build system repositories), using a standardized knowledge representation. 2.) It supports semantic relationships (e.g., owl:sameAs) and RDFS++ reasoning to infer new knowledge (e.g., identify vulnerable transitive dependencies), which are not explicitly found in any of these resources. 3.) Given the bi-directional links, our analysis can go beyond traditional inter project dependencies and include intra project dependencies. The result from our case study shows that the problem of depending on vulnerable third party components with known security vulnerabilities is a common and widespread problem [16].

B. *Case Study 2*

The case study illustrates that vulnerabilities can no longer be treated in a project specific context. With the globalization of the software industry, promoting sharing and integrating of knowledge across knowledge borders, vulnerabilities might have a wide spread impact on the software ecosystem. In our second experiment, the use of transitive properties and reasoning capabilities allow to transform a typical proprietary analysis implementation into a simple, customizable (SPARQL) query approach, which offloads much of the processing to the semantic reasoners. For example, the query in Listing 3, will not only return all projects that are directly but also indirectly depending on a vulnerable component.

C. *Threats to validity*

Internal Validity: An internal validity threat to our approach is that our experiments rely 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. Another internal validity threat is that the instance pair matches for our training set was manually created and potential be prone to human errors. In order to mitigate this, we conducted a cross validation of the annotation, where the links were evaluated by another person. Finally, the size of the dataset used to evaluate our approach might not be considered large enough. To mitigate this threat, we evaluated our approach on dataset sizes to study the effect of the dataset size on our results. Table IX shows a standard deviation of 0.04 and 0.09 for the precision and recall respectively. With the exception of the smallest evaluation size (50 instance pairs), our precision and recall for the various evaluation sizes are very close to the mean, indicating that increasing the dataset size, will most likely have no aversive effect on our results.

Table IX: Dataset size evaluation

Data Points	SV-AF (w=0.1)			
	Precision	|Distance from σ|	Recall	|Distance from σ|
50	0.76	**0.11**	0.38	**0.26**
100	0.87	0.00	0.62	0.02
150	0.88	0.01	0.69	0.05
200	0.9	0.03	0.69	0.05
250	0.89	0.02	0.68	0.04
300	0.86	0.01	0.63	0.01
350	0.87	0.00	0.66	0.02
400	0.87	0.00	0.68	0.04
450	0.88	0.01	0.67	0.03
500	0.88	0.01	0.68	0.04
Avg:	**0.87**	-	**0.64**	-
SD (σ)	**0.04**	-	**0.09**	-

External Validity: In terms of external threats to validity, the presented experiments might not be generalizable for non-MAVEN projects. This threat can be partially mitigated through our modeling approach. Given that our modeling approach is based on different levels of abstraction, we also consider and abstract common aspects of the domain of build repositories in our knowledge model. We do model the domain of build repositories as a domain of discourse in the domain-specific layer of our knowledge model. Another external 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 and the analysis of the result sets from an expert user perspective.

VII. Related work

Software Interrelationship Artifacts across Heterogeneous Ecosystems: In a recent study, Ilo et al. [18] present their Software Relationship Ontology (SWREL) to model information about software interrelationships across different ecosystems. However, their ontology design focuses on the conceptualization rather than the inference of new knowledge. In addition, the semantic linking in SWREL is based on the dependencies relations existing in the Maven repository and Debian[18] package repository. In contrast, our approach has more abstracted and generalizable features which can capture knowledge of different build-systems and package management repositories.

Tracking Security Vulnerabilities: A number of static analysis tools exist (e.g.,[19]) that identify vulnerabilities in the source code. However, their objectives differs from ours, since their focus is on identifying and tracking security vulnerabilities only for a given project. This is in contrast to our approach, which also allows for a global dependency analysis of vulnerabilities using different sources of information. Mitropoulos et al. [20] and Saini et al. [21] used in their approach a static analysis tool (e.g., FindBugs [22]), to locate major security defects in Java source codes. They used the collected information to then further study the evolution of security-related bugs in a given project [23].

Mircea et al. [15] 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 SV-AF approach in Section V. 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.

VIII. Conclusion and future work

In this paper, we introduce a Security Vulnerabilities Evaluation Framework (SV-EF), which introduces a unified ontological representation to establish bi-directional traceability links between security vulnerabilities databases and traditional software repositories. This framework not only eliminates some of the traditional information silos in which data resources have been resided, but also enables different types of dependency analysis. More specifically, our framework 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. Through two experiments, we showed the applicability of our framework, highlighting the potential impact of reusing vulnerable components in a global software ecosystem context.

As part of our future work, we plan to investigate potential vulnerability patterns based on the usage of vulnerable components. These patterns will provide us with additional insights in assessing and predicting the quality of software systems.

Reference

[1] P. Vermesan, Ovidiu and Friess, *Internet of things: converging technologies for smart environments and integrated ecosystems*. River Publishers, 2013.

[2] P. T. Devanbu and S. Stubblebine, "Software engineering for security," in *ICSE '00 Proceedings of the Conference on The Future of Software Engineering*, 2000, pp. 227–239.

[3] A. Williams, Jeff and Dabirsiaghi, "The unfortunate reality of insecure libraries," *Asp. Secur. Inc*, no. March, pp. 1–26, 2012.

[4] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Sci. Am.*, vol. 284, no. 5, pp. 34–43, May 2001.

[5] F. and others McGuinness, Deborah L and Van Harmelen, "OWL web ontology language overview," *W3C Recomm.*, vol. 10, p. 10, 2004.

[6] C. J. H. Mann, "The Description Logic Handbook – Theory, Implementation and Applications," *Kybernetes*, vol. 32, no. 9/10, Dec. 2003.

[7] B. DuCharme, *Learning SPARQL*, 2n Edition. O'Reilly Media, 2011.

[8] M. Würsch, G. Ghezzi, M. Hert, G. Reif, and H. C. Gall,

[18] https://www.debian.org/distrib/packages

"SEON: a pyramid of ontologies for software evolution and its applications," *Computing*, vol. 94, no. 11, pp. 857–885, Nov. 2012.

[9] S. S. Alqahtani, E. E. Eghan, and J. Rilling, "SE-GPS," 2015. [Online]. Available: http://aseg.cs.concordia.ca/segps. [Accessed: 26-Sep-2015].

[10] Y. Ballim, Afzal and Wilks, *Artificial believers: The ascription of belief*. Psychology Press, 1991.

[11] A. Kimmig, S. Bach, M. Broecheler, B. Huang, and L. Getoor, "A short introduction to Probabilistic Soft Logic.," in *Proceedings of NIPS Workshop on Probabilistic Programming: Foundations and Applications (NIPS Workshop-12)*, 2012.

[12] S. S. Jeremy Long, "OWASP Dependency Check," 2015. [Online]. Available: https://www.owasp.org/index.php/OWASP_Dependency_Check. [Accessed: 10-Mar-2015].

[13] A. M. Project, "Maven Central Repository." [Online]. Available: http://search.maven.org/. [Accessed: 15-Dec-2014].

[14] NIST, "National Vulnerability Database," 2007. [Online]. Available: http://web.nvd.nist.gov/view/vuln/search. [Accessed: 15-Dec-2014].

[15] M. Cadariu, E. Bouwers, J. Visser, and A. van Deursen, "Tracking known security vulnerabilities in proprietary software systems," in *IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 516–519.

[16] S. S. Alqahtani, E. E. Eghan, and J. Rilling, "Tracing known security vulnerabilities in software repositories – A Semantic Web enabled modeling approach," *Sci. Comput. Program.*, vol. 121, pp. 153–175, Jun. 2016.

[17] V. Livshits and M. Lam, "Finding security vulnerabilities in Java applications with static analysis," *... 14th Conf. USENIX Secur. ...*, pp. 1–17, 2005.

[18] N. Ilo, J. Grabner, T. Artner, M. Bernhart, and T. Grechenig, "Combining software interrelationship data across heterogeneous software repositories," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 571–575.

[19] N. Rutar, C. B. Almazan, and J. S. Foster, "A Comparison of Bug Finding Tools for Java," in *15th International Symposium on Software Reliability Engineering*, 2004, pp. 245–256.

[20] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios, and D. Spinellis, "The bug catalog of the maven ecosystem," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 372–375.

[21] V. Saini, H. Sajnani, J. Ossher, and C. V. Lopes, "A dataset for maven artifacts and bug patterns found in them," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 416–419.

[22] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Not.*, vol. 39, no. 12, p. 92, Dec. 2004.

[23] D. Mitropoulos, G. Gousios, and D. Spinellis, "Measuring the Occurrence of Security-Related Bugs through Software Evolution," in *16th Panhellenic Conference on Informatics*, 2012, pp. 117–122.