Recovering Semantic Traceability Links between APIs and Security Vulnerabilities: An Ontological Modeling Approach

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*— Over the last decade, a globalization of the software industry took place, which facilitated the sharing and reuse of code across existing project boundaries. At the same time, such global reuse also introduces new challenges to the software engineering community, with not only components but also their problems and vulnerabilities being now shared. For example, vulnerabilities found in APIs no longer affect only individual projects but instead might spread across projects and even global software ecosystem borders. Tracing these vulnerabilities at a global scale becomes an inherently difficult task since many of the existing resources required for such analysis still rely on proprietary knowledge representation.**

In this research, we introduce an ontology-based knowledge modeling approach that can eliminate such information silos. More specifically, we focus on linking security knowledge with other software knowledge to improve traceability and trust in software products (APIs). Our approach takes advantage of the Semantic Web and its reasoning services, to trace and assess the impact of security vulnerabilities across project boundaries. We present a case study, to illustrate the applicability and flexibility of our ontological modeling approach by tracing vulnerabilities across project and resource boundaries.

Keywords*— *Knowledge modeling, ontologies, reasoning, source code analysis, vulnerabilities and patches

I. INTRODUCTION

The globalization of the software industry has been a driving force in replacing traditional project boundaries by promoting a free flow of information, which facilitate reuse and sharing of resources and knowledge across resource boundaries [1] [2]. For example, open-source software (OSS) is published on the Internet using specialized code sharing portals e.g., Sourceforge[1], GitHub[2], to allow for components to be shared, reused and extended by developers around the world. At the same time, such global reuse also introduces new challenges to the software engineering community, with not only components but also their problems and vulnerabilities being now shared.

A recent report in 2012 [1] shows that 88% of the code in today's applications come from OSS libraries and frameworks; with 26% of these OSS frameworks/libraries having known vulnerabilities, which often remain undiscovered. In 2013, "*Using Components with Known Vulnerabilities"* [2] is ranked 9th in the *OWASP Top Ten* [3] list of software security flaws.

Current approaches for ensuring secure software fall in two main categories. The first category requires organizations to create barriers that prevent developers and end-users from performing potential risky actions, e.g., runtime protection. While this approach can reduce the exposure to vulnerabilities, it does not address the fundamental cause of such vulnerabilities. The other category involves techniques that avoid or reduce the introduction of potential vulnerabilities already at the development stage, by introducing and applying best secure coding practices e.g., black-box testing, and static analysis. Unfortunately, most of these analysis techniques are limited to artifacts created within a project context and do not consider in their analysis the reuse and sharing of third party components outside their original development scope.

In our research, we introduce a novel approach for automatically tracing source code vulnerabilities at the API level across project boundaries. More specifically, we take advantage of the Semantic Web and its technology stack (e.g., ontologies, Linked Data, reasoning services) to establish a unified knowledge representation that can link and analyze vulnerabilities across project boundaries. Through this unified representation, we can eliminate information silos that the current analysis approaches have to deal with and introduce new types of vulnerability analysis at a global scale.

In our prior work [4], we introduced our Security Vulnerabilities Analysis Framework (SV-AF), a modeling approach which establishes traceability links between security and software databases. In this research, we extend our previous SV-AF with knowledge from version control systems (VCS) repositories to provide additional analysis services such as: (1) Identifying and tracing the use of vulnerable code in APIs to projects; and (2) provide notifications about vulnerabilities found in APIs (and their dependent component) that can affect a specific project.

Motivating Example: Existing research on recommending APIs to developers (e.g., [5]) has focused on recommending potentially useful APIs to developers to reduce development and testing time.

[1] https://sourceforge.net/

[2] https://github.com/

978-1-5090-6031-3/17 $31.00 © 2017 IEEE
DOI 10.1109/ICST.2017.15

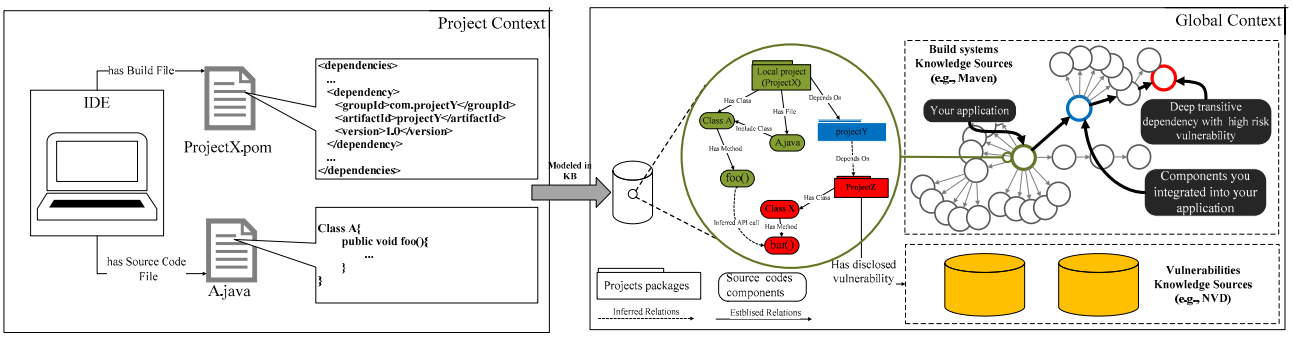

Fig. 1: Integrating code and build information with knowledge from other heterogeneous resources

For example, in [5], the authors explicitly recommend developers to use an older version of Apache Derby (version 10.1.1.0) due to its widespread usage/popularity. However, like any other software project, Apache Derby is also susceptible to security vulnerabilities. By recommending this particular older version of Derby, the author in [5] actually recommended a version of Apache Derby which has two known security vulnerabilities (Table I). These known vulnerabilities had already been published in the National Vulnerability Database (NVD)[3] repository.

TABLE I. SAMPLE DERBY VERSIONS WITH REPORTED VULNERABILITIES

Derby version	Release Year	Reported vulnerabilities in NVD
10.1.1.0	2005	3
10.5.3.0	2009	1

As the example illustrates, the author of the paper was most likely unaware of these reported vulnerabilities since this information is not readily available to developers. Making this information readily available to maintainers and security experts would allow for seamless knowledge integration and sharing. Furthermore, by using standardized and formal knowledge representation techniques (e.g., Semantic Web and its technology stack), novel analysis approaches across knowledge boundaries at both the intra and inter-project level can be introduced.

For example, Fig. 1 shows an example of an IDE with an open Maven[4] POM (*ProjectX.pom*) and Java file (*A.java)*. In our approach, we extend a developer's accessible knowledge from local project's *pom* and *Java* files, to knowledge resources outside the current project boundaries (Fig.1). Using an ontology-based knowledge modeling approach we can now integrate, share and reason upon these heterogeneous resources (even at a global scale). In this example, such a knowledge base includes project-specific resources (e.g., issue tracker, versioning repositories) as well as resources external to the project, such as NVD and Maven build dependencies from other projects. Using the reasoning services provided by the Semantic Web, we can now infer direct and indirect dependencies for the local project (ProjectX in Fig. 1). In addition, giving the bi-directional links in our modeling approach, we can expand our analysis to a global scale to answer questions like this: Which projects might be directly or indirectly affected by a vulnerable component/library? In our example, ProjectX has an indirect dependency on ProjectZ (via ProjectY's transitive dependencies) and makes use of a vulnerable ProjectZ component, using method *X.bar()* within that component.

As our example illustrates, integrating source code information with other knowledge resources (e.g., vulnerability and build repositories) can support for new types of analysis even at a cross-project boundary (global) scale. In addition, these analysis results can now be used to further enrich existing analysis tools. For example, existing tools cannot be extended to not only recommend suitable APIs but to instead recommend now suitable APIs with **no known** direct/indirect vulnerabilities or to automatically notify developers when an already used API **becomes exposed to a potential vulnerability**.

The remainder of this paper is organized as follows: Section II introduces background relevant to our research. Section III explains the methodology we used to instantiate our modeling approach for analyzing APIs vulnerability impacts, followed by Section IV which introduces our case study and its findings. Section V provides a discussion and implications of our findings. Section VI discusses the potential threats to the validity of our approach. In Section VII, we compare our work with related work, followed by Section VIII, which concludes the paper and discusses future work.

II. BACKGROUND

A. Ontologies in Software Engineering

In philosophy, ontology is the science of being [6] and has been adapted by the computing world as "a formal specification of a conceptualization" [7]. Despite ontologies and Knowledge Engineering sharing the same roots, ontologies emphasize aspects such as inter-agent communication and interoperability [8]. In details, an ontology defines a set of primitives to model a domain of knowledge or discourse. This set of representational primitives are typically classes (or sets), attributes (or properties), and relationships (or relations among class members) [9]. An essential aspect of ontologies is that they must be formal and, more precisely, understandable by a computer or "codified in a machine interpretable language" [10].

Ontologies in SE. Representing software in terms of knowledge rather than data, ontologies can be more abstract

[3] https://nvd.nist.gov/
[4] http://search.maven.org/

than, say, database schemata, and provide better support for semantics [6]. With the adoption of Description Logic (DL) as a major foundation of the recently introduced Semantic Web and Web Ontology Language (OWL) [11], there is a trend to utilize ontologies or introduce taxonomies as conceptual modeling techniques into software engineering domain. These existing approaches support knowledge representation and sharing, and automated reasoning. For example, in requirement engineering, ontologies have been used to support requirement management [12], traceability [13], and use case management [14]. In software testing domain, KITSS [15] is a knowledge-based system that can provide assistance in converting a semi-formal test case specification into an executable test script. In software maintenance domain, Ankolekar et al [16] provide ontology to model software, developers, and bugs. Ontologies have also been used to describe the functionality of components using a knowledge representation formalism that allows more convenient and powerful querying. For example, the KOntoR [17] system allows storing semantic descriptions of components in a knowledge base and performing semantic queries on it. In [18], Jin et al. discuss an ontological approach of service sharing among program comprehension tools.

Ontologies vs. Models. A model is "an abstraction that represents some view on reality, necessarily omitting details, and for a specific purpose" [19]. However, in SE, ontologies and models try to address the same problems (representing the software complexity in an abstract manner) but from very different perspectives. The differences between ontologies and models often result in different artifacts, uses, and possibilities. For example, modern SE practices advice developers to look for components that already exist when implementing functionality, since reuse can avoid rework, save money and improve the overall system quality [20]. In this example, ontologies can provide clear advantages over models in integrating information that normally resides isolated in several separate component descriptions. Furthermore, models (e.g., UML) rely on the close world assumption, while ontologies (e.g., OWL) support open-world semantics. OWL, an example of ontology languages, is a "computational logic-based language" that supports full algorithmic decidability in its OWL-DL (description logic) variant. It is not possible to use algorithms supported by OWL (e.g., subsumption) for modeling languages due to their different semantics. Additional differences between ontologies and models are reported and discussed in [21].

B. Source Code Analysis

The applicability of code analysis tools depends on their ability to represent embedded source code semantics and the specific analysis context they are used for. Many analysis tools and techniques have been developed to support specific source analysis contexts (e.g., points-to analysis [23, 24], dependency analysis [25], flow analysis [26], call graph construction [26], program slicing [27], and impact analysis [28]). The level and scope of these analyses techniques vary, ranging from those considering only the behavior of individual statements and declarations to those that include the complete source code of a program. However, common to these techniques is that they aim to provide analysis results that are as complete and precise as possible and exclude in their analysis other software artifacts (e.g., build repository or issue tracker information). Furthermore, given the complexity of software systems [29] many approaches rely on compilable project source code for their analysis, limiting their analysis to project-level scope. A large body of research exists in detecting vulnerabilities at the source code level, using both static and dynamical analysis approaches [19, 23]. In addition to such analysis tool support, public software security vulnerabilities databases have been introduced to create awareness of known vulnerabilities in the source code and to provide a reference to software vendors on how to mitigate these vulnerabilities (e.g., security patches).

C. 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. Such vulnerabilities represent security risks to be exploited by hackers to gain access to system information or capabilities [31]. As discussed in [32] new software vulnerabilities are often first reported in software repositories (e.g., issue trackers, mailing lists) of the affected projects or mentioned on Q&A sites (e.g., StackOverflow). A common characteristic of such early vulnerability reporting is that information about vulnerabilities is dispersed across multiple resources and their descriptions tend to be incomplete, inconsistent and informal. Advisory databases (e.g., NVD) were introduced to address some of these shortcomings. Their objective is not only to provide a central place for reporting vulnerabilities, but also to standardize their reporting. The Common Vulnerabilities and Exposures (CVE)[5] dataset creates a publicly available dictionary for vulnerabilities, allowing for a more consistent and concise use of security terminology in the software domain. Once a new vulnerability is revealed and verified by security experts, this vulnerability and other relevant information (e.g., unique identifier, source URL, vendor URL, affected resources and related vulnerabilities from the same family group) will be added to the CVE database. The source URL refers to the vulnerability (e.g., application vendor, external security advisories) by linking directly to the commit that contains the source code for patching or a document that describes on how to patch the vulnerability. In addition to the CVE entry, each vulnerability will also be classified using the Common Weakness Enumeration (CWE)[6] database. The CWE, therefore, provides a common language to describe software security weaknesses, by classifying them based on their reported weaknesses. NVD, CVE, and CWE can all be considering being part of a global effort to manage the reporting and classification of known software vulnerabilities.

[5] https://cve.mitre.org/

[6] https://cwe.mitre.org/

III. MODELING API VULNERABILITIES

It is generally accepted that inadvertent programming mistakes can lead to software security vulnerabilities and attacks [31]. Mitigating these vulnerabilities can become a major challenge for developers, since not only their own source code might contain exploitable code, but also the code of third-party APIs or external components used by their system. In what follows we introduce a methodology to guide developers in identifying the potential impact of vulnerabilities at both the system and global level (Fig. 2). Our methodology consists of three major steps: knowledge modeling; alignment of ontologies; and knowledge inferences and reasoning.

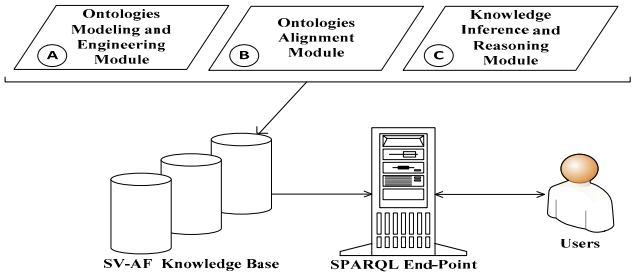

Fig. 2: System overview

A. Knowledge Modeling

A key premise of ontologies is their ability to share and extend existing knowledge. Our approach builds upon this premise, by reusing and extending the integrated software security and engineering ontologies introduced in [4]. In particular, we extend these ontologies through semantic integration (linking) with other repositories (e.g., code repositories, VCS systems). We then further enrich the semantics of our model by not only capturing domains of discourse but also to include semantic relations and properties that allow us to take advantage of inference services provided by the Semantic Web.

For our model, we followed a bottom-up modeling approach, where we first extract system specific concepts and then iteratively abstracted shared concepts in upper ontologies (see Fig 3). The resulting four-layer modeling hierarchy is similar to the metadata modeling approach introduced by the Object Management Group (OMG)[7]. Each of these layers differs in terms of their purpose and their design rationale.

General Concepts layer: Classes in this top-layer represent omnipresent general concepts found in the software evolution and security domain.

Domain-Spanning Concepts layer: This layer represents the concepts that span across a number of subdomains (e.g., security databases, VCS and source code).

Domain-Specific Concepts layer: Concepts in this layer are common across resources in a particular domain (e.g., domain of source code). At the core of the domain specific layer, we have several domain ontologies: (1) Software sEcurity Vulnerability ONTologies (SEVONT), (2) Software Evolution ONtologies (SEON) [33] and (3) Software Build Systems ONtologies (SBSON).

System-Specific Concepts layer**:** Concepts in this layer extend the knowledge from the upper layers through system-specific extensions.

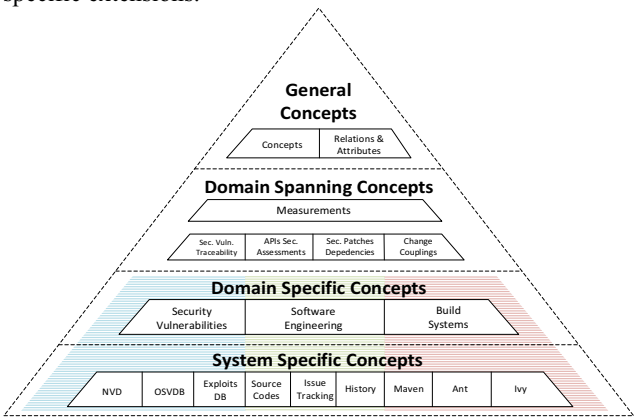

Fig. 3: The SV-AF Ontologies Abstraction Hierarchies [4]

To improve the readability of the paper, 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 [34]. Fig. 4 provides an overview of our knowledge model used for tracing API vulnerabilities. The core concepts used for our vulnerability analysis are *Vulnerabilities*, *SecurityPatches*, and *APIs*. Whenever a *Project* is identified to be affected by a *Vulnerability*, a *SecurityPatch* is developed by its project vendors. A *Committer* commits a new *Version* of a *VersionedFile* containing the security patch through a version system (e.g., SVN). *VersionedFiles* are *Files* managed by a version control system. *Files* are among the Artifacts that are produced when software is created. A project version which is released to the public or customer is referred to as a *BuildRelease* (a *BuildRelease* can dependOn *APIs* from other *BuildReleases*). A *SecurityPatch* corresponds to code changes introduced to fix some existing *VulnerableCode*, which is part of a *CodeEntity*, such as *ComplexType* (i.e., a Class, Interface, Enum, etc.) or a *Method*. For example, if a class or method is modified during a security patch, then this code change can be used to locate the original *VulnerableCode*. The OWL classes, *SecurityPatch* and *VulnerableCode*, are linked in our model through the object property identifies.

B. Ontologies Instances Alignment

For further knowledge integration among the individual ontologies, we take advantage of ontology alignment techniques to establish semantic traceability links. These links allow us to reduce the semantic gap between ontologies and are essential pre-requisites for supporting seamless knowledge integration.

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

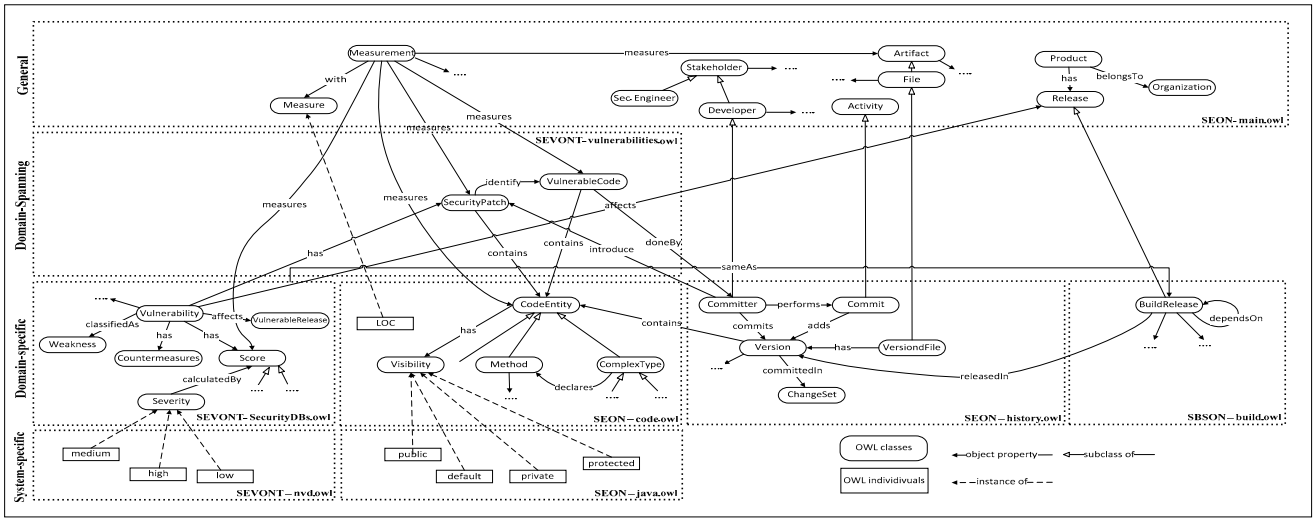

Fig. 4: The SV-AF's [4] ontologies concepts involved in an API

Alignment of SEVONT and SBSON Ontologies. In uncertain graphs [35], edges are associated with uncertainties; it measures the strength of connectivity between nodes and/or edges. An uncertain directed graph is defined as $G = (V, E, \omega)$, where V is a set of nodes, E is a set of edges (x, y), and ω: E → [0, 1] is the weight assignment function (e.g., ω(x, y) = 0.3 means the associated value on edge (x, y) is 0.3). Uncertainty values are interpreted as probabilities.

In our model, the knowledge base is treated as an uncertain graph; where V represents the modeled projects from security vulnerability databases and build repositories, E represents $owl{:}\,sameAs$ relations (edges) between projects' instances, and ω: E → [0,1] is the weight assignment function used by Probabilistic Soft Logic (PSL) framework [36]. For example, in Fig. 5, the project instance V_m from SBSON graph is similar to vulnerable product instance V_n from SEVONT graph through $owl{:}\,sameAs$ ($\omega(e)$) edge.

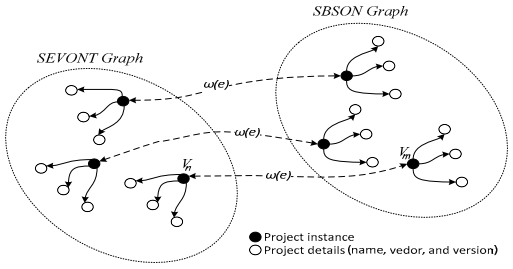

Fig. 5: SV-AF knowledge base similarity graphs

Note that m and n represent the projects original data sources, Maven and NVD respectively. Additional explanation on how the $owl{:}\,sameAs$ weights are created, and how PSL is implemented and tested to establish the sematic links are discussed in[4].

Alignment of SEVONT and SEON Ontology. For this alignment, we extend the process discussed in [4] to include also information from our versioning ontology. Disclosed vulnerabilities often contain references to patch information, such as explicit revisions/commits in which the vulnerability has been fixed. Having this information available, we can perform terminology matching to align instances from both data sources. For the alignment process, we take advantage of reasoning services provided by the Semantic Web to infer implicit relationships between vulnerabilities and commits. More specifically, for the alignment, we take advantage of Semantic Web Rule Language (SWRL)[8] rules (Listing 1) to establish links between vulnerability and commit instances. This alignment will take place if any of the two semantic rules will be satisfied:

Rule 1: Vulnerability ID is explicitly mentioned in a commit message.

Rule 2: Commit/revision ID is explicitly mentioned in the patch reference of a vulnerability.

SWRL rule 1:
$Commit(?\,c), fixNVDIssue(?\,c, ?\,ID),$
$Vulnerability(?\,v), hasVulnerabilityID(?\,v, ?\,ID)$
$\rightarrow vulnerabilityFixedIn(?\,v, ?\,c)$
SWRL rule 2:
$Vulnerability(?\,v), hasPatch(?\,v, ?\,p),$
$hasFixRevision(?\,p, ?\,ID), Commit(?\,c),$
$hasCommitID(?\,c, ?\,ID) \rightarrow vulnerabilityFixedIn(?\,v, ?\,c)$

Listing 1: SWRL rules for aligning CVE facts with the version ontology

Finally, it should be noted that there is no guarantee that any two ontologies in the same domain will align through shared concepts, due to ambiguity or lack of such shared concepts [4].

C. Knowledge Inferencing and Reasoning

The Semantic Web stack includes a scalable, persistent knowledge storage infrastructure. Triple-stores[9] not only

[8] https://www.w3.org/Submission/SWRL/

[9] Triple-store or RDF store is a purpose-built database for the storage and retrieval of triples through semantic queries. A triple is a data entity composed of subject-predicate-object [6].

provide data persistence but also support some basic scalable inference on big data (e.g., RDFS, RDFS++)[4]. In this section, we discuss how we take advantage of such inferences to a.) trace APIs and their vulnerabilities across knowledge boundaries and b.) infer implicit knowledge from these links. It should be noted that we omitted the ontology namespace prefixes (summarized in Table II) from our illustrative queries and rules to improve their readability.

TABLE II: ONTOLOGY NAMESPACES

Namespace	URL
RDF	http://www.w3.org/1999/02/22-rdf-syntax-ns#
OWL	http://www.w3.org/2002/07/owl#
SBSON	http://aseg.cs.concordia.ca/segps/ontologies/domain-spanning/2015/02/build.owl#
SEON	http://se-on.org/ontologies/domain-specific/2012/02/code.owl#
SEVONT	http://aseg.cs.concordia.ca/segps/ontologies/domain-spanning/2015/02/vulnerabilities.owl#

Same-As inference: A commonly used predicate for inferring new knowledge is $owl{:}\,sameAs$, which is used to align two concepts. As we discussed in our prior work [4], having the weighted alignment links between two ontologies, a SPARQL query can now be used to retrieve information across ontology boundaries. We align vulnerability information from the SEVONT ontology and their corresponding instances in SBSON ontology based on a similarity threshold. Using the following SPARQL query (Listing 2), we can now take 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

Transitive closure inference: The transitive closure of a binary relation R on a set of concepts C is the minimal transitive relation R' on C that contains R. Thus $a\ R'b$ for any instances a and b of C provided that there exist $m_0, m_1, \ldots, m_k$ with $m_0 = a$, $m_n = b$, and $m_r\ R\ m_{r+1}$ for all $0 \leq r < k$. The transitive closure $C(G)$ of a graph is a graph which contains an edge $\{u, v\}$ whenever there is a direct path from u to v [37], [38]. However, this can be expressed in OWL through the *owl:TransitiveProperty* construct. We define code:invokesMethod to be a bi-directional transitive property of type *owl:TransitiveProperty* (e.g., code:invokesMethod rdf:type *owl:TransitiveProperty*). Through this transitive construct, we are now able to retrieve a list of all methods that have a direct and transitive invocation dependency to a specified method, and vice versa (see Listing 3).

```
SELECT ?method
WHERE {
   ?method rdf:type code:Method.
   ?method code:invokesMethod <subjectMethodURI> option(transitive).
}
```

Listing 3: SPARQL query returning transitive method calls

Subsumption inference: A crucial aspect of an ontology model is the availability of a subsumption hierarchy between its concepts [39]. For example, a *Method* or *Class* is a sub-concept of a *CodeEntity*. Subsumption hierarchies add significant power to ontologies [29] in global source code (APIs) analysis because many of the attributes of an entity (concept or instance) are attached to its super concepts. Given a set of concepts C, the goal of the inference engine is to discover all subsumption relationships among pairs of concepts in C. More formally, we can denote that concept c_1 is a subconcept of c_2 by $c_1 \sqsubseteq c_2$. Subsumption is directional [39]: if $c_1 \sqsubseteq c_2$, then $c_2 \not\sqsubseteq c_1$ unless c_1 and c_2 are synonyms. A similar subsumption can be inferred from OWL properties that can subsume each other.

In our approach, we create a simple hierarchy of object properties to support such subsumption inference. Fig. 6 shows the property hierarchy we use to model source code dependencies. Given this property hierarchy and the subsumption inference, a simple query (Listing 4) can now identify all code entities that transitively depend on a given code entity independent of their type (property) (e.g., method invocations, interface implementation). Note, subsumption differs from the IsA relationship that typically holds between an instance and a concept (e.g., *ClassX* `IsA` *CodeEntity*).

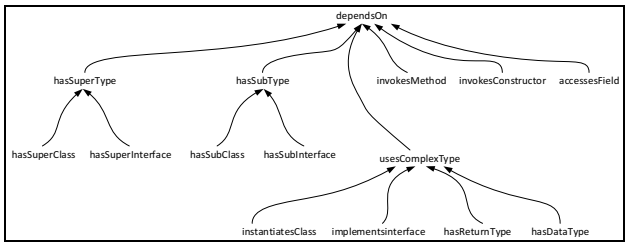

Fig. 6: Hierarchy of code properties

```
SELECT ?entity
WHERE {
   ?entity rdf:type code:CodeEntity.
   ?entity main:dependsOn <subjectEntityURI> option(transitive).
}
```

Listing 4: dependsOn subsumption query

IV. CASE STUDY

In what follows, we discuss the applicability of our modeling approach tin tracing and analyzing known vulnerabilities at intra and inter-project level.

A. Case Study: CVE-2015-0227

Objective: The objective of our case study is to show, how our modeling approach can support the analysis and tracing of potential security vulnerability impacts across software components (APIs). Furthermore, the study also highlights the flexibility of our modeling approach, in terms of its seamless knowledge and analysis result integration, as well as the use of Semantic Web reasoning to infer new knowledge.

Approach: For the case study, we take advantage of *same-as* and *transitive* inferences to identify projects that are directly and indirectly affected by known security vulnerabilities. In addition, we also take advantage of *transitive* and *subsumption* inferences applied at the source code level to identify vulnerable APIs and trace their impact to external dependencies. The inferences consider both,

dependencies within and across software project boundaries (Fig. 7).

Case study setting: We use a publicly disclosed vulnerability, which has been reported in the NVD repository as CVE-2015-0227 and describes the following vulnerability for Apache WSS4J[10]:

"*Apache WSS4J before 1.6.17 and 2.x before 2.0.2 allows remote attackers to bypass the requireSignedEncryptedDataElements configuration via a vectors related to 'wrapping attacks'.*".

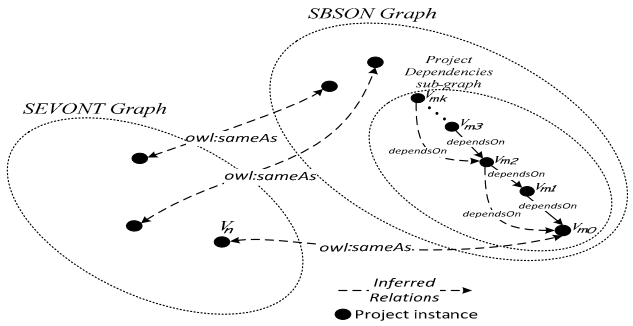

Fig. 7: Inferred project dependencies in SBSON

This vulnerability affects the management of permissions, privileges and other security features that are used to perform access control to Apache WSS4J versions before 1.6.17 and to version 2.x before 2.0.2.

Apache WSS4J is an API which provides a Java implementation of the primary security standards for Web Services and is commonly used by projects as an external component. In this example, a vulnerability is disclosed for this API. Developers using Apache WSS4J in their project have now to determine whether their application is affected by this vulnerability or not. Existing source code analysis tools are capable of identifying, whether a vulnerable code fragment (e.g., code fragment or variable), which has been reported in the NVD vulnerability, is used directly within a project. However, they are not capable of identifying whether the external libraries used by the developer's project might have been affected by this vulnerability.

In what follows, we discuss how our approach takes advantages of originally heterogeneous knowledge resources: NVD, VCS (for only Apache WSS4J), and Maven and integrates these resources to determine direct and indirect dependencies to vulnerable components. In the process, we extract and populate facts from a) NVD: information for the CVE-2015-0227 vulnerability (including patch references); b.) VCS: source code and commit messages for Apache WSS4J (version 1.6.16 and 1.6.17) and c.) Maven repository: all build dependencies on Apache WSS4J 1.6.16 (242 dependencies).

Tracing vulnerability patch information to commit: Security databases provide descriptions of vulnerabilities, their potential effects, and corresponding patches (if applicable). The objective of our study is to establish a traceability link between the unique vulnerability identifier (CVE) and the commit which fixes this vulnerability. For establishing these links, we apply a two-step process, by first mining the NVD repository for patch links that include a reference an entry in a versioning repository. We then extract all commit logs within the versioning repository that have a reference to a CVE-ID. Fig8 shows an example of such a commit log message entry: " *[CVE-2015-0227] Improving required signed elements detection.* "

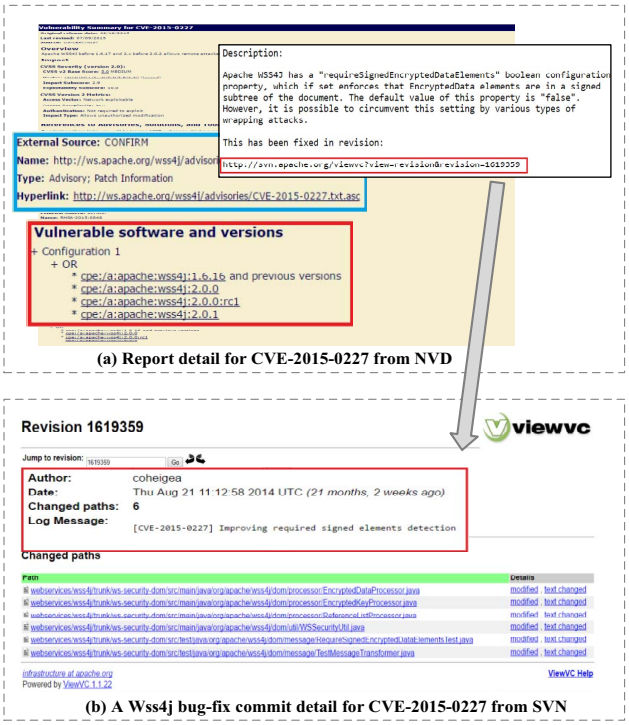

Fig. 8: Extracting patch relevant information from NVD and commit messages

Identify vulnerable code fragments in APIs: A vulnerable code fragment corresponds to a set of lines of code (LoC), which has been modified to fix a vulnerability [40]. In our approach, we use the standard diff command to identify the vulnerable code fragments, by comparing it with its unpatched version. Fig. 9 shows an excerpt of the diff output for WSSecurityUtil.java revisions r1619358 and r1619359. The example shows that method `verifySignedElement` can be identified to contain the vulnerable code fragment. Using the same approach, we can now populate any method or class that has been either deleted or modified as part of a vulnerability fix (commit) in our *sevont:VulnerableCode* class (see Fig. 4).

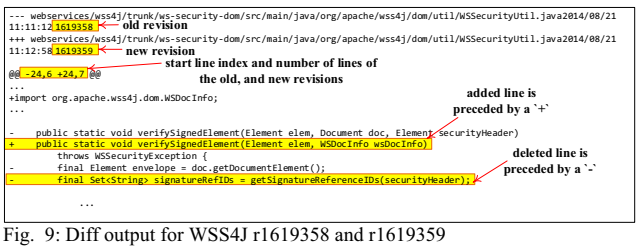

Fig. 9: Diff output for WSS4J r1619358 and r1619359

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

Given our populated ontologies, we can now infer a similarity link between instances of the vulnerable product (e.g., Apache WSS4J 1.6.16) in SEVONT and SBSON (Build repository) and links between the vulnerability patch reference (CVE-2015-0227) and SEON (using the rules in Listing 1) to the commit containing the patch.

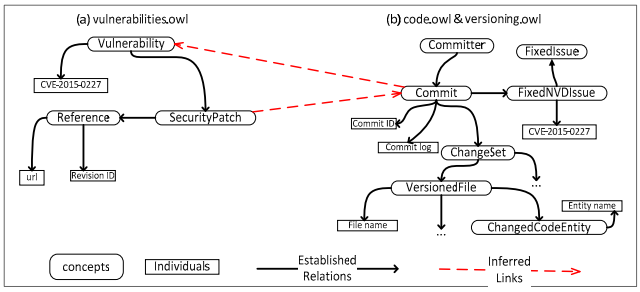

Fig. 10: Inferred links between vulnerabilites.owl, code.owl, and versioning.owl

Based on the inferred links (see Fig. 10) and using the SPARQL query in Listing 5, we can now further restrict our transitive dependency analysis to include only those components that have an actual call dependency to the vulnerable source code.

```
SELECT ?project ?code
WHERE {
    ?project rdf:type <sevont:VulnerableRelease>.
    ?project code:containsCodeEntity ?code.
    ?vulnerableCode rdf:type <code:VulnerableCode>.
    ?code main:dependsOn ?vulnerableCode.
}
```

Listing 5: Query to retrieve vulnerable code fragments across project boundaries

Findings: Table III summarizes the results from our case study for CVE-2015-0227. We report on the manually verified results obtained from executing our SPARQL queries (Listings 4 and 5). Table III shows that 15 of the 242 crawled dependent projects actually use the API from our vulnerable project. The results highlight that there are still systems (6.19%) that rely on libraries with known security vulnerabilities. Moreover, 10 of these 15 dependent projects not only include the API but also call the class *WSSecurityUtil*, which contains the vulnerable code. However, it should be noted that for our specific case study, none of the projects actually called and executed the vulnerable method (`verifySignedElement`) within the *WSSecurityUtil*.

TABLE III: RESULTS

Project	Crawled Dependencies	Actual usage	Vuln. Classes usage	Vuln. Methods usage
Apache WSS4J 1.6.16	**242**	**15**	**10**	**0**

In order to evaluate if our approach is capable of correctly identifying calls to vulnerable methods, we conducted an additional controlled experiment. For this experiment, we manually seeded a method call in Apache CXF-bundle 2.6.15 that invokes the vulnerability in *Apache WSS4J* API. More specifically, we downloaded the source code for Apache CXF-bundle 2.6.15 and modified its `org.apache.cxf.ws.security.wss4j.policyhandlers` package. Fig. 11 shows the partial class diagram of the modified packages. We modified the `includeToken` method of the `AbstractBindingBuilder` class to include a direct call to the vulnerable `WSSecurityUtil.verifySignedElement` method. We also added the `SVAFSymmetricBindingHandler` and `SVAFAsymmetricBindingHandler` to extend `SymmetricBindingHandler` and `AsymmetricBindingHandler` to be able to see if our approach also supports the transitive call dependency analysis correctly. We then re-populate the source code ontologies with the new (modified) code facts and invoke again the same query we used earlier in the case study.

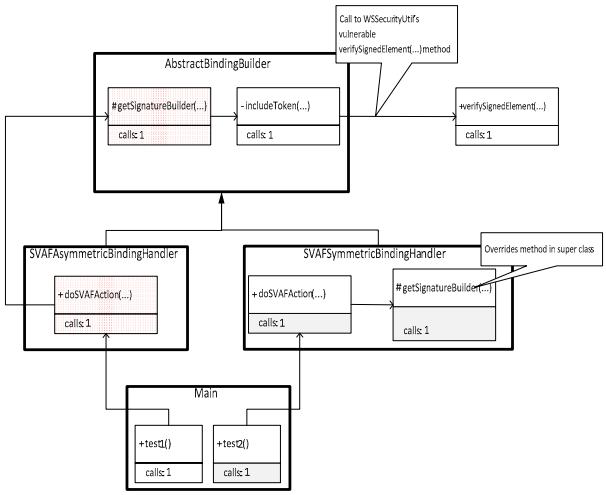

Fig. 11: Class diagram for our modified package

The results of this query are shown in Table IV, which includes the classes within our modified project that directly or indirectly invoke the vulnerable method `WSSecurityUtil.verifySignedElement`.

TABLE IV: RESULTS OF DIRECT AND INDIRECT USAGE OF THE VULNERABLE METHOD WSSecurityUtil.verifySignedElement

Class	# Indirect Vulnerable Methods	Indirect Vulnerable Methods
AbstractBindingBuilder.java	**4**	handleSupportingTokens(.SupportingToken,boolean, Map, Token, Object)
		getSignatureBuilder(TokenWrapper, Token, boolean, boolean)
		getSignatureBuilder(TokenWrapper, Token, boolean)
		doSVAFAction()
Main.java	**1**	test1()

For sake of simplicity and readability, we only include public and protected methods in the result set. We observed that the vulnerability introduced in `AbstractBindingBuilder.includeToken` propagates through several methods. More specifically, the `doSVAFAction` method in this example is indirectly affected due to its usage of the `getSignatureBuilder` method. `SVAFAsymmetricBindingHandler` extends `AbstractBindingBuilder` and overrides the `getSignatureBuilder` method. When the method

`doSVAFAction` is invoked from `test2`, the overridden method from subclass `SVAFAsymmetricBindingHandler` is called and method `test2` is correctly identified by our approach as not being affected by the vulnerability.

B. Comparison against existing tools

We further evaluated our approach, by comparing it against existing tools that detect known security vulnerabilities in source code across project boundaries. For our comparison, we consider the following tools: OWASP Dependency-Check (DC) [41], which is an open source tool, and a closed-source tool from SAP labs [30].

OWASP DC performs a static dependency analysis to determine if libraries with known vulnerabilities are included in an application. During the analysis, the tool collects information about the vendor, product, and version. The information is then used to identify the Common Platform Enumeration (CPE). If a CPE is identified, a listing of associated Common Vulnerabilities and Exposure (CVE) entries are reported.

```
<entry id=" CVE-2016-9878 ">
...
  <vuln:vulnerable-software-list>
    <vuln:product> cpe:/a:pivotal_software:spring_framework:3.2.2 </vuln:product>
    <vuln:product> cpe:/a:pivotal_software:spring_framework:3.2.3 </vuln:product>
    <vuln:product> cpe:/a:pivotal_software:spring_framework:3.2.4 </vuln:product>
    ...
```

The SAP tool relies on a dynamic source code level analysis to identify if a vulnerable piece of code is either used directly or indirectly. The tool uses execution traces which are collected after instrumenting the code and all bundled libraries. Since we did not have direct access to the SAP tool, we replicated their experiments to compare our results with the ones reported in [30].

Given that the OWASP DC tool does distinguish whether a vulnerable library code is used or not, we limit our comparison to: "id*entify if a project depends on libraries with disclosed vulnerabilities independent of the use of the vulnerable source code"*. Table V reports the results from our comparison, which includes true positives (TP), false negatives (FN), false positives (FP) and true negatives (TN). The results show that for CVE-2013-2186, both, our approach and OWASP DC, did not report the vulnerable API. This miss is due to the fact that NVD did not include FileUpload 1.2.2 in the list of affected products. The vulnerability, however, is reported in several JBoss projects, which make use of the DiskFileItem class in Apache FileUpload. Our approach currently models only products explicitly mentioned to be affected in NVD.

OWASP DC reported CVE-2014-9527 as a vulnerability in POI 3.11 Beta 1. A manual inspection of the patch showed that the class "org.apache.poi.hslf.HSLFSlideShow" contains the patch for the vulnerable code but is not used by "poi-3.11.beta1.jar". Instead, this patch is distributed as part of the POI-HSLF component.

For the vulnerability CVE-2013-0248, the patch is located in the default configuration file "using.xml" and the comment of the Java class "DiskFileItemFactory" (but not any executable code). As a result, the SAP tool does not identify the archive as being affected by vulnerable code.

TABLE V: COMPARISON OF ANALYSIS RESULTS

Vulnerability	Library	Our Approach	SAP tool	OWASP DC
CVE-2014-0050	Apache FileUpload 1.2.2	TP	TP	TP
CVE-2013-2186		**FN**	TP	**FN**
CVE-2013-0248		TP	**FN**	TP
CVE-2012-2098	Apache Compress 1.4	TP	TP	TP
CVE-2014-3577	Apache HttpClient 4.3	TP	TP	TP
CVE-2014-9527	Apache POI 3.11 Beta 1	TN	TN	**FP**
CVE-2014-3574		TP	TP	TP
CVE-2014-3529		TN	TN	TN

V. IMPLICATIONS

As our case study illustrates, our ontology-based knowledge modeling approach can integrate information originating from different heterogeneous knowledge resources. In what follows, we discuss how our approach overcomes a number of challenges identified with both OWASP and SAP tools.

Data integration challenges. Vulnerability and dependency management make use of different naming schemes and nomenclatures: There exist many language-dependent approaches for referencing entities, making the linking of entities across knowledge resources often a difficult task. Consider the following example: Mapping the Spring Core 4.0.3.RELEASE between Maven and NVD. Maven GAV identifier represents this component as *groupId*=org.springframework; *artifactId*=spring-core; *version*=4.0.3.RELEASE. While the CPE for the same component in NVD is: *vendor=pivotal; product=spring_framework; version=4.03*

As a result of this identifier naming inconsistency, the automatic mapping between GAV identifiers in Maven with their corresponding CPE in NVD becomes a major challenge e.g., the vendor in our example should be `Pivotal` and not `springframework`. While a human can easily recognize the correct mapping, this is not the case for an automated solution. Both OWASP DC and the SAP tool compute the SHA-1 of the archives and perform a lookup in Maven central to address this problem. While this approach improves the recall (number of correct mappings found), it also introduces many false positives and false negatives, which affects the accuracy of these tools. Moreover, both tools are limited in their ability to match vulnerabilities and CPEs, making them not only prone to errors but also limit the scope of the analysis to direct dependencies. In contrast, our approach addresses these challenges by taking advantage of the PSL alignment framework. This eliminates the need for one-to-one assignments and establishes weighted links between instances of different modeled ontologies for different data sources. Moreover, our semantic approach takes advantage of semantic reasoning to infer transitive dependencies.

Flexibility. While the use of run-time information (traces) can improve the precision (SAP tool), this type of analysis depends on the quality and coverage achieved by these traces. Furthermore, the SAP tool focuses on intra-project analysis,

whereas our approach also supports inter-project analysis. As we further show in our case study, by taking advantage of automated reasoning we are able to infer sub-properties (subsumption) and transitive closure dependencies. Using these inferences, we can transform often complex and proprietary source code analysis tasks to simpler and easy to write SPARQL queries. For example, the isSubClassOf, isSubInterfaceOf, invokesMethod, and invokesConstructor are all sub-properties of the transitive dependsOn property. As such, a simple query (Listing 5), can now identify all code entities that transitively depend on a given vulnerable code entity independent of the type, method invocations or inherited classes/interfaces (via subsumption). As we showed in our controlled study, vulnerable classes can create a backdoor (e.g., through inheritance) for the invocation of vulnerable methods, if these methods are not overridden within the client. With the growing popularity of using 3rd-party APIs [42], the risk of such transitive vulnerable method invocations increases.

Information silos challenges. Although both analysis tools, SAP and OWASP DC are linking different data sources, these resources still remain information silos. They still lack the standardization, knowledge sharing and analysis result integration required to make them true information hubs. In contrast, our approach introduces a unified standardized representation using ontologies, which support seamless knowledge integration, interoperability and sharing even on a global scale. RDF based triple-stores ensure not only persistence of the data but also provide scalability and the use unique resource identifiers (URIs), eases the integration with other knowledge resources, even at a global scale.

VI. Threats to Validity

An internal threat to validity is that our case study relies on our ability to mine facts from both, the Maven and NVD repositories to populate our ontologies. A common problem then mining software repositories is that repositories often contain noise in their data, due to data ambiguity, inconsistency or incompleteness. We are able to mitigate this threat since vulnerabilities published in NVD are manually validated and managed by security experts and therefore make 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, limiting not only ambiguities and inconsistency at the project build but also for the complete dataset. Regarding the knowledge obtained from NVD, not all identified vulnerabilities include complete references to the actual source commit of patches, limiting our ability to automatically extract the source code information related to such a particular patch.

In terms of external threats, the presented experiments might not be generalizable for non-MAVEN projects. This threat is mitigated by our modeling approach with its different abstraction layers. More specifically, we extract and model domain-specific ontologies (e.g. build ontology), which share concepts and their relations that are common this domain (e.g., the domain of build repositories). Another external threat to validity for our research is that for our evaluations we relied on quantitative analysis, 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

Several approaches for static vulnerability analysis and detection in source code exist (e.g., [24], [26], [43], and [23]). Plate et. al [43] proposed a technique that supports the impact analysis of vulnerability based on code changes introduced by security fixes. Their approach relies on a dynamic analysis to determine if a vulnerable code was executed within a given project. In contrast, while less precise in some cases, we provide a more holistic approach, which not only considers all possible executions but also supports a more general intra and inter-project dependency analysis. We also take advantage of semantic reasoning services to infer implicit facts about the vulnerable code usages within the system, to support bi-directional dependency analysis – including both impacts to external dependencies and vice versa.

Nguyen et. al [44] proposed an automated method to identify vulnerable code based on older releases of a software system. Their approach scans the code base of each prior version for code containing vulnerable code fragments. In contrast, our approach takes advantage of multiple knowledge resources, providing a greater flexibility in the analysis.

Mircea et al. [45] introduce in their Vulnerability Alert Service (VAS) an approach that notifies users if a vulnerability is reported for software systems. VAS depends on the OWASP Dependency-Check tool. VAS reports the vulnerable projects identified by the OWASP tool and therefore also lacks the support for transitive dependencies analysis of vulnerable components.

VIII. Conclusion and Future Work

This paper presented an ontological-based modeling approach that allows us to trace API security impact within application boundaries and its global dependencies. Using multi-layers of abstraction, our modeling approach can not only provide a generic analysis approach but also supports the seamless integration of other knowledge resources in the software engineering domain. This formal knowledge representation allows us to take advantage of inference services provided by the Semantic Web, providing additional flexibility compared to traditional proprietary analysis approaches.

As part of our future work, we will further extend our knowledge base to include other vulnerability and software engineering knowledge resources. We will develop an Eclipse plugin, to include a software developer's context in the dependency analysis to further improve the relevance of analysis results.

REFERENCES

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

[2] OWASP, "Using Components with Known Vulnerabilities," 2013. [Online]. Available: https://www.owasp.org/index.php/Top_10_2013-A9-Using_Components_with_Known_Vulnerabilities. [Accessed: 23-Sep-2016].

[3] OWASP, "Top 10," 2013. [Online]. Available: https://www.owasp.org/index.php/Top_10_2013-Top_10. [Accessed: 23-Sep-2016].

[4] S. S. Alqahtani, E. E. Eghan, and J. Rilling, "SV-AF – A Security Vulnerability Analysis Framework," in *IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, 2016.

[5] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining trends of library usage," in *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, 2009, pp. 57--62.

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

[7] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowl. Acquis.*, vol. 5, no. 2, pp. 199–220, Jun. 1993.

[8] M. Uschold and M. Gruninger, "Ontologies: principles, methods and applications," *Knowl. Eng. Rev.*, vol. 11, no. 2, p. 93, Jun. 1996.

[9] O. Corcho, M. Fernández-López, and A. Gómez-Pérez, "Ontological Engineering: Principles, Methods, Tools and Languages," in *Ontologies for Software Engineering and Software Technology*, Springer Berlin Heidelberg, pp. 1–48.

[10] F. Ruiz and J. R. Hilera, "Using Ontologies in Software Engineering and Technology," in *Ontologies for Software Engineering and Software Technology*, Springer Berlin Heidelberg, pp. 49–102.

[11] F. Baader, I. Horrocks, and U. Sattler, "Description Logics as Ontology Languages for the Semantic Web," in *Mechanizing Mathematical Reasoning*, 2005, pp. 228–248.

[12] B. Decker, J. Rech, E. Ras, B. Klein, and C. Hoecht, "Selforganized Reuse of Software Engineering Knowledge Supported by Semantic Wikis," in *Proceedings of the Workshop on Semantic Web Enabled Software Engineering (SWESE)*, 2005, p. 76.

[13] Y. Zhang, J. Rilling, and V. Haarslev, "An Ontology-based Approach to Software Comprehension - Reasoning about Security Concerns," in *Computer Software and Applications Conference, 2006. COMPSAC'06. 30th Annual International*, 2006, pp. 333--342.

[14] B. Wouters, D. Deridder, and E. Van Paesschen, "The use of ontologies as a backbone for use case management," in *European Conference on Object-Oriented Programming (ECOOP 2000), Workshop: Objects and Classifications, a natural convergence*, 2000.

[15] U. Nonnenmann and J. K. Eddy, "KITSS-a functional software testing system using a hybrid domain model," in *Proceedings Eighth Conference on Artificial Intelligence for Applications*, pp. 136–142.

[16] A. Ankolekar, K. Sycara, J. Herbsleb, R. Kraut, and C. Welty, "Supporting online problem-solving communities with the semantic web," *Proc. 15th Int. Conf. World Wide Web - WWW '06*, p. 575, 2006.

[17] H. Hans-Jörg, A. Korthaus, S. Seedorf, and P. Tomczyk, "KOntoR: An Ontology-enabled Approach to Software Reuse," in *Proceedings of 18th International Conference on Software Engineering and Knowledge Engineering*, 2006.

[18] D. Jin and J. R. Cordy, "A service sharing approach to integrating program comprehension tools," in *Proceedings of the European Software Engineering Conference, Helsinki, Finland*, 2003.

[19] B. Henderson-Sellers, "Bridging metamodels and ontologies in software engineering," *J. Syst. Softw.*, vol. 84, no. 2, pp. 301–313, Feb. 2011.

[20] R. Witte, Y. Zhang, and J. Rilling, "LNCS 4519 - Empowering Software Maintainers with Semantic Web Technologies," pp. 37–52.

[21] and K. K. A. C. M. Gutheil, "On the Relationship of Ontologies and Models," in *Proceedings of the 2nd International Workshop on Meta-Modelling (WoMM)*, 2006, pp. 47–60.

[22] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for Java," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 1, pp. 1–41, Jan. 2005.

[23] M. Hirzel, D. Von Dincklage, A. Diwan, and M. Hind, "Fast online pointer analysis," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 2, p. 11–es, Apr. 2007.

[24] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: a clustering tool for the recovery and maintenance of software system structures," in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). "Software Maintenance for Business Change" (Cat. No.99CB36360)*, 1999, pp. 50–59.

[25] M. P. Robillard, "Topology analysis of software dependencies," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 4, pp. 1–36, Aug. 2008.

[26] J.-D. Choi, M. Burke, and P. Carini, "Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects," in *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '93*, 1993, pp. 232–245.

[27] M. Weiser, "Program Slicing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984.

[28] A. Rountev, S. Kagan, and M. Gibas, "Static and dynamic analysis of call chains in java," *ACM SIGSOFT Softw. Eng. Notes*, vol. 29, no. 4, p. 1, Jul. 2004.

[29] D. Binkley, "Source Code Analysis: A Road Map," in *Future of Software Engineering (FOSE '07)*, 2007, pp. 104–119.

[30] H. Plate, S. E. Ponta, and A. Sabetta, "Impact assessment for vulnerabilities in open-source software libraries," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 411–420.

[31] A. Williams, Jeff and Dabirsiaghi, "The unfortunate reality of insecure libraries," *Asp. Secur. Inc*, pp. 1--26, 2012.

[32] N. McNeil, R. A. Bridges, M. D. Iannacone, B. Czejdo, N. Perez, and J. R. Goodall, "PACE: Pattern Accurate Computationally Efficient Bootstrapping for Timely Discovery of Cyber-security Concepts," in *2013 12th International Conference on Machine Learning and Applications*, 2013, pp. 60–65.

[33] M. Würsch, G. Ghezzi, M. Hert, G. Reif, and H. C. Gall, "SEON: a pyramid of ontologies for software evolution and its applications," *Computing*, vol. 94, no. 11, pp. 857–885, Nov. 2012.

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

[35] M. Potamias, F. Bonchi, A. Gionis, and G. Kollios, "k-nearest neighbors in uncertain graphs," *Proc. VLDB Endow.*, vol. 3, no. 1–2, pp. 997–1008, Sep. 2010.

[36] 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.

[37] A. V. Aho, M. R. Garey, and J. D. Ullman, "The Transitive Reduction of a Directed Graph," *SIAM J. Comput.*, vol. 1, no. 2, pp. 131–137, Jun. 1972.

[38] S. Skiena, *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Addison-Wesley, 1990.

[39] D. Movshovitz-Attias, S. E. Whang, N. Noy, and A. Halevy, "Discovering Subsumption Relationships for Web-Based Ontologies," in *Proceedings of the 18th International Workshop on Web and Databases - WebDB'15*, 2010, pp. 62–69.

[40] V. H. Nguyen, S. Dashevskyi, and F. Massacci, "An automatic method for assessing the versions affected by a vulnerability," *Empir. Softw. Eng.*, Dec. 2015.

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

[42] Y. Mileva, V. Dallmeier, and A. Zeller, "Mining API popularity," *Testing--Practice Res. Tech.*, pp. 173–180, 2010.

[43] H. Plate, S. E. Ponta, and A. Sabetta, “Impact assessment for vulnerabilities in open-source software libraries,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 411–420.

[44] V. H. Nguyen, S. Dashevskyi, and F. Massacci, “An automatic method for assessing the versions affected by a vulnerability,” *Empir. Softw. Eng.*, Dec. 2015.

[45] 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.